

Perflso: Performance Isolation for Commercial Latency-Sensitive Services

Călin Iorgulescu
EPFL

Reza Azimi
Brown University

Youngjin Kwon
University of Texas

Sameh Elnikety

Manoj Syamala
Microsoft Research

Vivek Narasayya

Herodotus Herodotou
Cyprus University of Technology

Paulo Tomita

Alex Chen

Jack Zhang

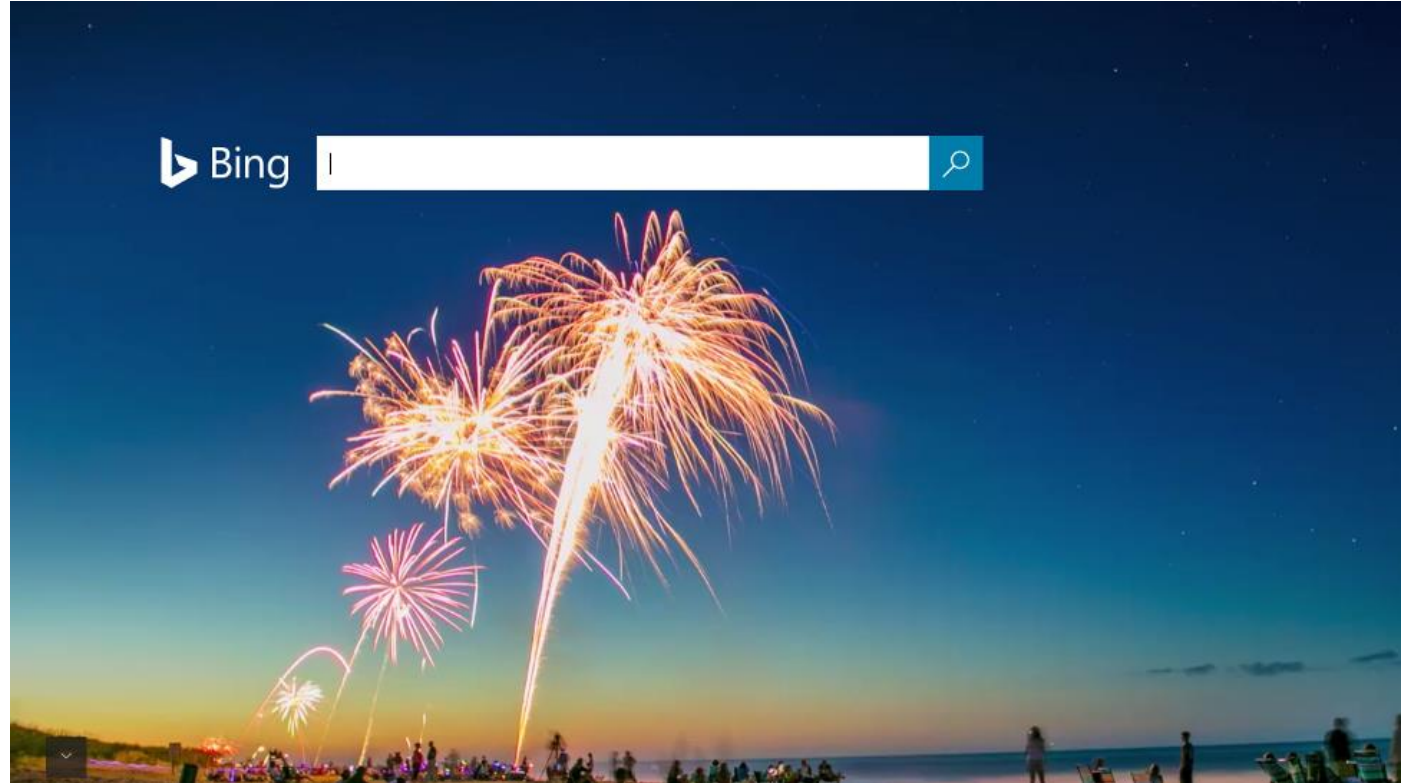
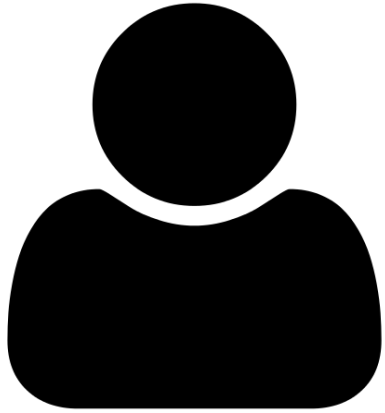
Junhua Wang

Microsoft Bing

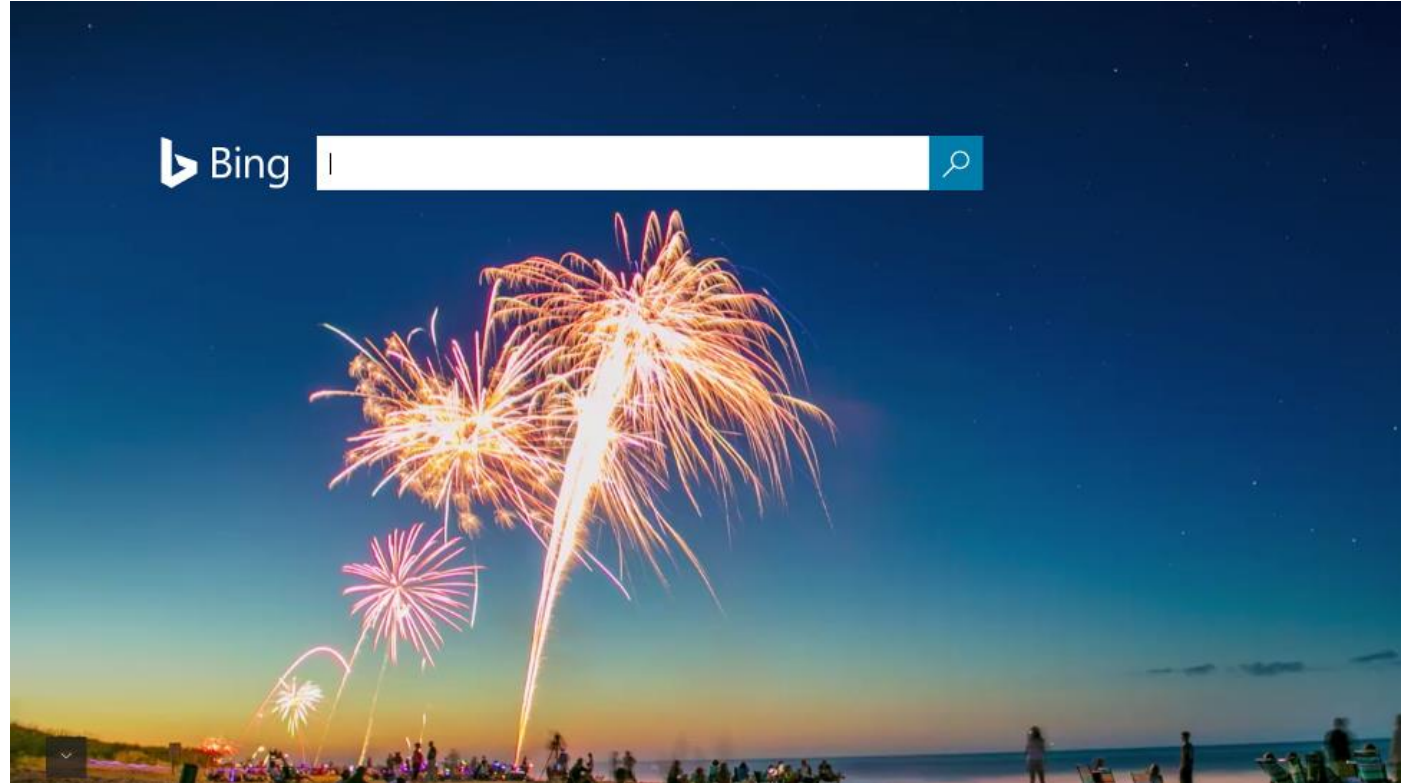
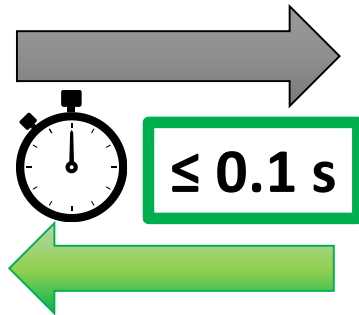
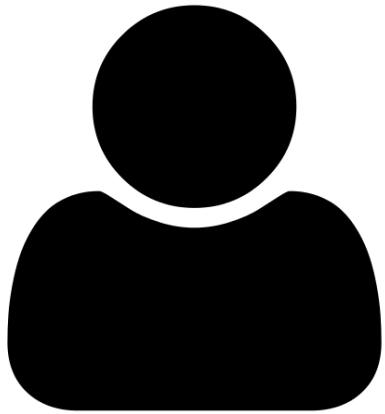


Interactive services must feel instantaneous

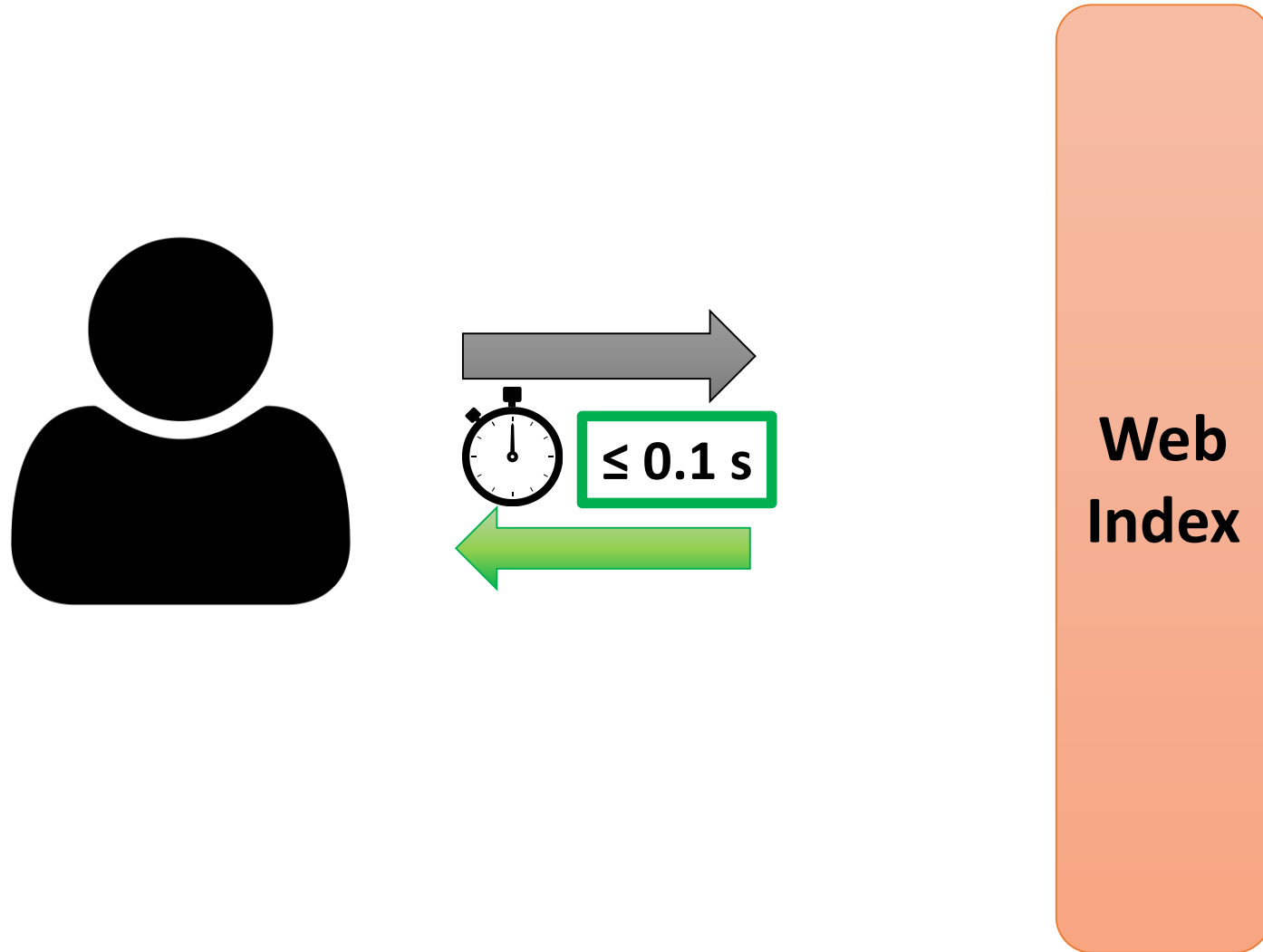
Interactive services must feel instantaneous



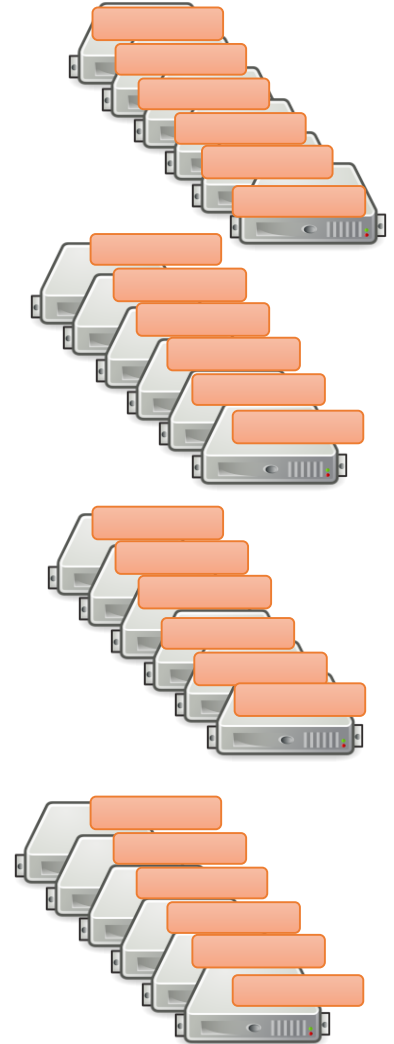
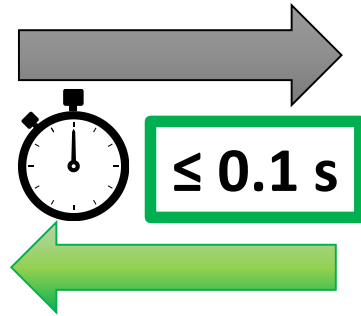
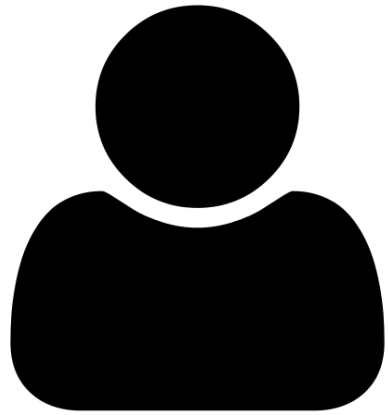
Interactive services must feel instantaneous



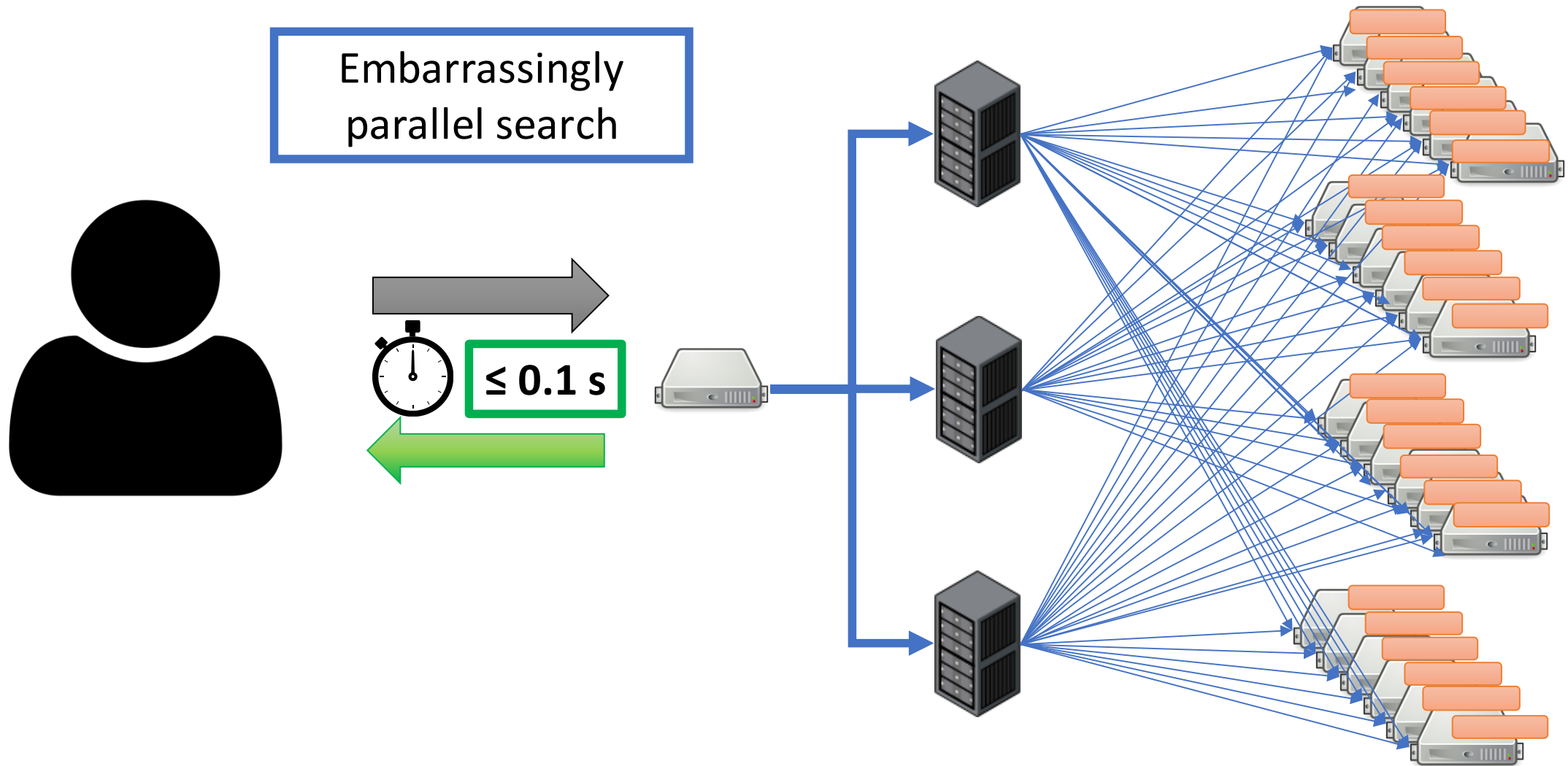
A single query involves hundreds of machines!



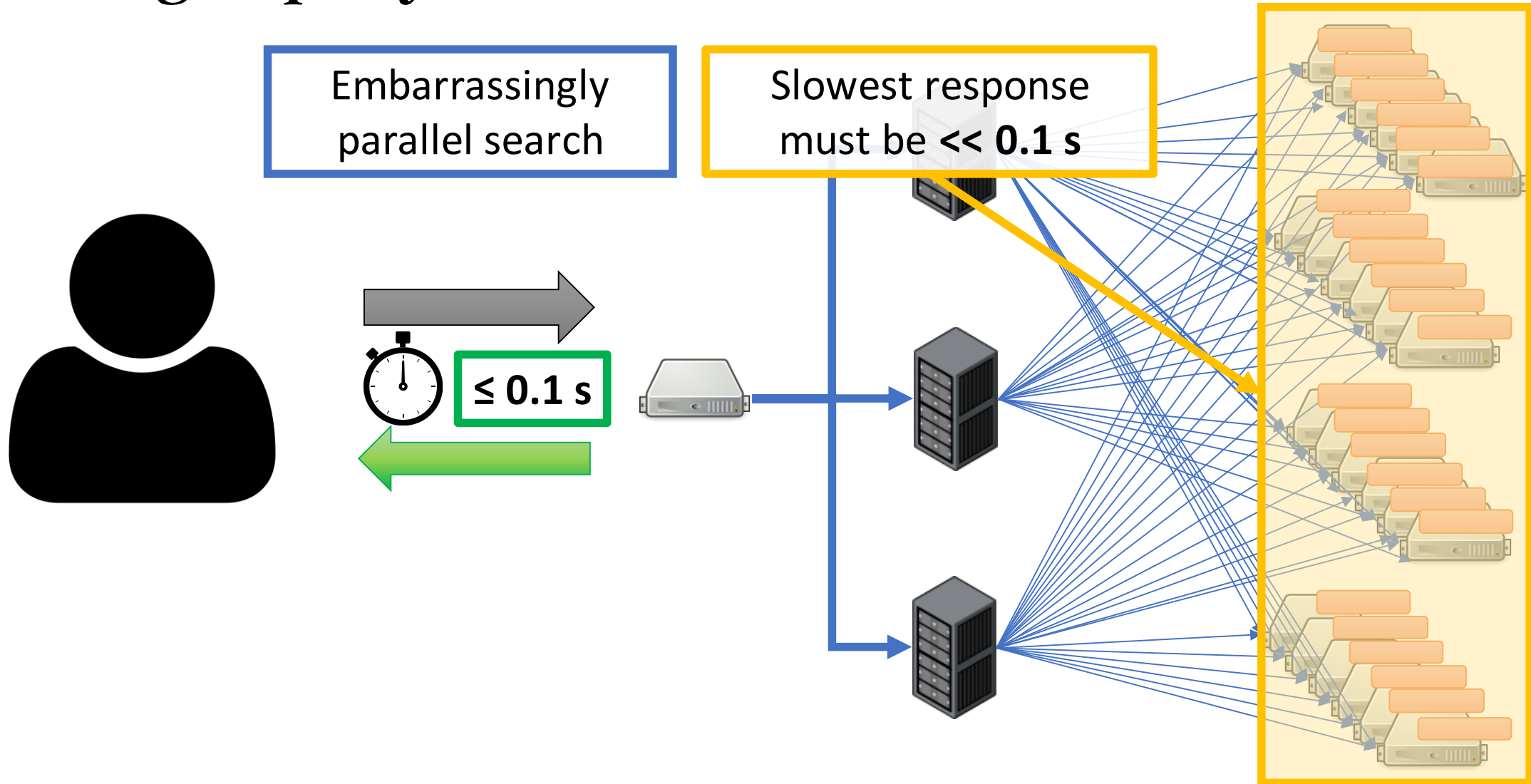
A single query involves hundreds of machines!



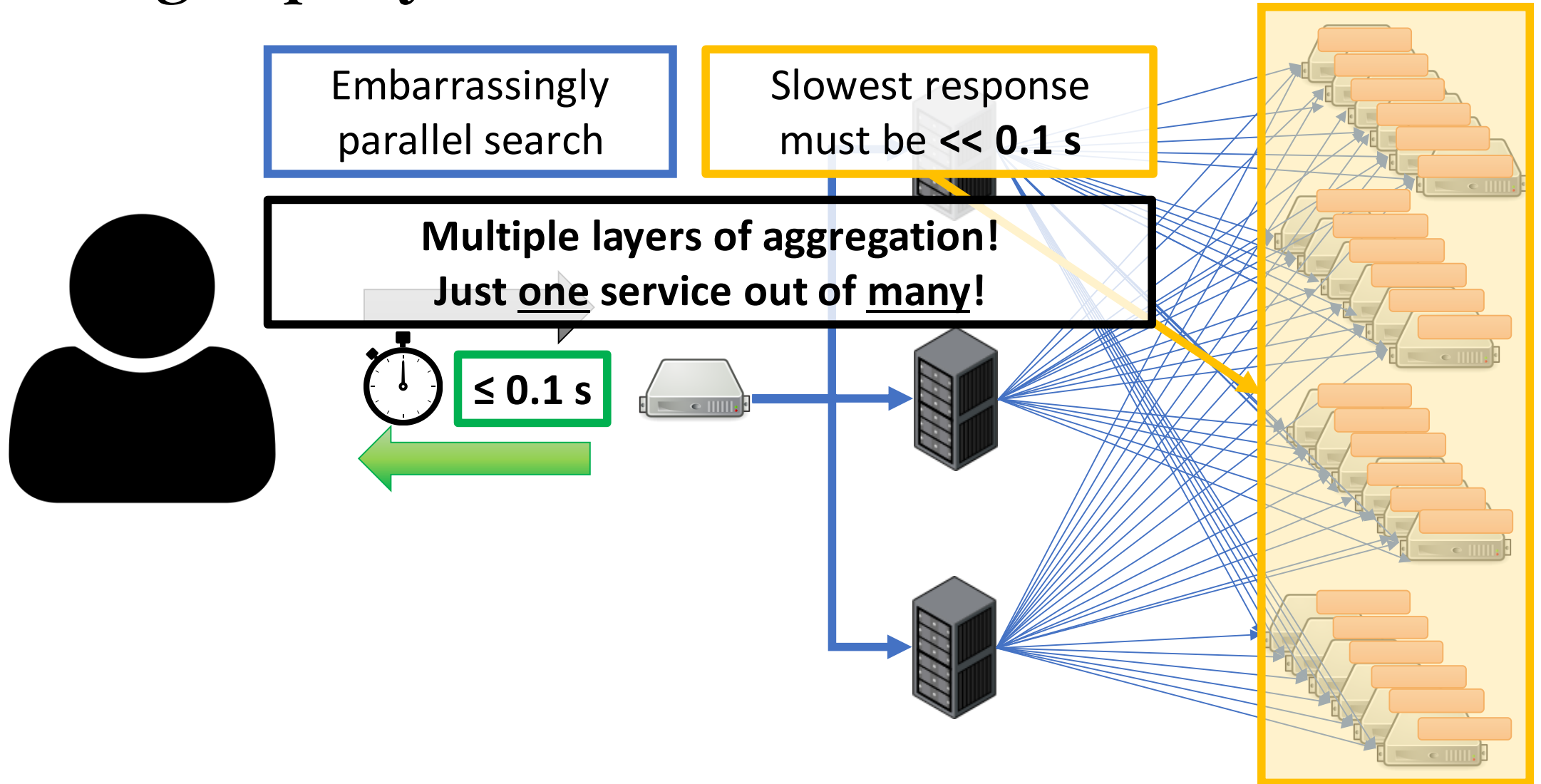
A single query involves hundreds of machines!



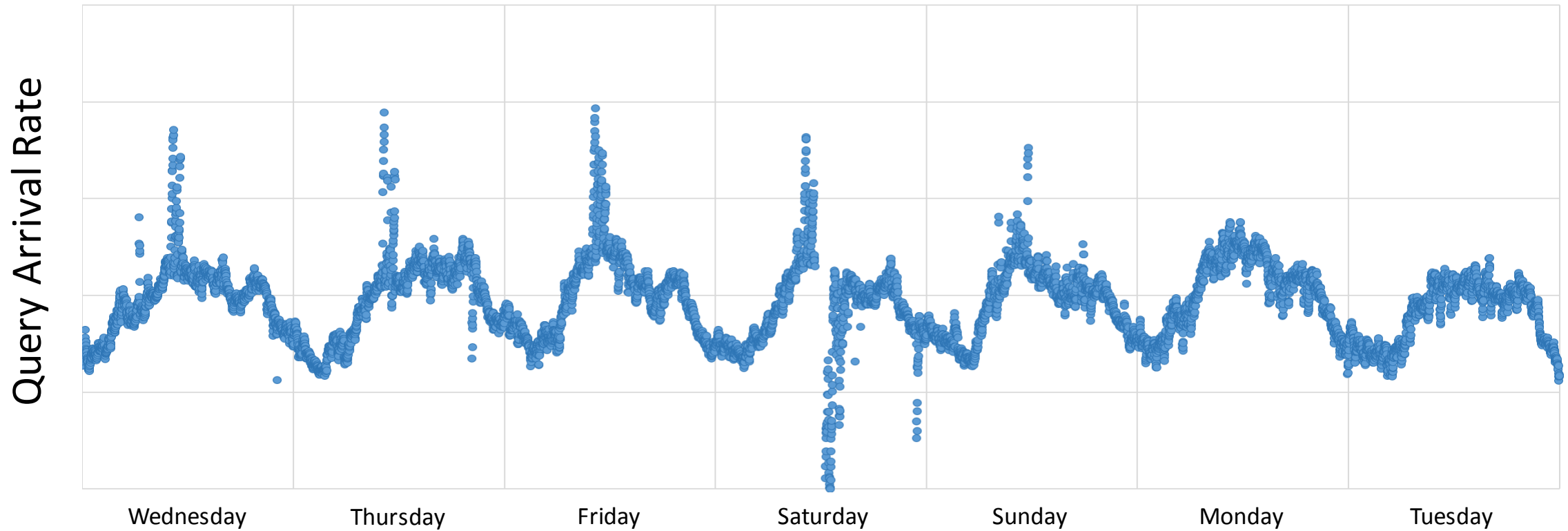
A single query involves hundreds of machines!



A single query involves hundreds of machines!

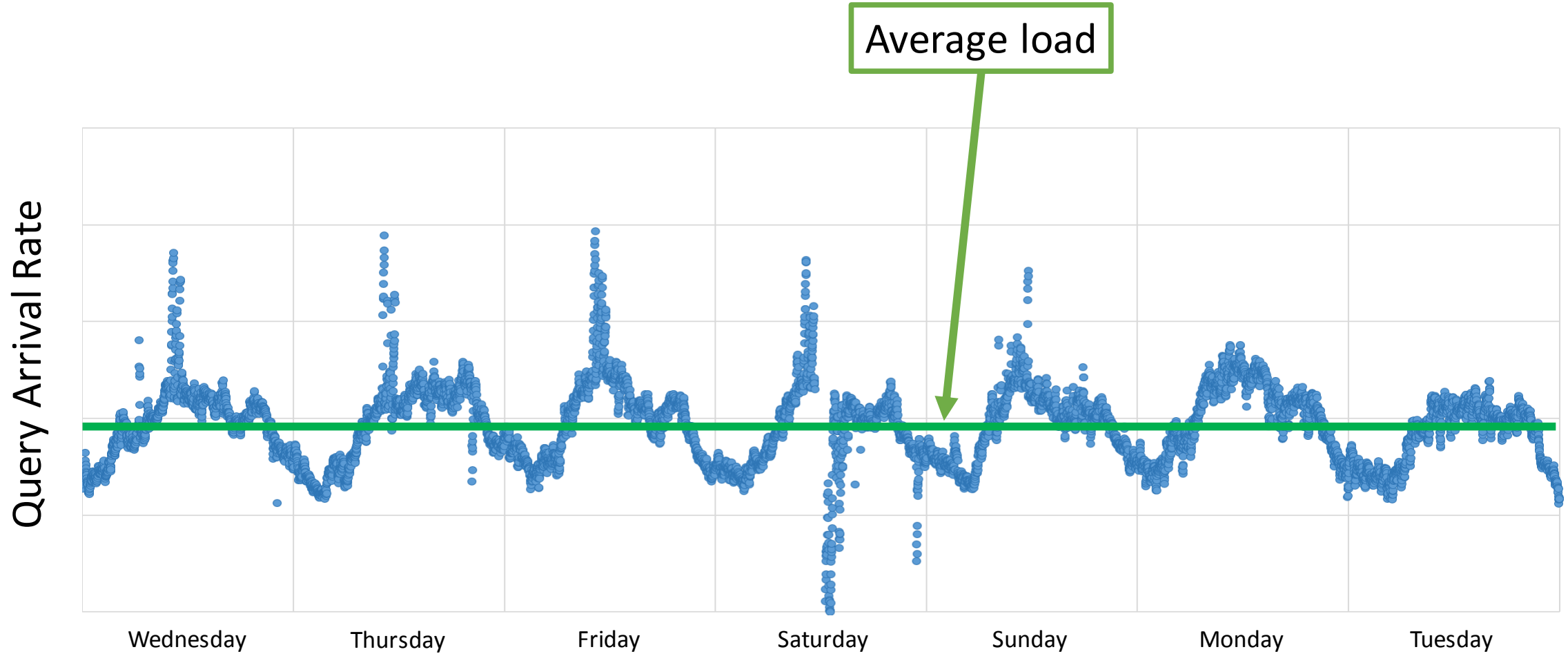


Machines are provisioned for peak load



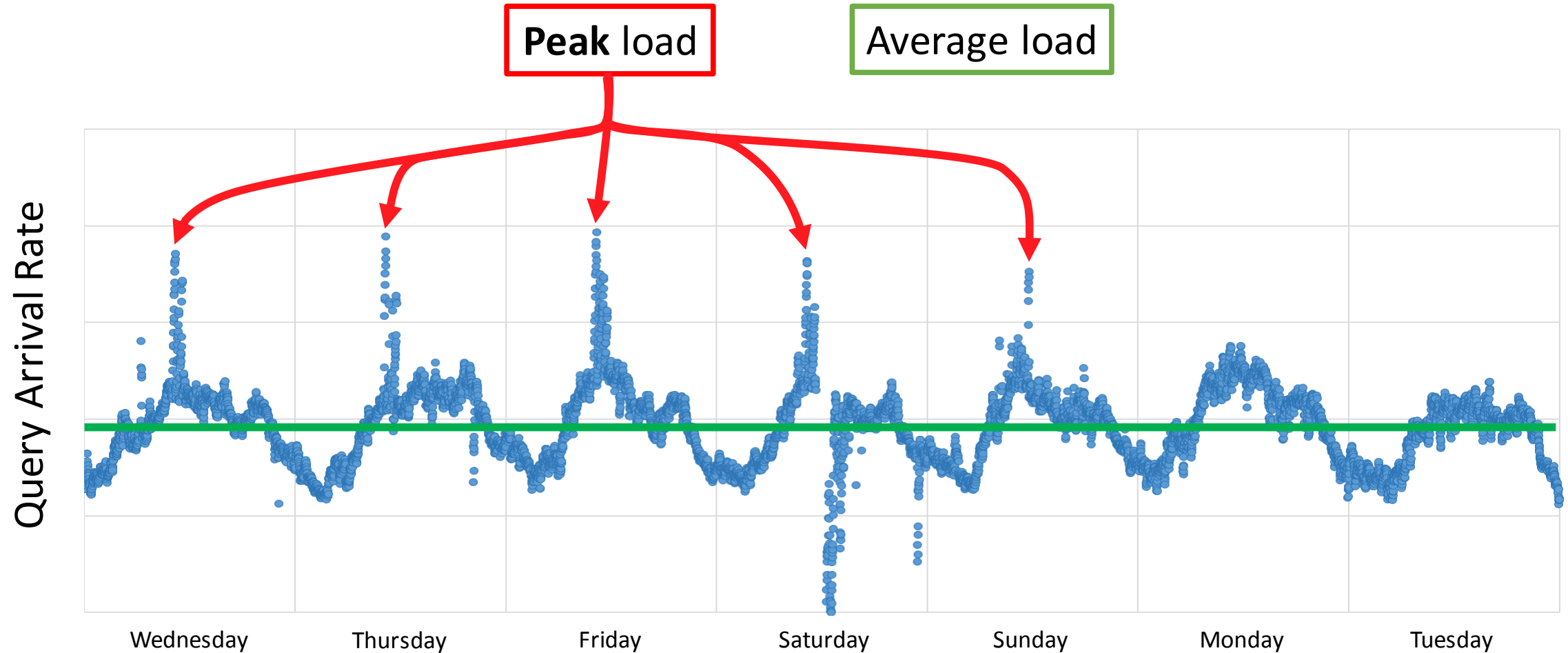
Request-rate variation for a *Microsoft Bing* sub-cluster over 1 week in 2017

Machines are provisioned for peak load



Request-rate variation for a *Microsoft Bing* sub-cluster over 1 week in 2017

Machines are provisioned for peak load



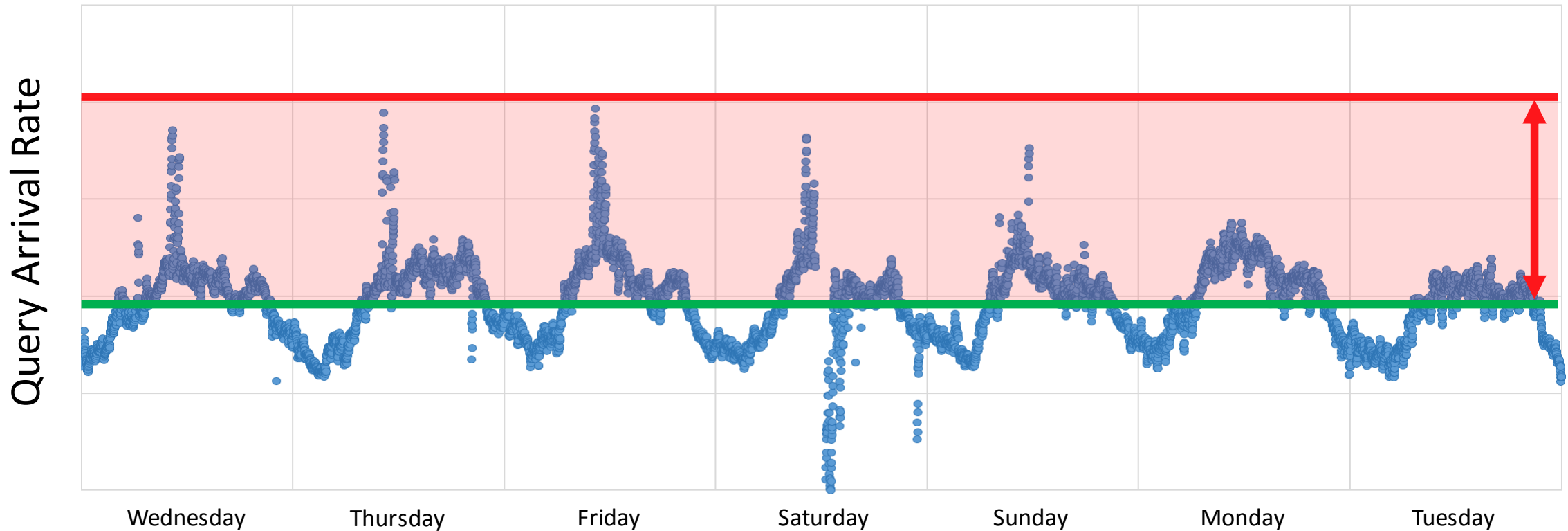
Request-rate variation for a *Microsoft Bing* sub-cluster over 1 week in 2017

Machines are provisioned for peak load

Peak load

>>

Average load



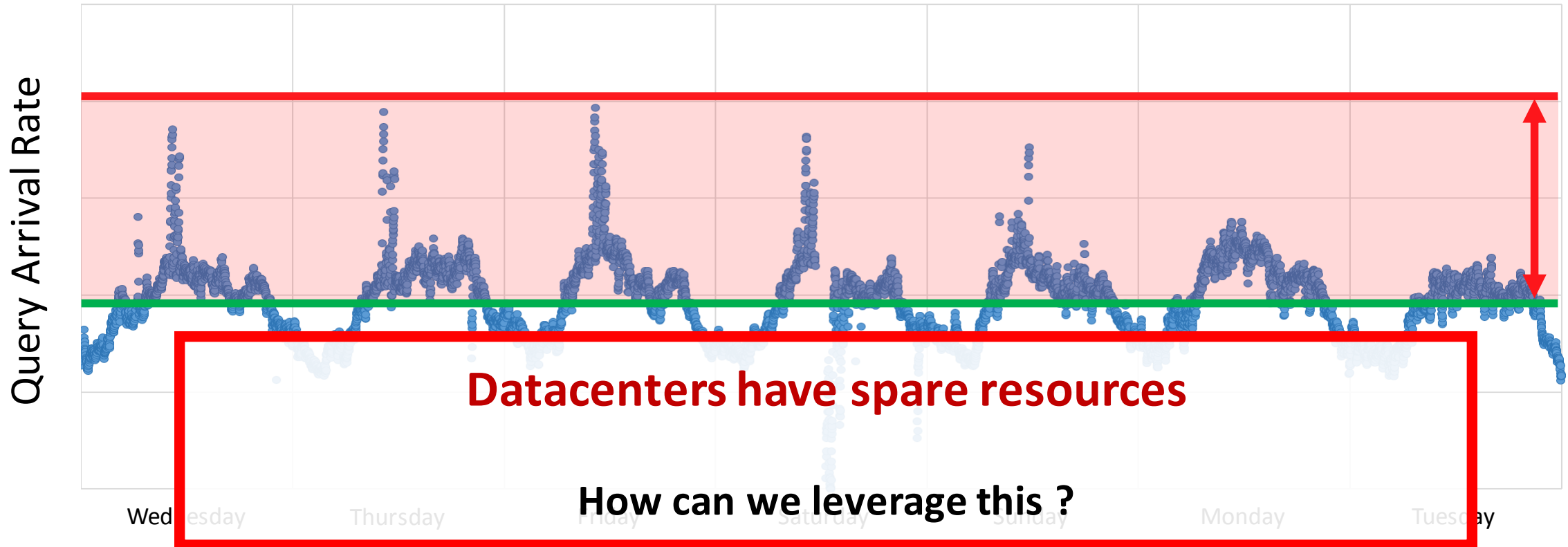
Request-rate variation for a *Microsoft Bing* sub-cluster over 1 week in 2017

Machines are provisioned for peak load

Peak load

>>

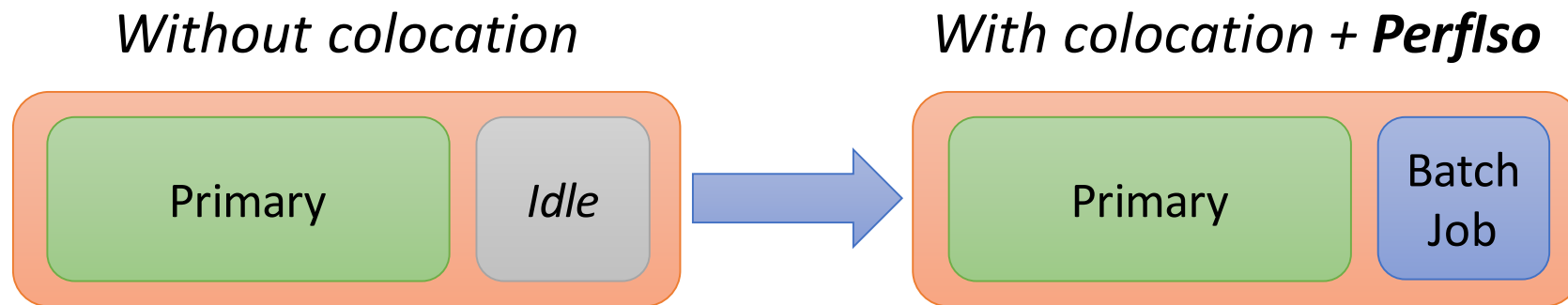
Average load



Request-rate variation for a *Microsoft Bing* sub-cluster over 1 week in 2017

Solution: colocate batch jobs with online services

- Get spare resources to do useful work
- **Primary** tenant – guaranteed performance
 - e.g., Bing IndexServe
- **Secondary** tenant – best-effort performance
 - e.g., Apache Spark



***PerfIso*: performance isolation for online services**

PerfIso: performance isolation for online services

Provides **performance isolation** of Primary

- Maintains P99 of response-times (10s of ms) under colocation

PerfIso: performance isolation for online services

Provides **performance isolation** of Primary

- Maintains P99 of response-times (10s of ms) under colocation

Increases **system efficiency**

- 45% of the CPU is used to do useful batch work

PerfIso: performance isolation for online services

Provides **performance isolation** of Primary

- Maintains P99 of response-times (10s of ms) under colocation

Increases **system efficiency**

- 45% of the CPU is used to do useful batch work

Deployed on over **90,000 servers**

- Many different interactive services and hardware setups

Many papers published on performance isolation

Quasar [ASPLOS '14]

Quasar: Resource-Efficient and QoS-Aware Cluster Management

Christina Delimitrou and Christos Kozyrakis

Stanford University
(codel, kozyraki)@stanford.edu

Abstract

Cloud computing promises flexibility and high performance for users and high cost-efficiency for operators. Nevertheless, most cloud facilities operate at very low utilization, hurting both cost effectiveness and future scalability.

We present Quasar, a cluster management system that increases resource utilization while providing consistently high application performance. Quasar employs three techniques. First, it does not rely on resource reservations, which lead to underutilization as users do not necessarily understand workload dynamics and physical resource requirements of complex codebases. Instead, users express performance constraints for each workload, letting Quasar determine the right amount of resources to meet these constraints at any point. Second, Quasar uses classification techniques to quickly and accurately determine the impact of the amount of resources (scale-out and scale-up), type of resources, and interference on performance for each workload and dataset. Third, it uses the classification results to jointly perform resource allocation and assignment, quickly exploring the large space of options for an efficient way to pack workloads on available resources. Quasar monitors workload performance and adjusts resource allocation and assignment when needed. We evaluate Quasar over a wide range of workload scenarios, including combinations of distributed analytics frameworks and low-latency, stateful services, both on a local cluster and a cluster of dedicated EC2 servers. At steady state, Quasar improves resource utilization by 47% in the 200-server EC2 cluster, while meeting performance constraints for workloads of all types.

Categories and Subject Descriptors: C.5.1 [Computer Systems Implementation]: Super (very large) computers; D.4.1 [Process Management]: Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '14, March 1-5, 2014, Salt Lake City, Utah, USA.
Copyright 2014 ACM 978-1-4503-2954-3/14/03...\$15.00.
ACM 978-1-4503-2954-3/14/03...\$15.00.
http://dx.doi.org/10.1145/2541980.2541941

Keywords: Cloud computing, datacenters, resource efficiency, quality of service, cluster management, resource allocation and assignment.

1. Introduction

An increasing amount of computing is now hosted on public clouds, such as Amazon's EC2 [2], Windows Azure [65] and Google Compute Engine [25], or on private clouds managed by frameworks such as VMware vCloud [61], OpenStack [48], and Mesos [32]. Cloud platforms provide two major advantages for end-users and cloud operators: *flexibility and cost efficiency* [9, 10, 31]. Users can quickly launch jobs that range from short, single-process applications to large, multi-tier services, only paying for the resources used at each point. Cloud operators can achieve economies of scale by building large-scale datacenters (DCs) and by sharing their resources between multiple users and workloads.

Nevertheless, most cloud facilities operate at very low utilization which greatly reduces cost effectiveness [9, 51]. This is the case even for cloud facilities that use cluster management frameworks that enable cluster sharing across workloads. In Figure 1, we present a utilization analysis for a production cluster at Twitter with thousands of servers, managed by Mesos [32] over one month. The cluster mostly hosts user-facing services. The aggregate CPU utilization is consistently below 20%, even though reservations reach up to 80% of total capacity (Fig. 1.a). Even when looking at individual servers, their majority does not exceed 50% utilization on any week (Fig. 1.c). Typical memory use is higher (40-50%) but still differs from the reserved capacity. Figure 1.d shows that very few workloads reserve the right amount of resources (compute resources shown here, similar for memory); most workloads (70%) overestimate reservations by up to 10x, while many (20%) underestimate reservations by up to 5x. Similarly, Reis et al. showed that a 12,000-server Google cluster managed with the more mature Borg system consistently achieves aggregate CPU utilization of 25-35% and aggregate memory utilization of 40% [51]. In contrast, reserved resources exceed 75% and 60% of available capacity for CPU and memory respectively.

Twitter and Google are in the high end of the utilization spectrum. Utilization estimates are even lower for cloud facilities that do not co-locate workloads the way Google and

Heracles [ISCA '15]

Heracles: Improving Resource Efficiency at Scale

David Lo¹, Liqun Cheng², Rama Govindaraj³, Parthasarathy Ranganathan² and Christos Kozyrakis¹
Stanford University¹ Google, Inc.^{2,3}

Abstract

User-facing, latency-sensitive services, such as websearch, underutilize their computing resources during daily periods of low traffic. Reusing those resources for other tasks is rarely done in production services since the contention for shared resources can cause latency spikes that violate the service-level objectives of latency-sensitive tasks. The resulting under-utilization hampers both the affordability and energy-efficiency of large-scale datacenters. With technology scaling slowing down, it becomes important to address this opportunity.

We present Heracles, a feedback-based controller that enables the safe collocation of best-effort tasks alongside a latency-critical service. Heracles dynamically manages multiple hardware and software isolation mechanisms, such as CPU, memory, and network isolation, to ensure that the latency-sensitive job meets latency targets while maximizing the resources given to best-effort tasks. We evaluate Heracles using production latency-critical and batch workloads from Google and demonstrate average server utilizations of 90% without latency violations across all the load and collocation scenarios that we evaluated.

1 Introduction

Public and private cloud frameworks allow us to host an increasing number of workloads in large-scale datacenters with tens of thousands of servers. The business models for cloud services emphasize reduced infrastructure costs. Of the total cost of ownership (TCO) for modern energy-efficient datacenters, servers are the largest fraction (50-70%) [7]. Maximizing server utilization is therefore important for continued scaling.

Until recently, scaling from Moore's law provided higher compute per dollar with every server generation, allowing datacenters to scale without raising the cost. However, with several imminent challenges in technology scaling [21, 23], alternate approaches are needed. Some efforts seek to reduce the server cost through balanced designs or cost-effective components [31, 48, 42]. An orthogonal approach is to improve the return on investment and utility of datacenters by raising server utilization. Low utilization negatively impacts both operational and capital components of cost efficiency. Energy proportionality can reduce operational expenses at low utilization [6, 47].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '15, June 13 - 17, 2015, Portland, OR, USA.
©2015 ACM. ISBN 978-1-4503-3402-0/15/06\$15.00.
DOI: http://dx.doi.org/10.1145/2749459.2749475

But, to amortize the much larger capital expenses, an increased emphasis on the effective use of server resources is warranted.

Several studies have established that the average server utilization in most datacenters is low, ranging between 10% and 50% [14, 74, 66, 7, 19, 13]. A primary reason for the low utilization is the popularity of latency-critical (LC) services such as social media, search engines, software-as-a-service, online maps, webmail, machine translation, online shopping, and advertising. These user-facing services are typically scaled across thousands of servers and access distributed state stored in memory or Flash across these servers. While their load varies significantly due to diurnal patterns and unpredictable spikes in user accesses, it is difficult to consolidate load on a subset of highly utilized servers because the application state does not fit in a small number of servers and moving state is expensive. The cost of such under-utilization can be significant. For instance, Google websearch servers often have an average idleness of 30% over a 24 hour period [47]. For a hypothetical cluster of 10,000 servers, this idleness translates to a wasted capacity of 3,000 servers.

A promising way to improve efficiency is to launch best-effort batch (BE) tasks on the same servers and exploit any resources underutilized by LC workloads [52, 51, 18]. Batch analytics frameworks can generate numerous BE tasks and derive significant value even if these tasks are occasionally deferred or restarted [19, 10, 13, 16]. The main challenge of this approach is interference between colocated workloads on shared resources such as caches, memory, I/O channels, and network links. LC tasks operate with strict service level objectives (SLOs) on tail latency, and even small amounts of interference can cause significant SLO violations [51, 54, 39]. Hence, some of the past work on workload collocation focused only on throughput workloads [58, 15]. More recent systems predict or detect when a LC server suffers significant interference from the colocated tasks, and avoid or terminate the collocation [75, 60, 19, 50, 51, 81]. These systems protect LC workloads, but reduce the opportunities for higher utilization through collocation.

Recently introduced hardware features for cache isolation and fine-grained power control allow us to improve collocation. This work aims to enable aggressive collocation of LC workloads and BE jobs by automatically coordinating multiple hardware and software isolation mechanisms in modern servers. We focus on two hardware mechanisms, shared cache partitioning and fine-grained power/frequency settings, and two software mechanisms, core/thread scheduling and network traffic control. Our goal is to eliminate SLO violations at all levels of load for the LC job while maximizing the throughput for BE tasks.

There are several challenges towards this goal. First, we must carefully share each individual resource: conservative allocation will minimize the throughput for BE tasks, while optimistic allocation will lead to SLO violations for the LC tasks. Second, the performance of both types of tasks depends on multiple resources, which leads to a large allocation space that must be

Elfen [USENIX ATC '16]

Elfen Scheduling: Fine-Grained Principled Borrowing from Latency-Critical Workloads using Simultaneous Multithreading

Xi Yang¹ Stephen M. Blackburn² Kathryn S. McKinley²
¹Australian National University ²Microsoft Research

Abstract

Web services from search to games to stock trading impose strict Service Level Objectives (SLOs) on tail latency. Meeting these objectives is challenging because the computational demand of each request is highly variable and load is bursty. Consequently, many servers run at low utilization (10 to 45%); turn off simultaneous multithreading (SMT); and execute only a single service — wasting hardware, energy, and money. Although co-running batch jobs with latency critical requests to utilize multiple SMT hardware contexts (lanes) is appealing, unmitigated sharing of core resources induces non-linear effects on tail latency and SLO violations.

We introduce *principled borrowing* to control SMT hardware execution in which batch threads borrow core resources. A batch thread executes in a reserved batch SMT lane when no latency-critical thread is executing in the partner request lane. We instrument batch threads to quickly detect execution in the request lane, step out of the way, and promptly return the borrowed resources. We introduce the *macrocopy* system call to stop the batch thread's execution without yielding its lane to the OS scheduler, ensuring that requests have exclusive use of the core's resources. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when request lane is idle improves utilization between 90% and 25% on one core depending on load, without compromising request SLOs. Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

1 Introduction

Latency-critical web services, such as search, trading, games, and social media, must consistently deliver low-latency responses to attract and satisfy users. This requirement translates into Service Level Objectives (SLOs) governing latency. For example, an SLO may include an average latency constraint and a *tail constraint*, such as that 99% of requests must complete within 100 ms [6, 7, 13, 34]. Many such services, such as Google Search and Twitter [6, 8, 18], systematically underutilize the available hardware to meet SLOs. Furthermore,

servers often execute only one service to ensure that latency-critical requests are free from interference. The result is that server utilizations are as low as 10 to 45%. Since these services are widely deployed in large numbers of datacenters, their poor utilization incurs enormous commensurate capital and operating costs. Even small improvements substantially improve profitability.

Meeting SLOs in these highly engineered systems is challenging because: (1) requests often have variable computational demands and (2) load is unpredictable and bursty. Since computation demands of requests may differ by factors of ten or more and load bursts induce queuing delay, overloading a server results in highly non-linear increases in tail latency. The conservative solution providers often take is to significantly over-provision.

Interference arises in chip multiprocessors (CMPs) and in simultaneous multithreading (SMT) cores when contending for shared resources. A spate of recent research explores how to predict and model interference between different workloads executing on distinct CMP cores [8, 23, 25, 28], but these approaches target and exploit large scale diurnal patterns of utilization, e.g., co-locating batch workloads at night when load is low. Lo et al. explicitly rule out SMT because of the highly unpredictable and non-linear impact on tail latency (which we confirm) and the inadequacy of high-latency OS scheduling [23]. Zhang et al. do not attempt to reduce SMT-induced overheads, but rather they accommodate them using a model of interference for co-running workloads [35]. Their approach requires ahead-of-time profiling of all co-located workloads and over-provisioning. Prior work lacks dynamic mechanisms to monitor and control batch workloads on SMT with low latency.

This research exploits SMT resources to increase utilization without compromising SLOs. We introduce *principled borrowing*, which dynamically identifies idle cycles and borrows these resources. We implement borrowing in the ELFEN¹ scheduler, which co-runs batch threads and latency-critical requests, and meets request SLOs. Our work is complementary to managing shared cache and memory resources. We first show that latency-critical workloads impose many idle periods and they are short. This result confirms that scheduling at OS granu-

¹In the Grimm fairy tale, Die Wichtelmännchen, elves borrow a kobler's tools while he sleeps, making him beautiful shoes.

Many papers published on performance isolation

Quasar [ASPLOS '14]

Quasar: Resource-Efficient and QoS-Aware Cluster Management

Christina Delimitrou and Christos Kozyrakis

Stanford University
(codel, kozyrakis)@stanford.edu

Abstract

Cloud computing promises flexibility and high performance for users and high cost-efficiency for operators. Nevertheless, most cloud facilities operate at very low utilization, hurting both cost effectiveness and energy efficiency.

We present Quasar, a cloud management system that increases resource utilization while providing consistently high application performance. Quasar employs three techniques. First, it does not rely on resource reservations, which lead to underutilization as users do not necessarily understand workload dynamics and physical resource requirements of complex codebases. Instead, users express performance constraints for each workload, letting Quasar determine the right amount of resources to meet these constraints at any point. Second, Quasar uses classification techniques to quickly and accurately determine the impact of the amount of resources (scale-out and scale-up), type of resources, and interference on performance for each workload and dataset. Third, it uses the classification results to jointly perform resource allocation and assignment, quickly exploring the large space of options for an efficient way to pack workloads on available resources. Quasar monitors workload performance and adjusts resource allocation and assignment when needed. We evaluate Quasar over a wide range of workload scenarios, including combinations of distributed analytics frameworks and low-latency, stateful services, both on a local cluster and a cluster of dedicated EC2 servers. At steady state, Quasar improves resource utilization by 47% in the 200-server EC2 cluster, while meeting performance constraints for workloads of all types.

Categories and Subject Descriptors: C.5.1 [Computer System Implementation]: Super (very large) computers; D.4.1 [Process Management]: Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or fee. Request permissions from permissions@acm.org.
ASPLOS '14, March 1-5, 2014, Salt Lake City, Utah, USA.
Copyright 2014 ACM 978-1-4503-2914-0/14/03...\$15.00.
ACM 978-1-4503-2914-0/14/03...\$15.00.
http://dx.doi.org/10.1145/2541940.2541941

Heracles [ISCA '15]

Heracles: Improving Resource Efficiency at Scale

David Lo¹, Liqun Cheng², Rama Govindaraj², Parthasarathy Ranganathan² and Christos Kozyrakis¹
Stanford University¹, Google, Inc.²

Abstract

User-facing, latency-sensitive services, such as websearch, require high resource efficiency. But, to amortize the much larger capital expenses, an increased emphasis on the effective use of server resources is warranted. Several studies have established that the average server utilization in production services since the construction for shared resources is only 10-15%.

In production services since the construction for shared resources is only 10-15%. This paper introduces Heracles, a system that improves resource efficiency at scale by building large-scale datacenters (DCs) and by sharing their resources between multiple users and workloads. Heracles is the first system to provide latency-sensitive jobs with latency targets while maximizing the resources given to best-effort tasks. We evaluate Heracles using production latency-critical and batch workloads from Google and demonstrate average server utilizations of 90% without latency violations across all the load and collocation scenarios that we evaluated.

Heracles improves resource efficiency at scale by building large-scale datacenters (DCs) and by sharing their resources between multiple users and workloads. Heracles is the first system to provide latency-sensitive jobs with latency targets while maximizing the resources given to best-effort tasks. We evaluate Heracles using production latency-critical and batch workloads from Google and demonstrate average server utilizations of 90% without latency violations across all the load and collocation scenarios that we evaluated.

1 Introduction

Public and private cloud frameworks allow us to host an increasing number of workloads in large-scale datacenters with tens of thousands of servers. The business models for cloud services emphasize reduced infrastructure costs. Of the total cost of ownership (TCO) for modern energy-efficient datacenters, servers are the largest fraction (50-70%) [7]. Maximizing server utilization is therefore important for continued scaling.

Until recently, scaling from Moore's law provided higher compute per dollar with every server generation, allowing datacenters to scale without raising the cost. However, with several imminent challenges in technology scaling [21, 23], alternate approaches are needed. Some efforts seek to reduce the server cost through balanced designs or cost-effective components [31, 48, 42]. An orthogonal approach is to improve the return on investment and utility of datacenters by raising server utilization. Low utilization negatively impacts both operational and capital components of cost efficiency. Energy proportionality can reduce operational expenses at low utilization [6, 47].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or fee. Request permissions from permissions@acm.org.
ISCA '15, June 13-17, 2015, Portland, OR, USA.
©2015 ACM. ISBN 978-1-4503-3422-2/15/06\$15.00.
DOI: http://dx.doi.org/10.1145/2749459.2749475

Elfen [USENIX ATC '16]

Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads using Simultaneous Multithreading

Xi Yang¹, Stephen M. Blackburn², Kathryn S. McKinley²
¹Australian National University, ²Microsoft Research

With server farms growing in size, the cost of server farms is increasing. Latency-critical requests, such as search, trading, games, and social media, must consistently deliver low-latency responses to attract and satisfy users. This requirement translates into Service Level Objectives (SLOs) governing latency. For example, an SLO may include an average latency constraint and a tail constraint, such as that 99% of requests must complete within 100ms [6, 7, 13, 34]. Many such services, such as Google Search and Twitter [6, 8, 18], systematically underutilize the available hardware to meet SLOs. Furthermore, one service to ensure that latency-critical requests are as low as 10 to 45%. Since these services are widely deployed in large numbers, their underutilization incurs enormous operating costs. Even small improvements substantially improve profitability. Modern datacenters are highly engineered systems in which batch threads borrow core resources. A batch thread executes in a reserved batch SMT lane when no latency-critical thread is executing in the partner request lane. We instrument batch threads to quickly detect execution in the request lane, step out of the way, and promptly return the borrowed resources. We introduce the *principle* system call to stop the batch thread's execution without yielding its lane to the OS scheduler, ensuring that requests have exclusive use of the core's resources. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when request lane is idle improves utilization between 90% and 25% on one core depending on load, without compromising request SLOs. Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

We introduce *principled borrowing* to control SMT hardware execution in which batch threads borrow core resources. A batch thread executes in a reserved batch SMT lane when no latency-critical thread is executing in the partner request lane. We instrument batch threads to quickly detect execution in the request lane, step out of the way, and promptly return the borrowed resources. We introduce the *principle* system call to stop the batch thread's execution without yielding its lane to the OS scheduler, ensuring that requests have exclusive use of the core's resources. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when request lane is idle improves utilization between 90% and 25% on one core depending on load, without compromising request SLOs. Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

1 Introduction

Latency-critical web services, such as search, trading, games, and social media, must consistently deliver low-latency responses to attract and satisfy users. This requirement translates into Service Level Objectives (SLOs) governing latency. For example, an SLO may include an average latency constraint and a tail constraint, such as that 99% of requests must complete within 100ms [6, 7, 13, 34]. Many such services, such as Google Search and Twitter [6, 8, 18], systematically underutilize the available hardware to meet SLOs. Furthermore,

one service to ensure that latency-critical requests are as low as 10 to 45%. Since these services are widely deployed in large numbers, their underutilization incurs enormous operating costs. Even small improvements substantially improve profitability. Modern datacenters are highly engineered systems in which batch threads borrow core resources. A batch thread executes in a reserved batch SMT lane when no latency-critical thread is executing in the partner request lane. We instrument batch threads to quickly detect execution in the request lane, step out of the way, and promptly return the borrowed resources. We introduce the *principle* system call to stop the batch thread's execution without yielding its lane to the OS scheduler, ensuring that requests have exclusive use of the core's resources. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when request lane is idle improves utilization between 90% and 25% on one core depending on load, without compromising request SLOs. Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

This research exploits SMT resources to increase utilization without compromising SLOs. We introduce *principled borrowing*, which dynamically identifies idle cycles and borrows these resources. We implement borrowing in the ELFEN¹ scheduler, which co-runs batch threads and latency-critical requests, and meets request SLOs. Our work is complementary to managing shared cache and memory resources. We first show that latency-critical workloads impose many idle periods and they are short. This result confirms that scheduling at OS granularity is not sufficient to meet SLOs.

This research exploits SMT resources to increase utilization without compromising SLOs. We introduce *principled borrowing*, which dynamically identifies idle cycles and borrows these resources. We implement borrowing in the ELFEN¹ scheduler, which co-runs batch threads and latency-critical requests, and meets request SLOs. Our work is complementary to managing shared cache and memory resources. We first show that latency-critical workloads impose many idle periods and they are short. This result confirms that scheduling at OS granularity is not sufficient to meet SLOs.

¹In the Grimm fairy tale, *Die Wichtelmännchen*, elves borrow a kobler's tools while he sleeps, making him beautiful shoes.

PerfIso: Requirements

PerfIso: Requirements

1. **“Black-box”**: Fewest assumptions about tenants (wider applicability)

PerfIso: Requirements

1. **“Black-box”**: Fewest assumptions about tenants (wider applicability)
2. **“Standalone”**: Primary acts like it runs alone (negligible interference)

PerfIso: Requirements

1. **“Black-box”**: Fewest assumptions about tenants (wider applicability)
2. **“Standalone”**: Primary acts like it runs alone (negligible interference)
3. **“Integrability”**: Minimize software-stack changes (easy deployment)

Why is Performance Isolation hard?

Interactive services – highly sensitive to interference!

Leaf-servers keep 99th percentile low

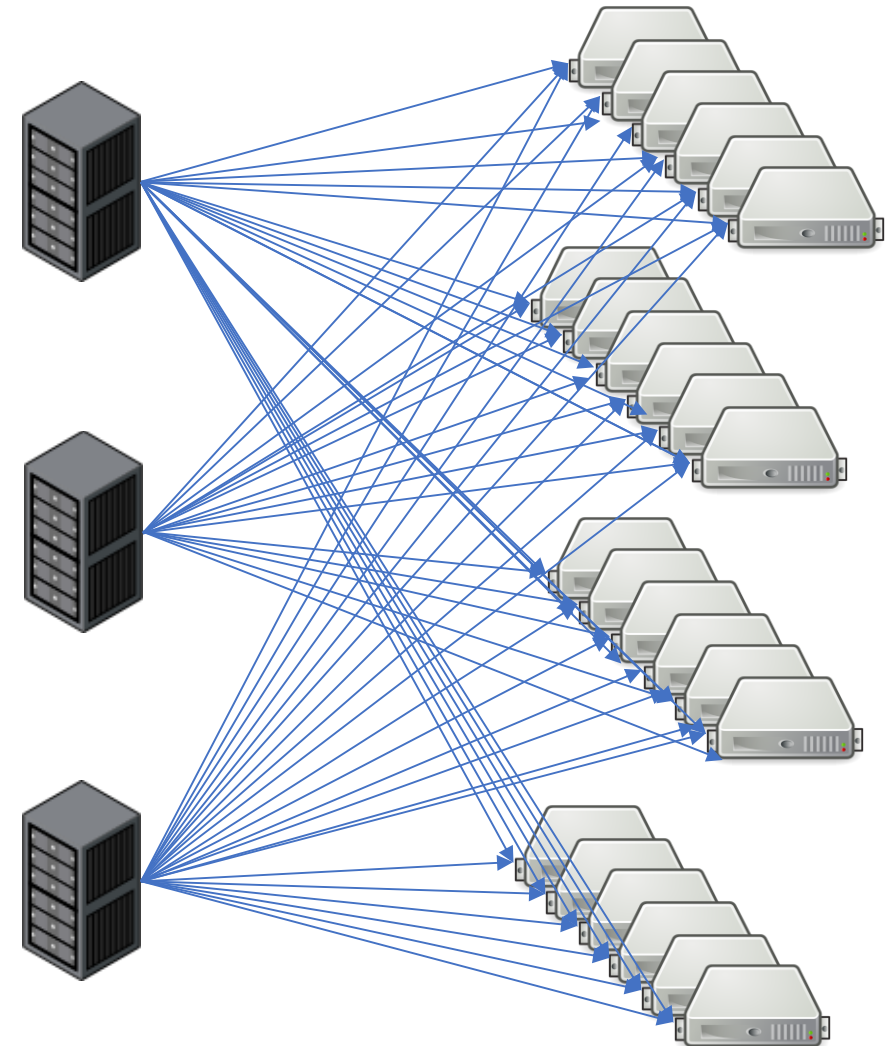
- Over 10 years of optimization work!
 - e.g., compression, adaptive parallelism, etc.

How often does the 99th percentile occur?

- For 10,000 queries / s \rightarrow 100 times / s

What happens in a 100-node fanout?

- Every query runs at the 99th percentile!



The Primary demands many resources quickly

- **Bing IndexServe**: multi-threaded web-index server
 - Up to 15 threads wake up in $5\mu s$ ¹

¹Constant query rate 4,000 Q/s, 500k queries experiment

The Primary demands many resources quickly

- **Bing IndexServe**: multi-threaded web-index server
 - Up to 15 threads wake up in $5\mu s$ ¹
- **Burstiness due to query-processing optimizations!**
 - some queries will spawn many workers

¹Constant query rate 4,000 Q/s, 500k queries experiment

The Primary demands many resources quickly

- **Bing IndexServe**: multi-threaded web-index server
 - Up to 15 threads wake up in $5\mu s$ ¹
- **Burstiness due to query-processing optimizations!**
 - some queries will spawn many workers
- **Workload arrives in bursts – exacerbates problem**

¹Constant query rate 4,000 Q/s, 500k queries experiment

The Primary must behave as if it were *standalone*

The Primary must behave as if it were *standalone*

- Primary's resource demands must be fulfilled instantly.

The Primary must behave as if it were *standalone*

- Primary's resource demands must be fulfilled instantly.
- Any delays → performance penalties incurred

The Primary must behave as if it were *standalone*

- Primary's resource demands must be fulfilled instantly.
- Any delays → performance penalties incurred
- Any resource can become a performance bottleneck.

The Primary must behave as if it were *standalone*

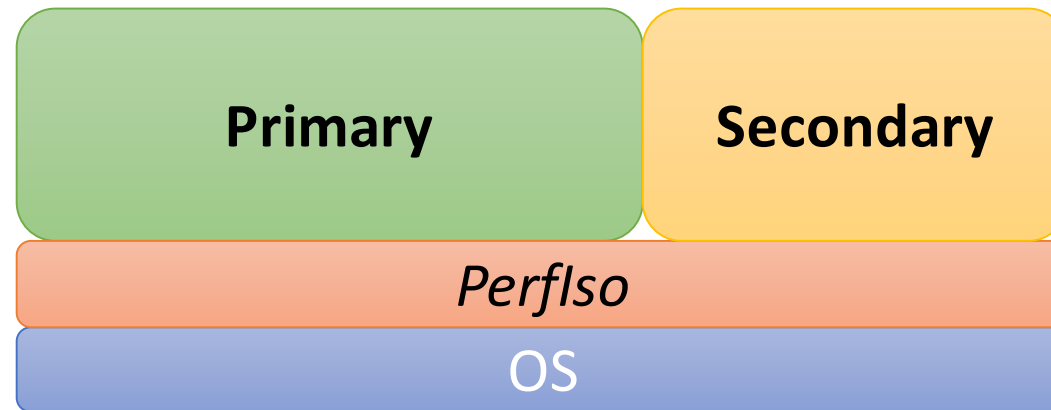
- Primary's resource demands must be fulfilled instantly.
- Any delays → performance penalties incurred
- Any resource can become a performance bottleneck.

If a query is delayed, it is already too late!

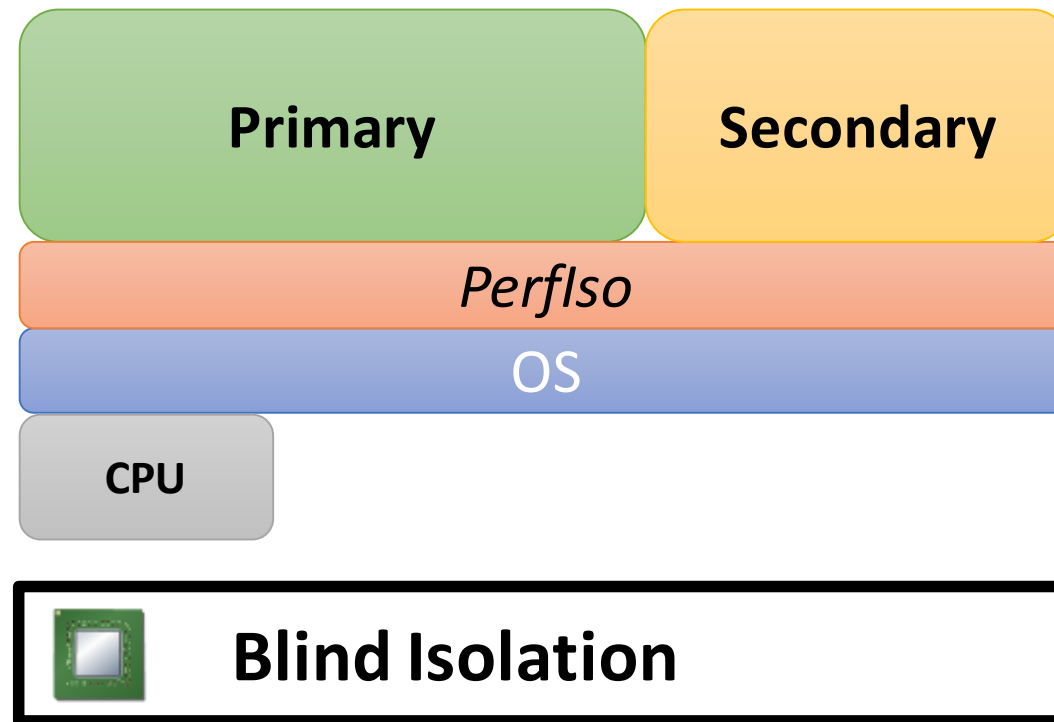
Perflso

PerfIso: Implemented as a user-mode service

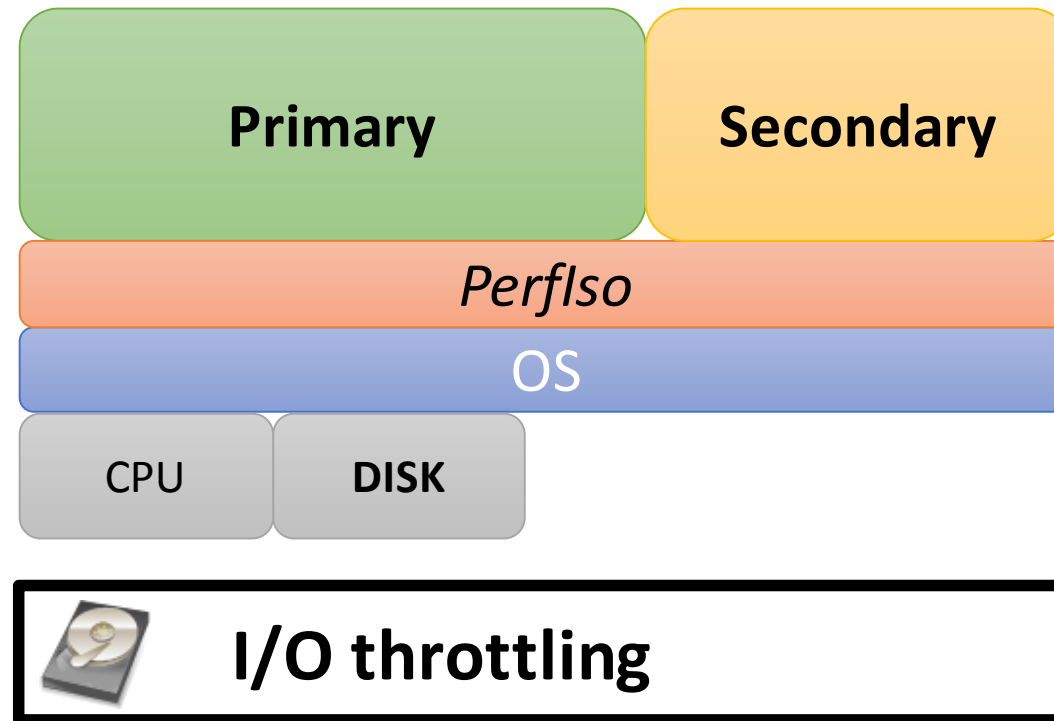
- Only keeps track of **Secondary**'s PID



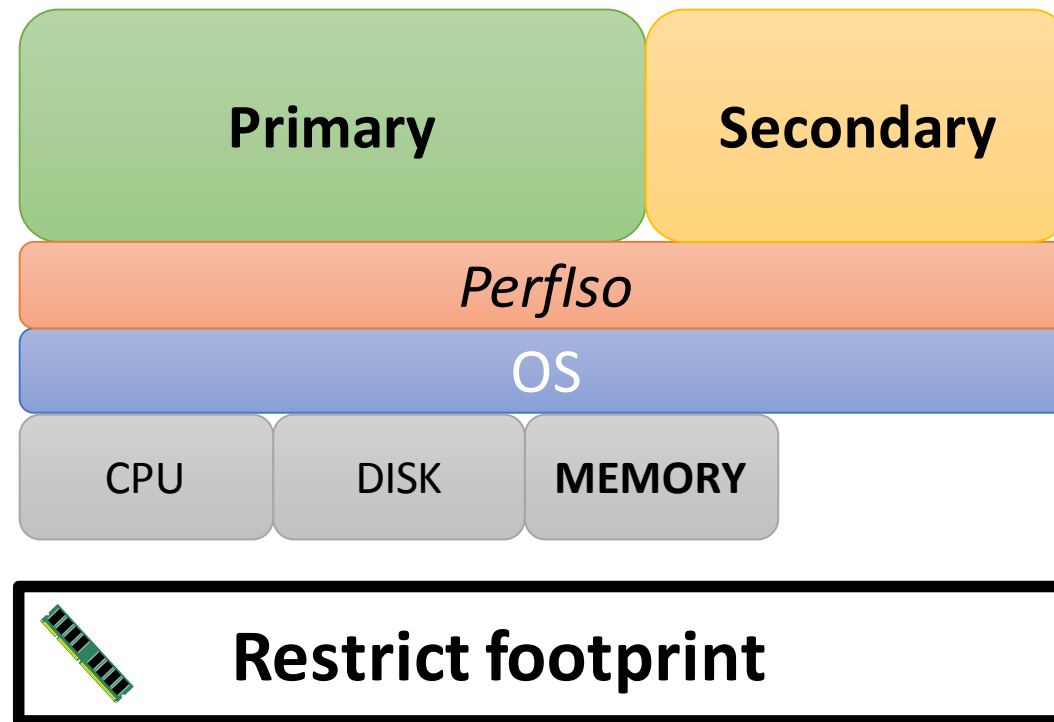
PerfIso: Managed resources



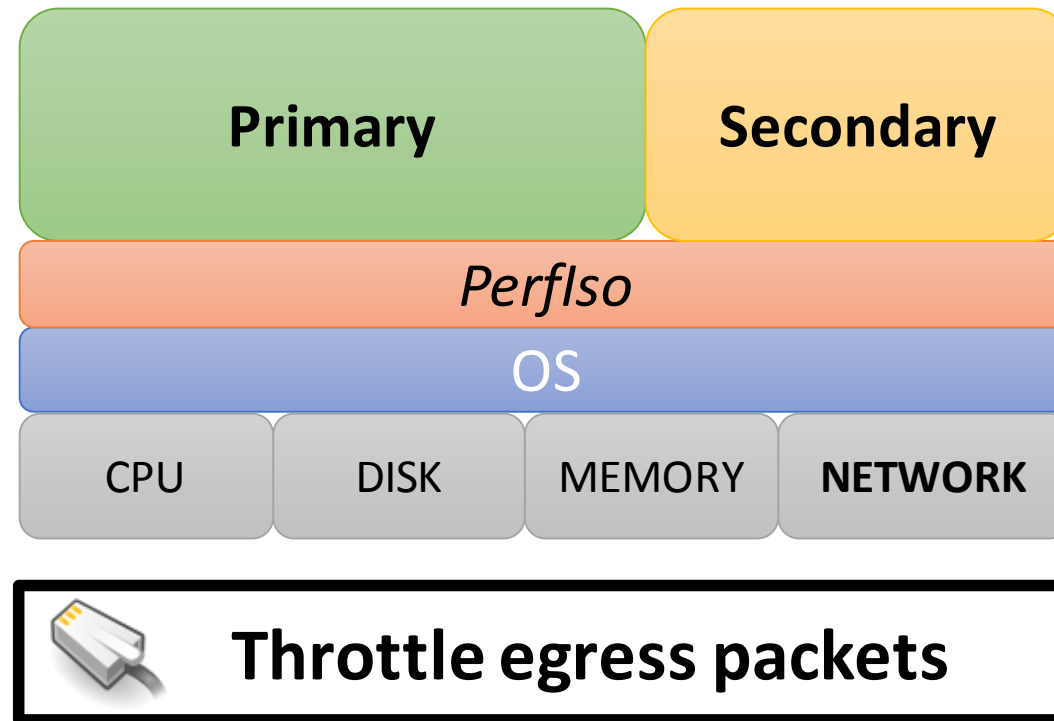
PerfIso: Managed resources



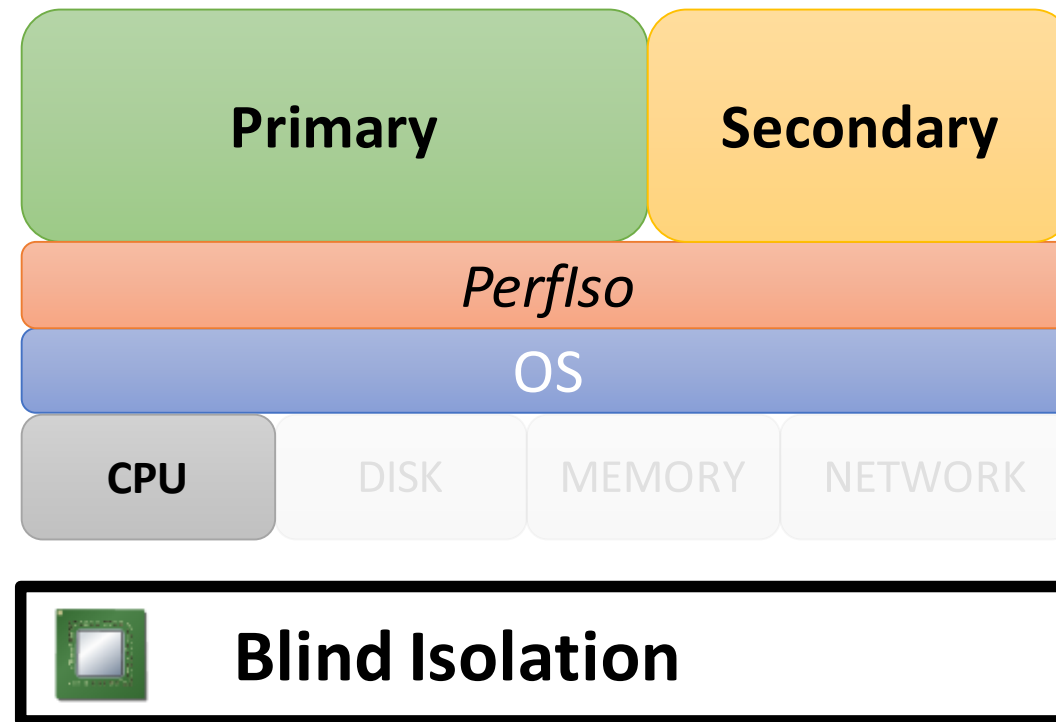
PerfIso: Managed resources



PerfIso: Managed resources

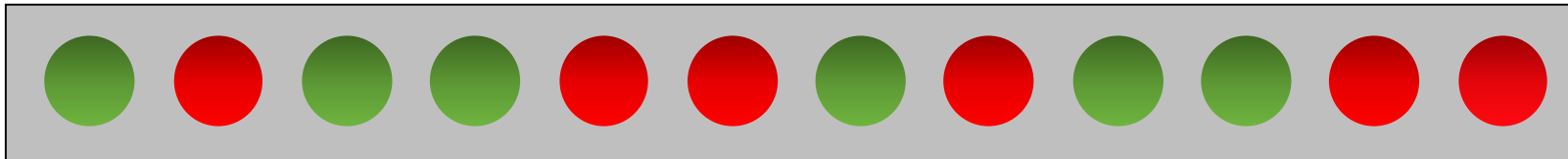
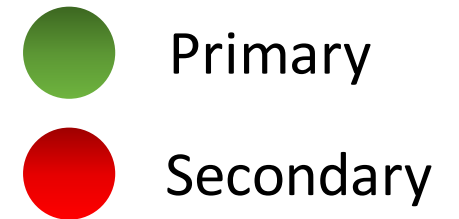


PerfIso: CPU is the most important resource



CPU sharing without *PerfIso*

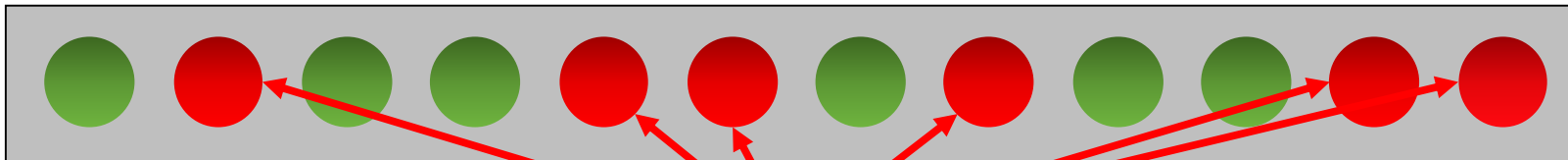
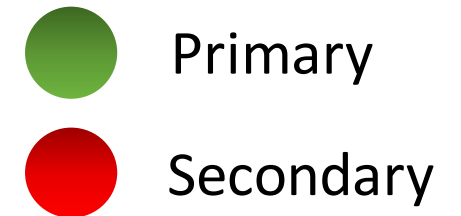
- **Primary** and **Secondary** compete for cores.
- **Secondary** is aggressive: no idle cores exist.



Machine with **12 cores**

CPU Blind Isolation: Keep a “buffer” of idle cores

- *Perflso* only knows the **Secondary**.
- Restrict **Secondary** by changing *core affinities*.

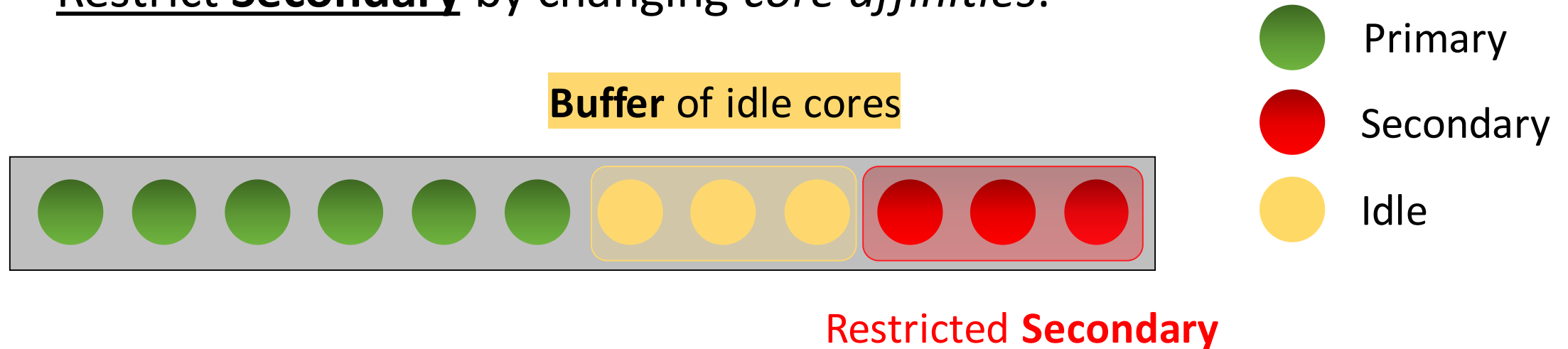


Restrict Secondary to create a
buffer of idle cores.

Machine with **12 cores**

CPU Blind Isolation: Keep a “buffer” of idle cores

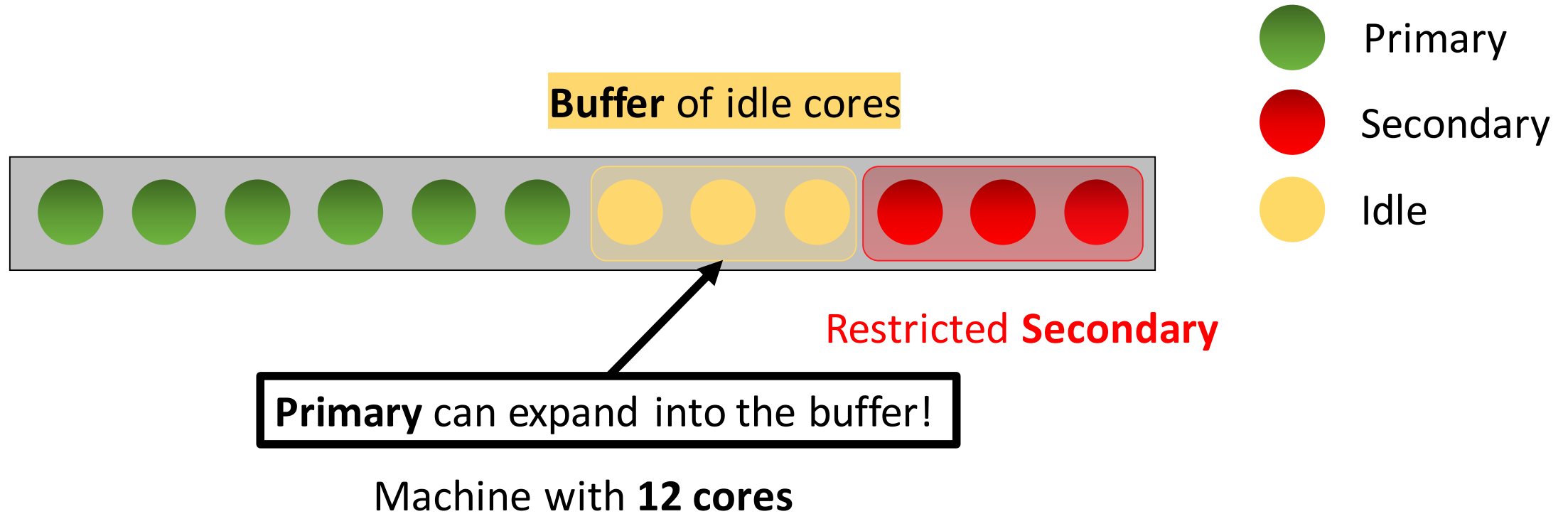
- *Perflso* only knows the **Secondary**.
- Restrict **Secondary** by changing *core affinities*.



Machine with **12 cores**

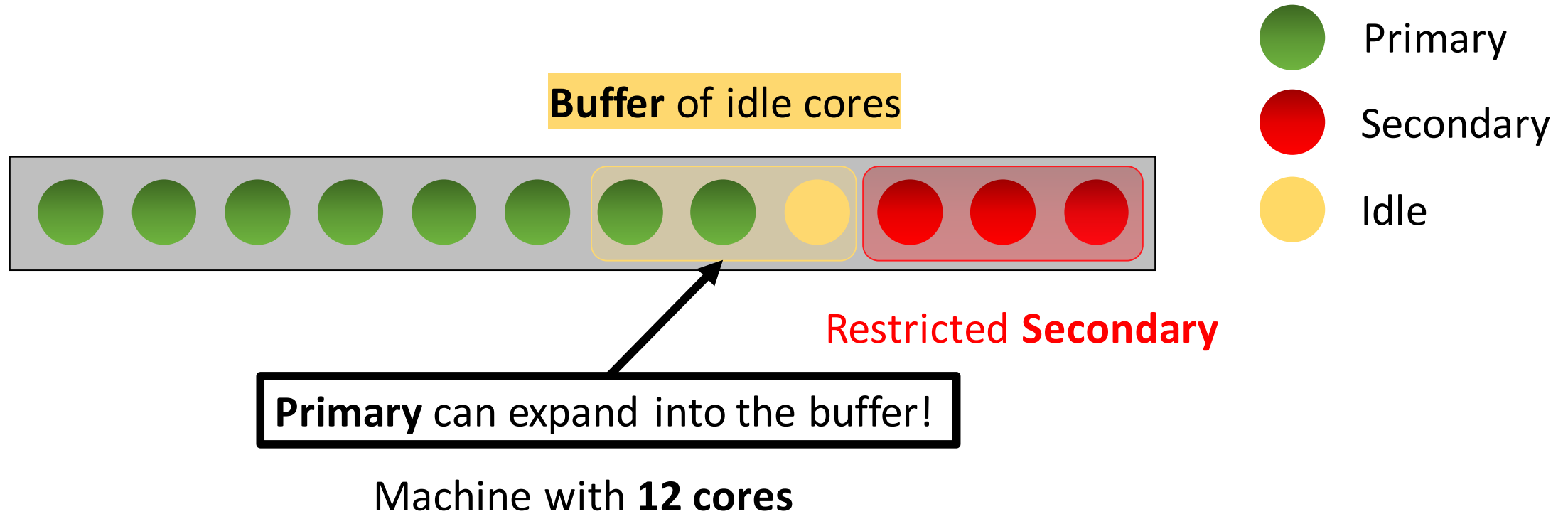
CPU Blind Isolation: Keep a “buffer” of idle cores

- **Primary** is unrestricted. **Secondary** is restricted.



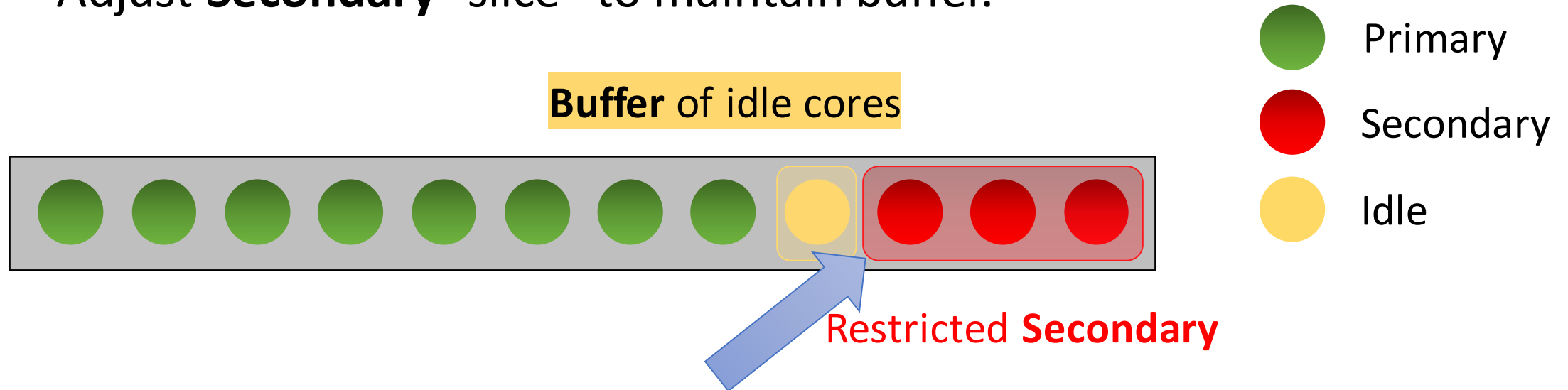
CPU Blind Isolation: Keep a “buffer” of idle cores

- **Primary** is unrestricted. **Secondary** is restricted.



CPU Blind Isolation: React to bursts from Primary

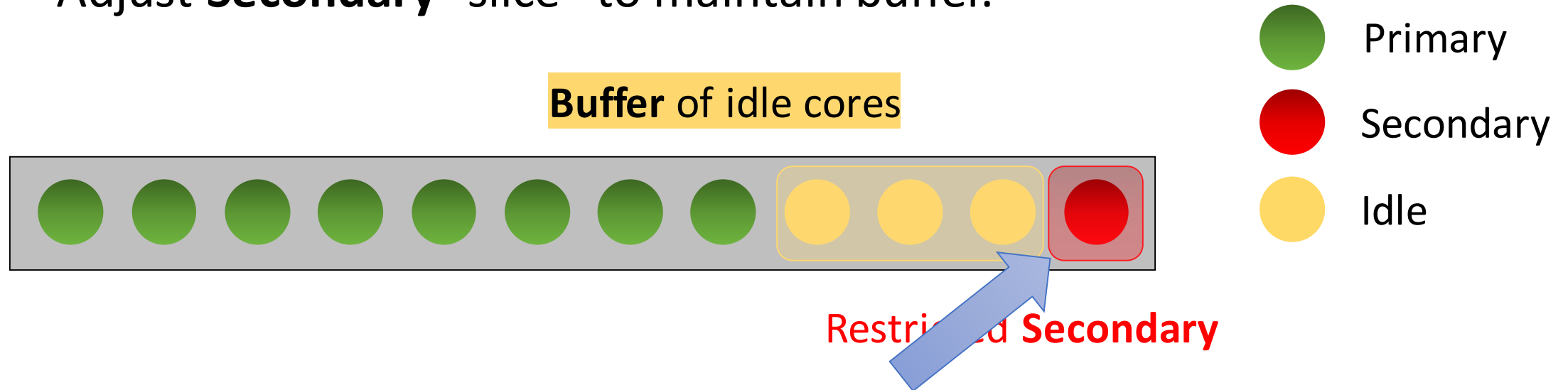
- Continuously read idle core status.
- Adjust **Secondary** "slice" to maintain buffer.



Machine with **12 cores**

CPU Blind Isolation: React to bursts from Primary

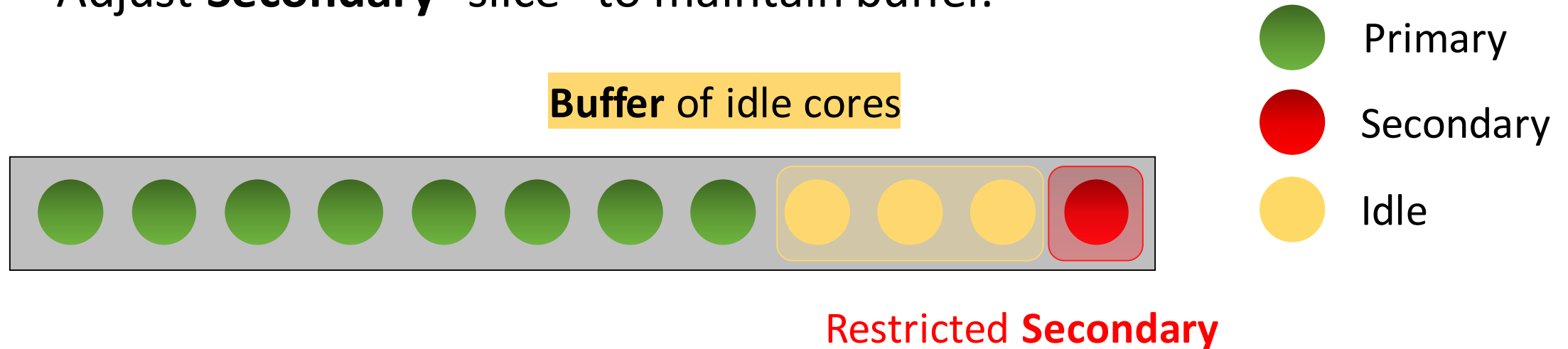
- Continuously read idle core status.
- Adjust **Secondary** "slice" to maintain buffer.



Machine with **12 cores**

CPU Blind Isolation: React to bursts from Primary

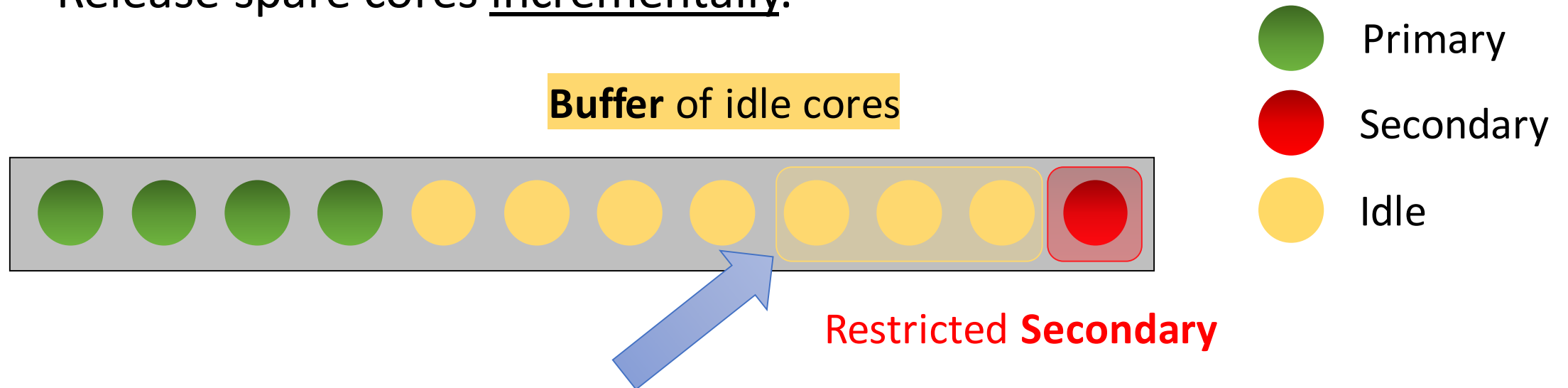
- Continuously read idle core status.
- Adjust **Secondary** "slice" to maintain buffer.



Machine with **12 cores**

CPU Blind Isolation: Secondary gets spare cores

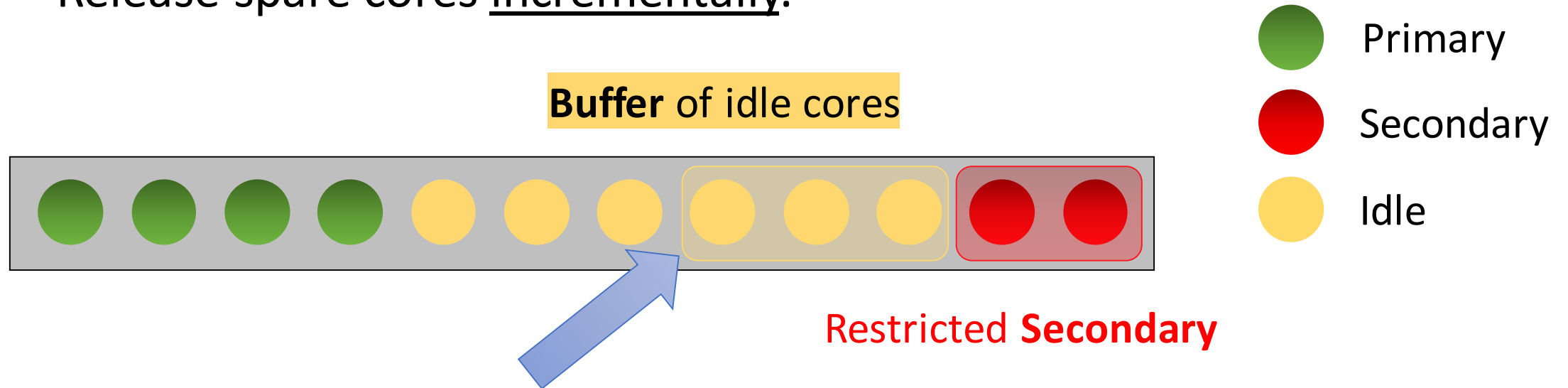
- Allow **Secondary** to use spare idle cores.
- Release spare cores incrementally.



Machine with **12 cores**

CPU Blind Isolation: Secondary gets spare cores

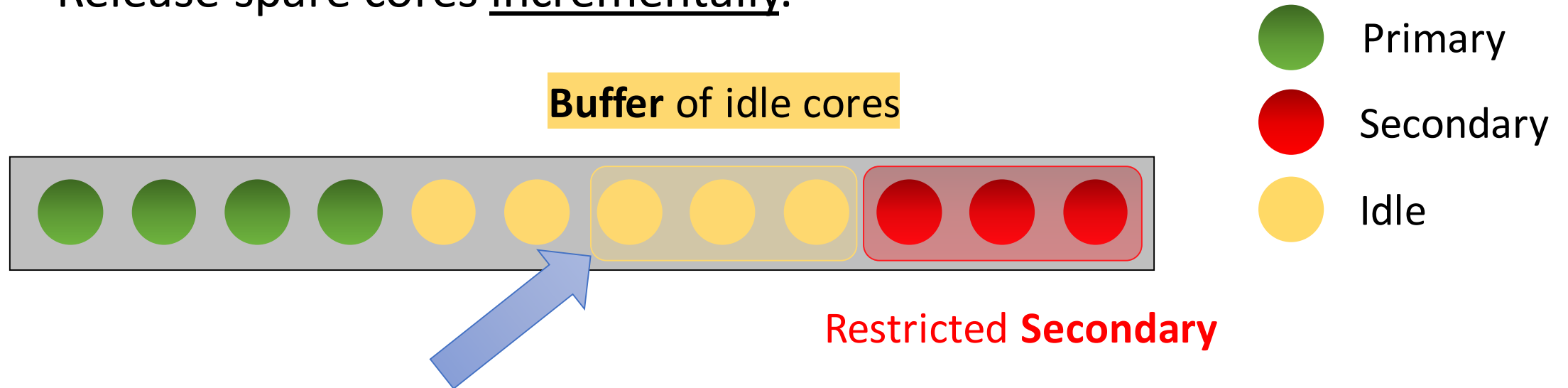
- Allow **Secondary** to use spare idle cores.
- Release spare cores incrementally.



Machine with **12 cores**

CPU Blind Isolation: Secondary gets spare cores

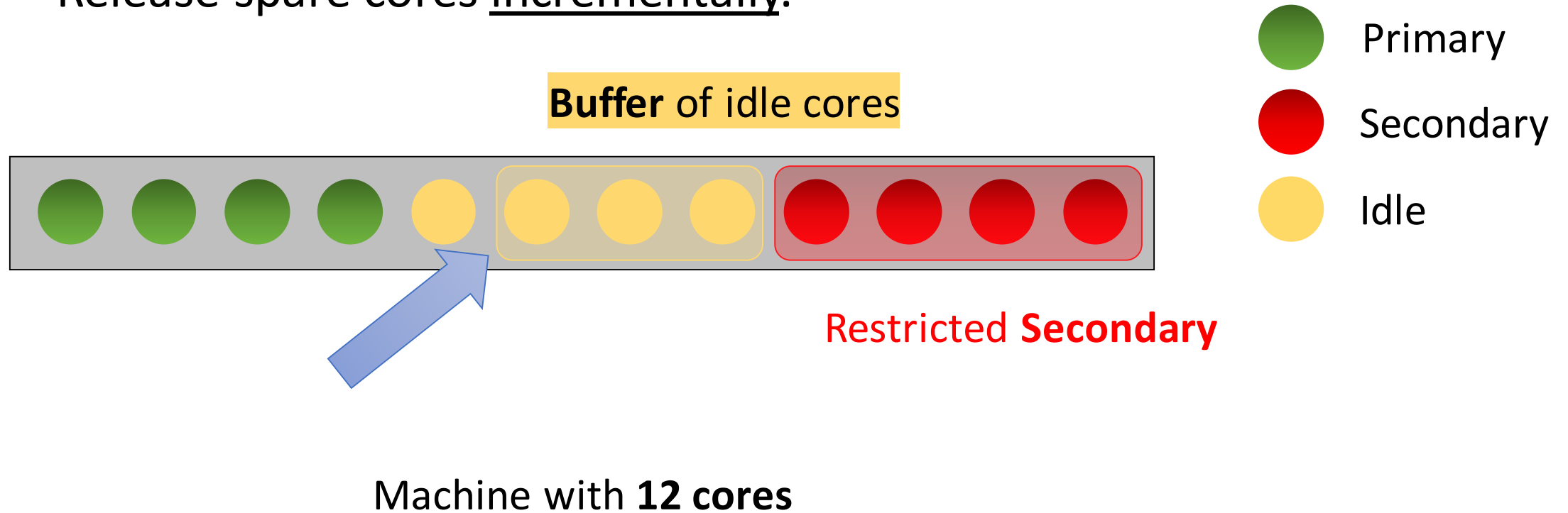
- Allow **Secondary** to use spare idle cores.
- Release spare cores incrementally.



Machine with **12 cores**

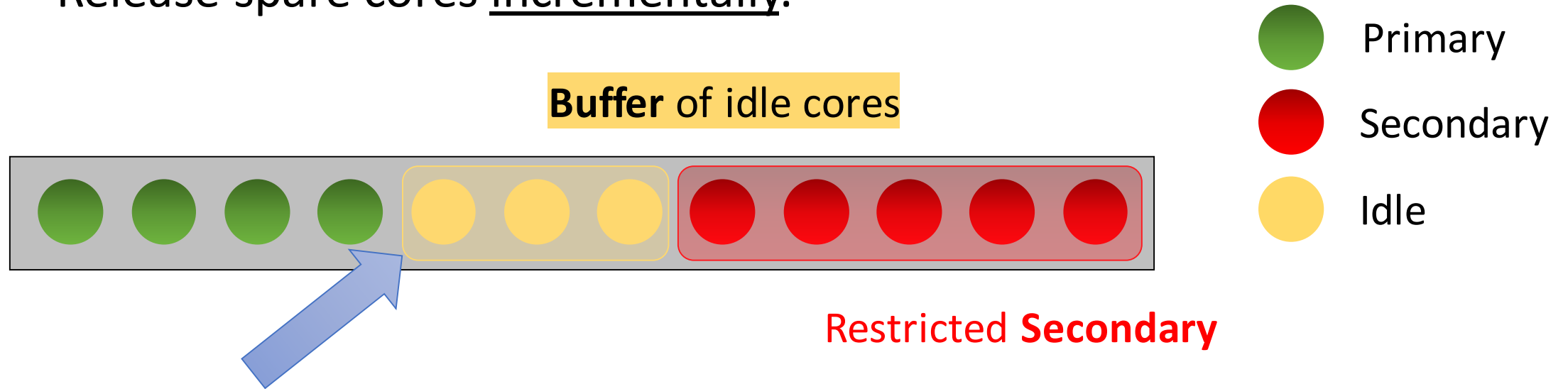
CPU Blind Isolation: Secondary gets spare cores

- Allow **Secondary** to use spare idle cores.
- Release spare cores incrementally.



CPU Blind Isolation: Secondary gets spare cores

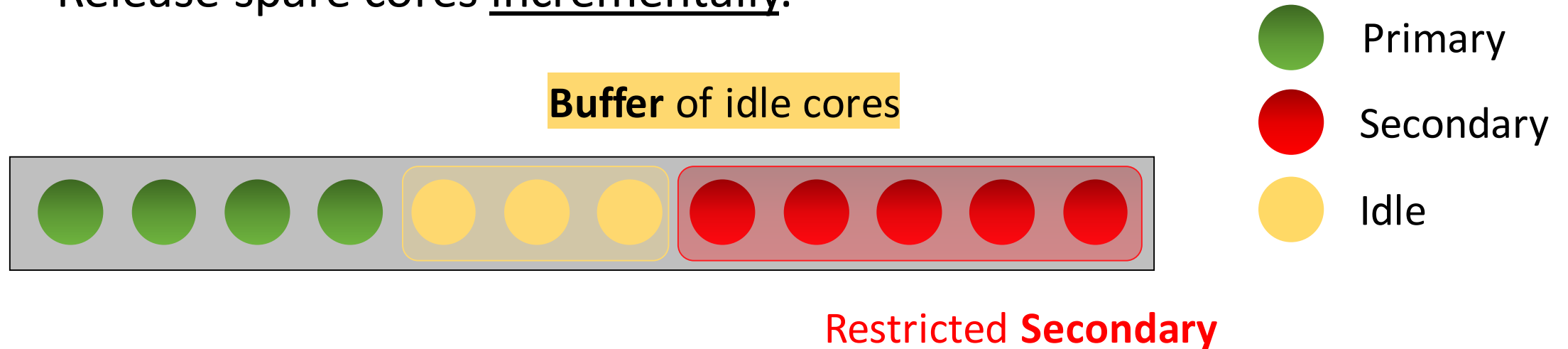
- Allow **Secondary** to use spare idle cores.
- Release spare cores incrementally.



Machine with **12 cores**

CPU Blind Isolation: Secondary gets spare cores

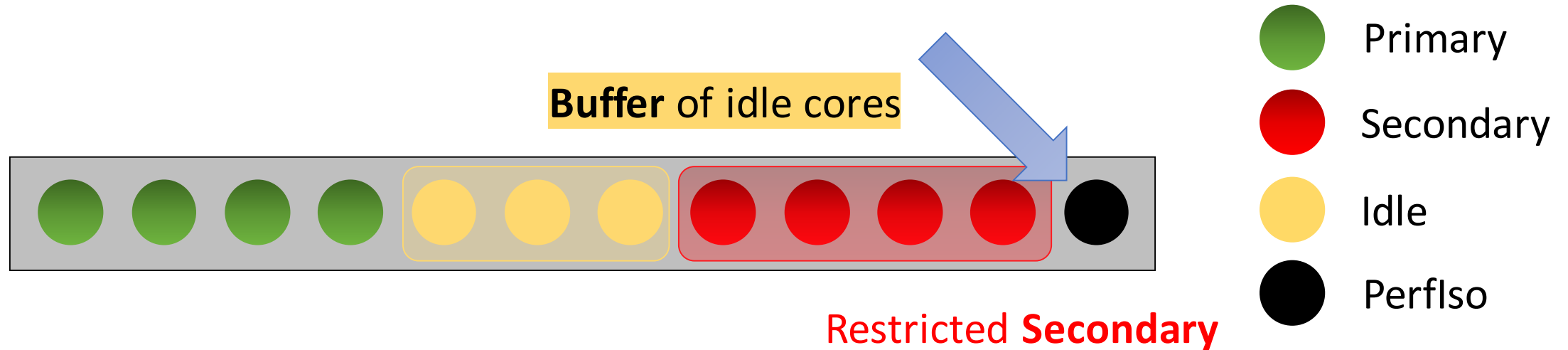
- Allow **Secondary** to use spare idle cores.
- Release spare cores incrementally.



Machine with **12 cores**

CPU Blind Isolation: We dedicate 1 core to *Perflso*

- *Perflso* does continuous polling → we affinitize it to 1 core.



Machine with **12 cores**

Evaluation

Experiment testbed

Hardware

- Intel Xeon E5 – 24 cores (48 w/ HT)
- 128GB RAM

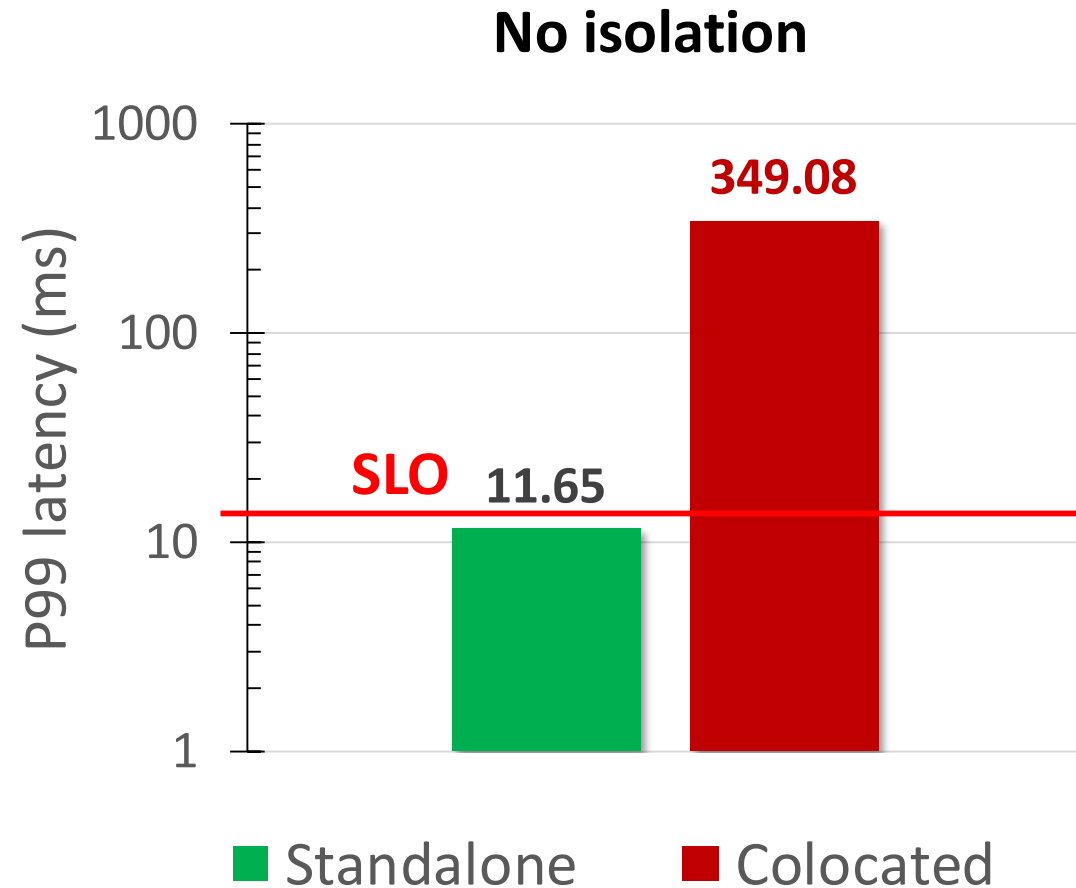
Primary: *Bing IndexServe*

- 569 GB index-slice
- Open-loop client
- 500,000 queries @ 2,000 Q / s

Secondary: CPU micro-benchmark

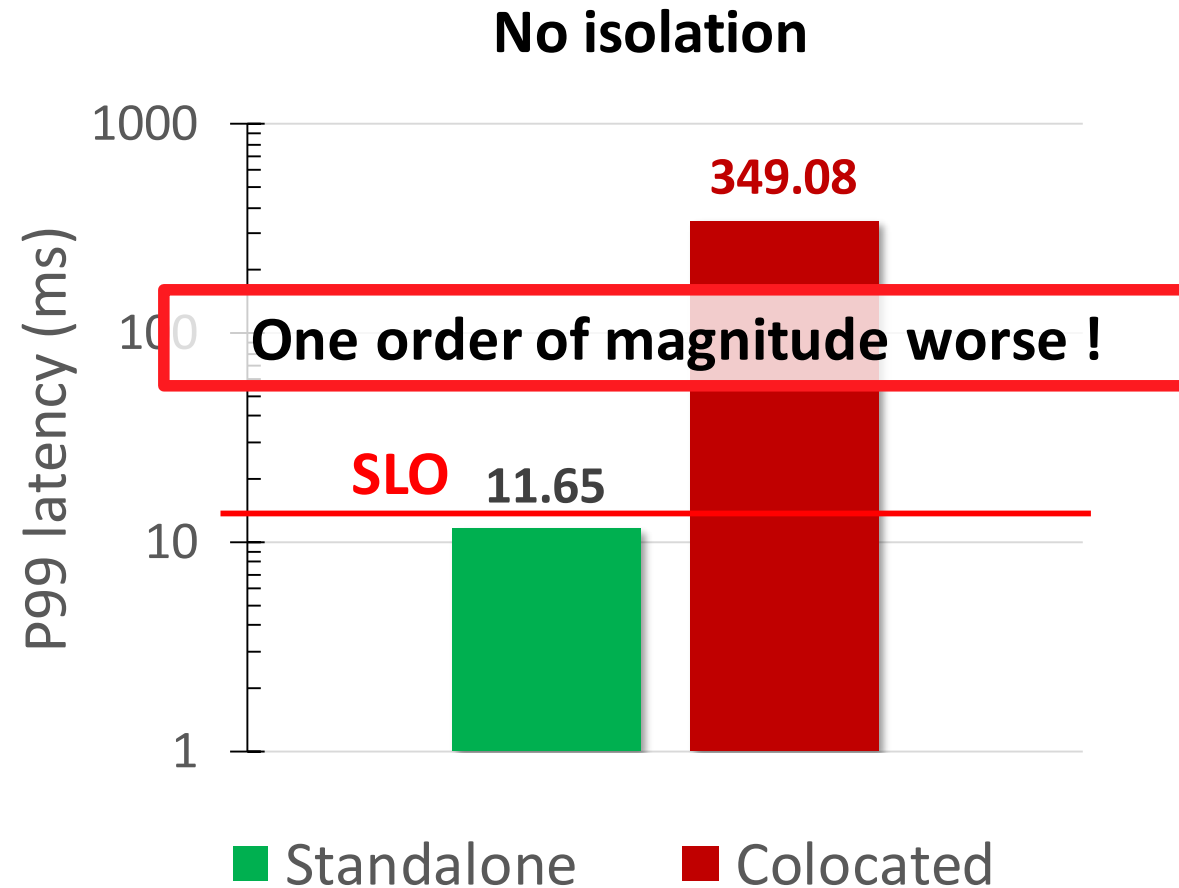
Single server: *PerfIso* protects tail-latency

Secondary: CPU-intensive micro-benchmark



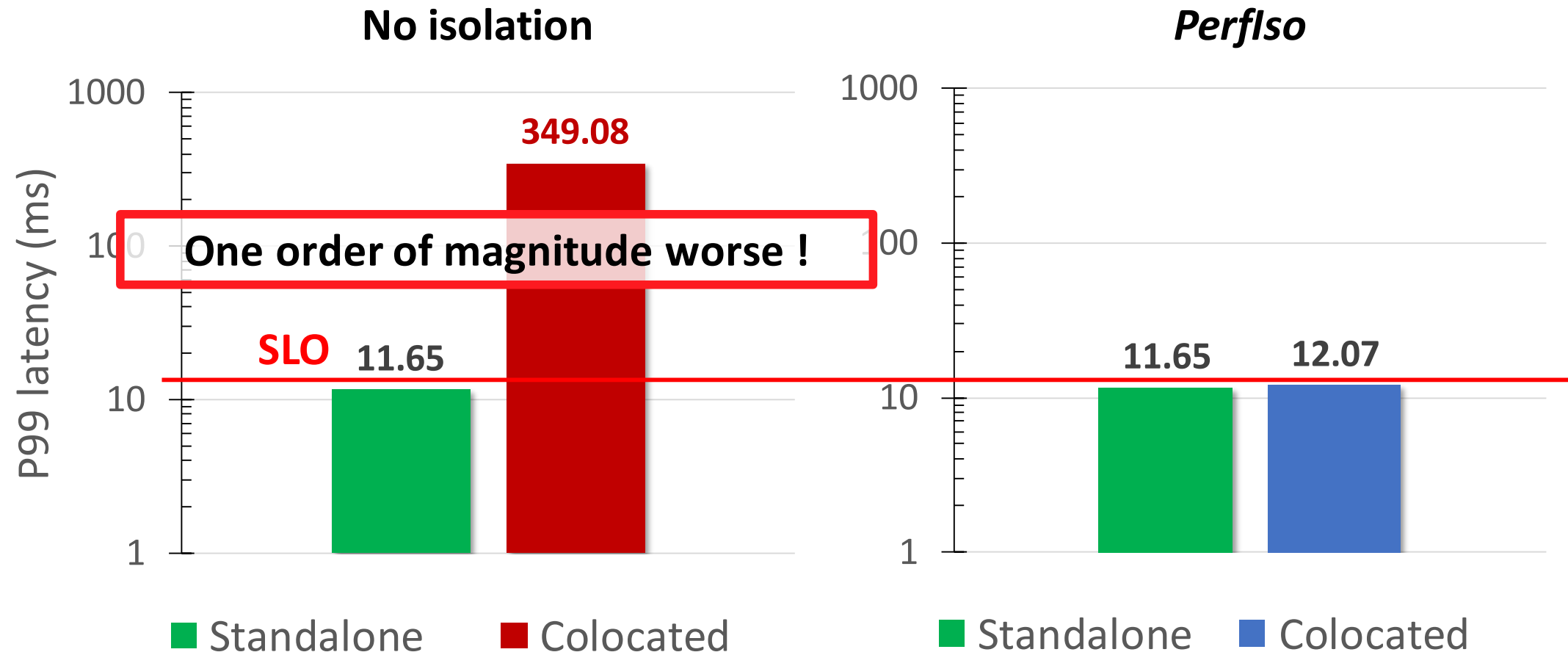
Single server: *PerfIso* protects tail-latency

Secondary: CPU-intensive micro-benchmark



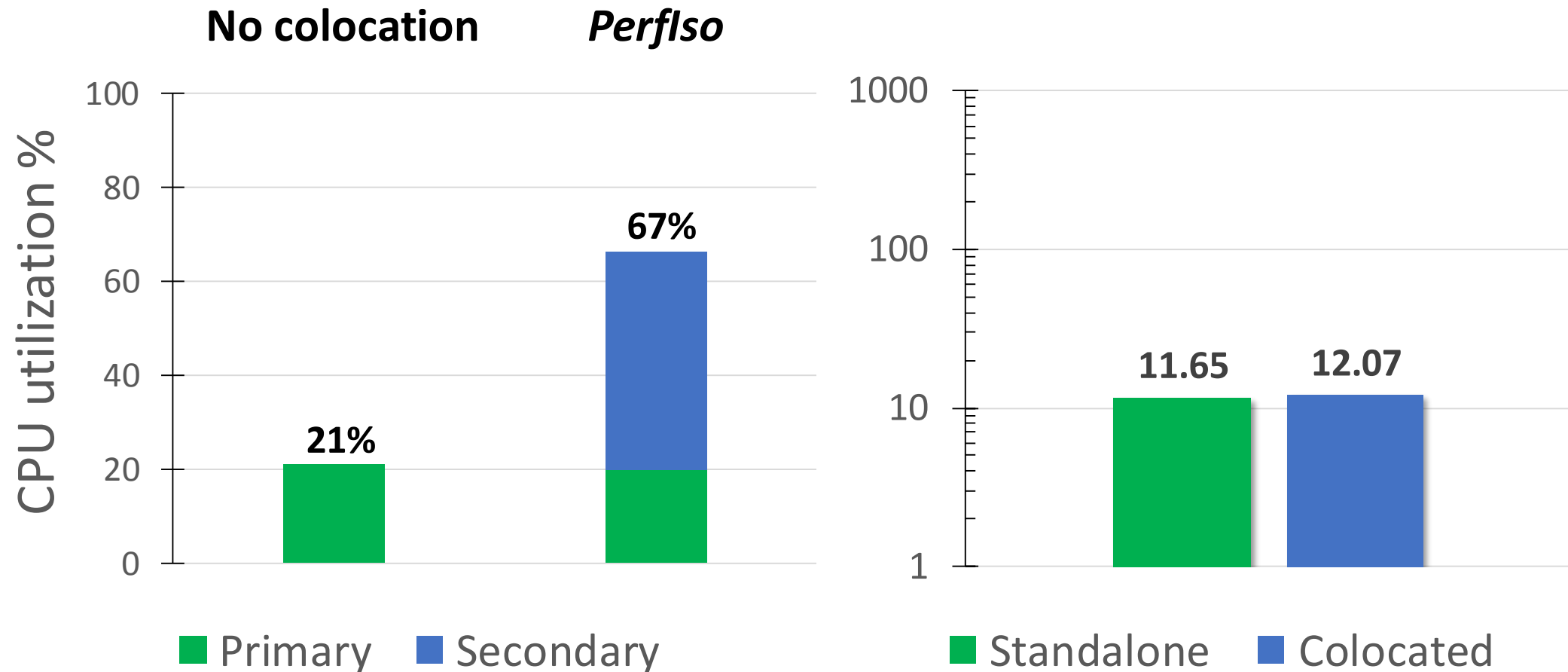
Single server: *Perflso* protects tail-latency

Secondary: CPU-intensive micro-benchmark



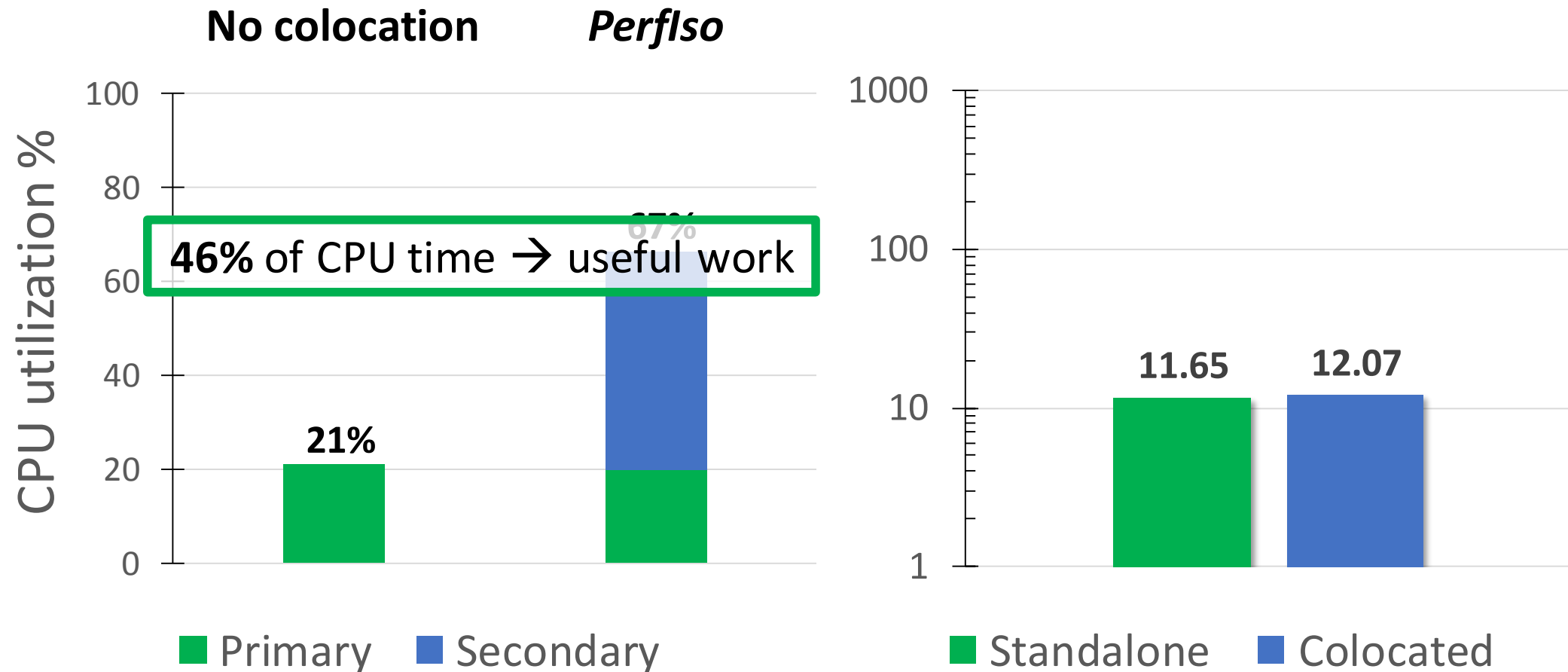
Single server: CPU utilization 3x higher!

Secondary: CPU-intensive micro-benchmark



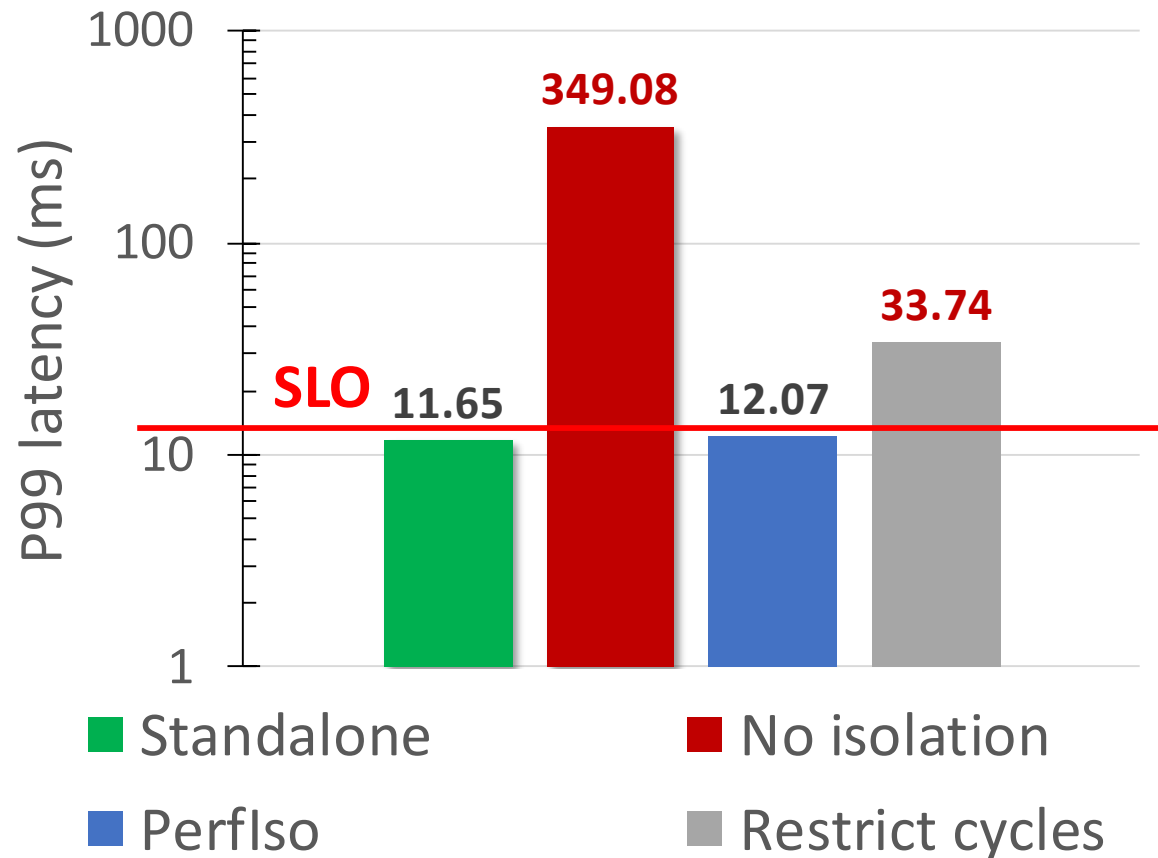
Single server: CPU utilization 3x higher!

Secondary: CPU-intensive micro-benchmark



Restricting CPU cycles does not work

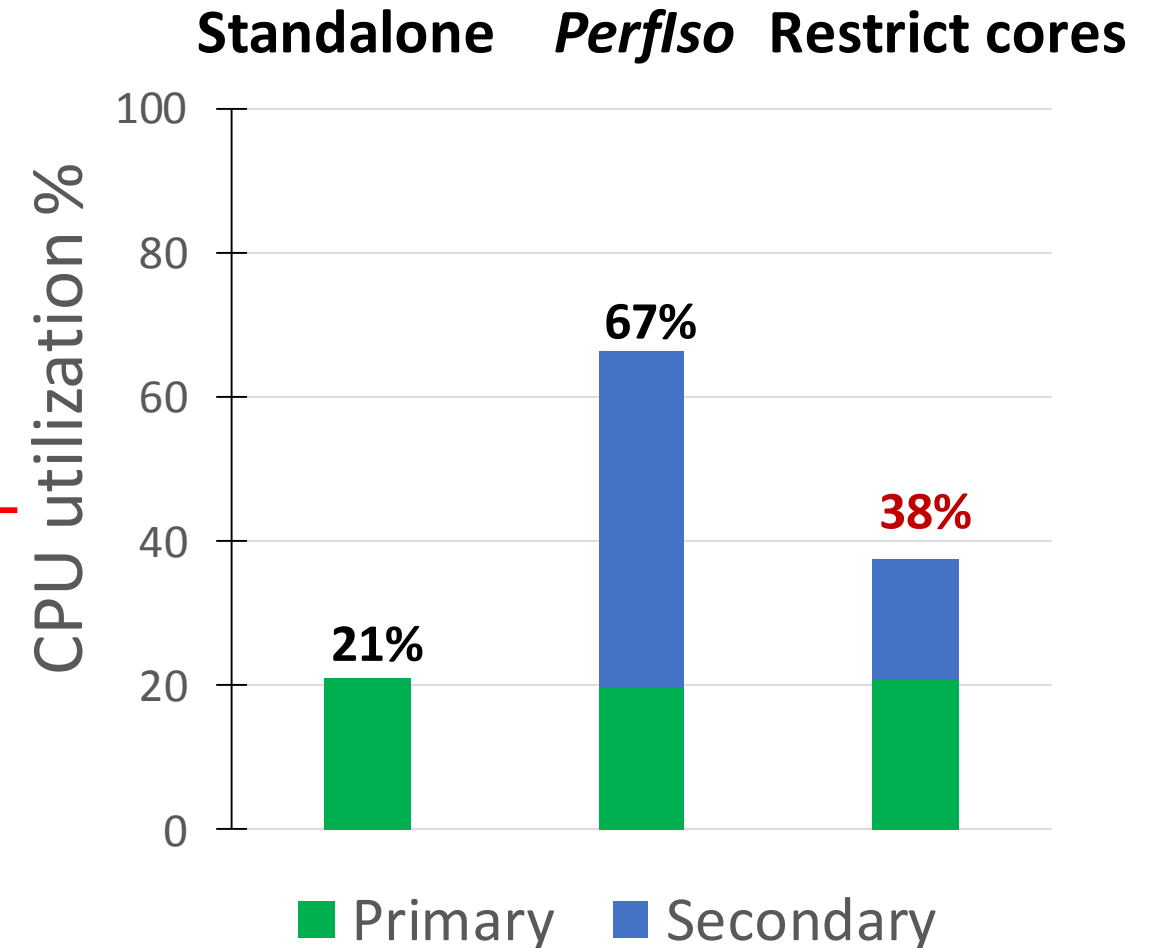
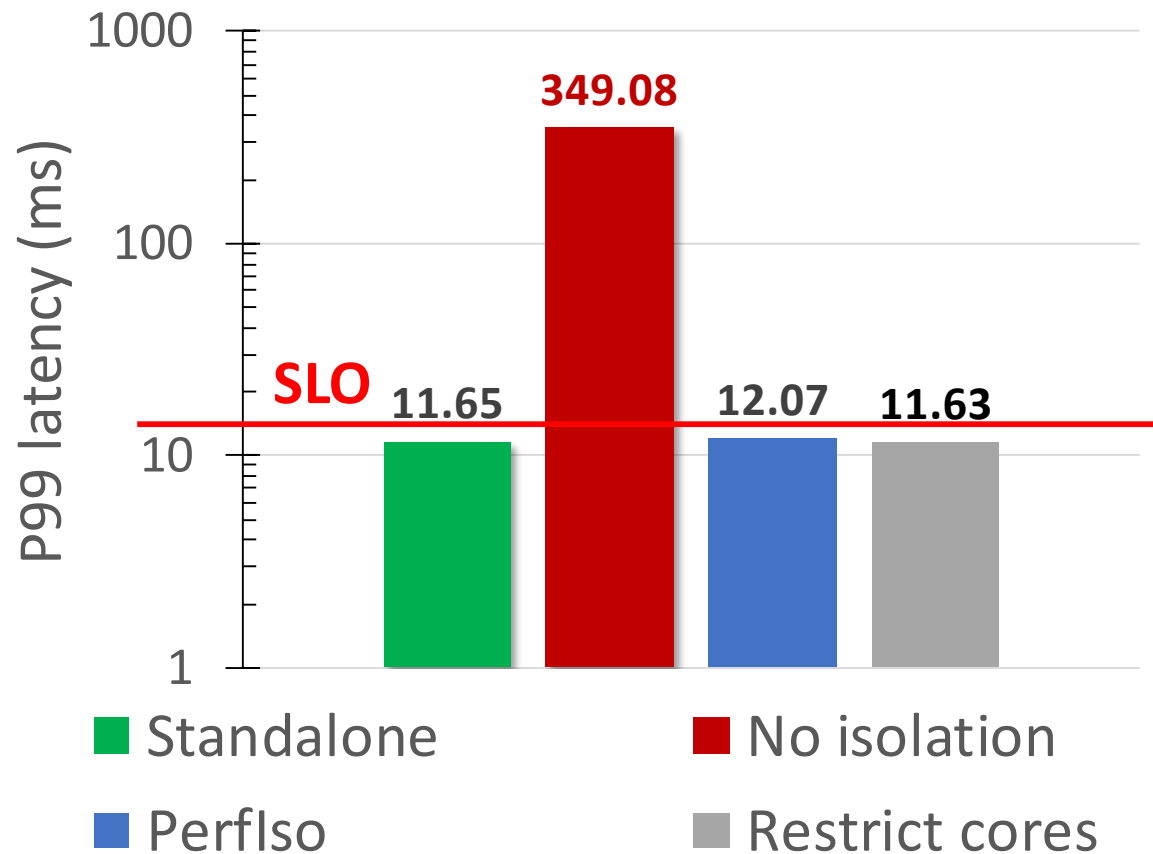
Secondary: CPU-intensive micro-benchmark



Secondary → 5% of CPU cycles
P99 latency – **3x higher than SLO!**

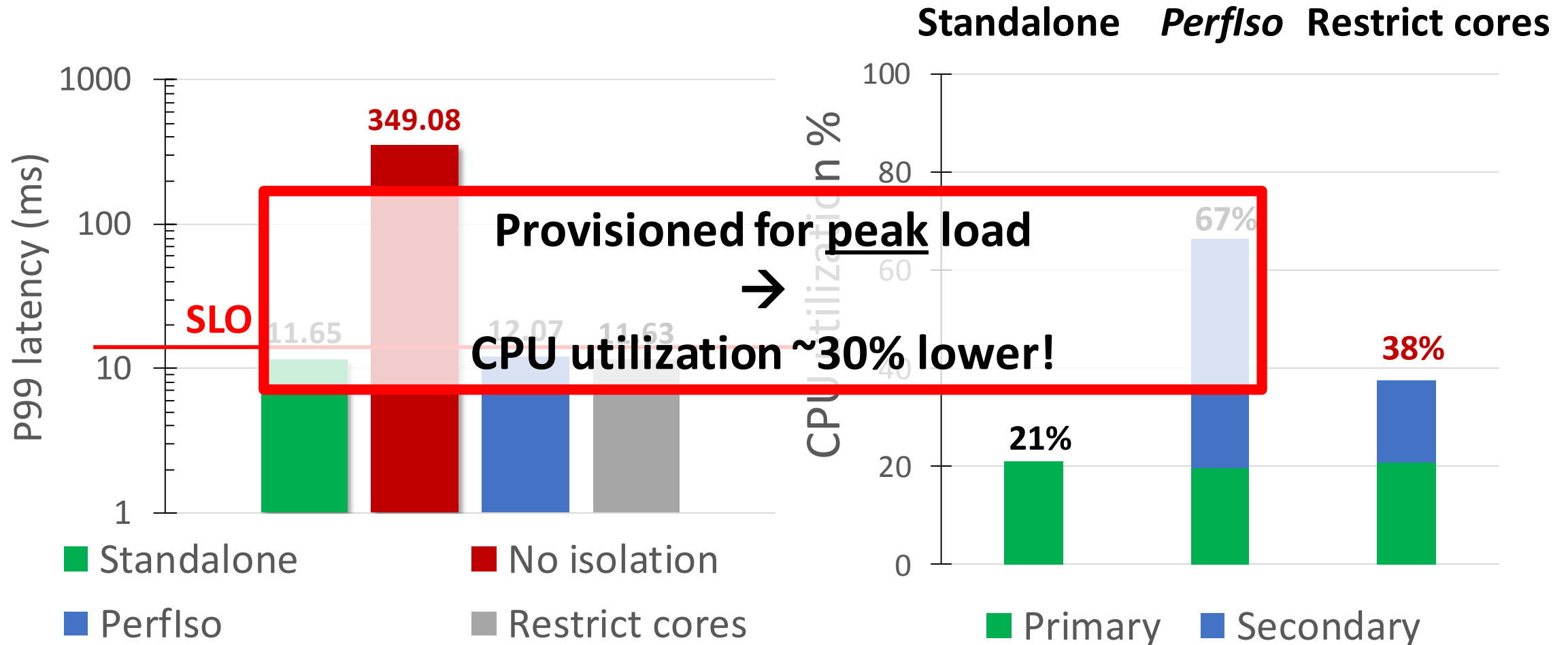
Restricting CPU cores does not work

Secondary: CPU-intensive micro-benchmark



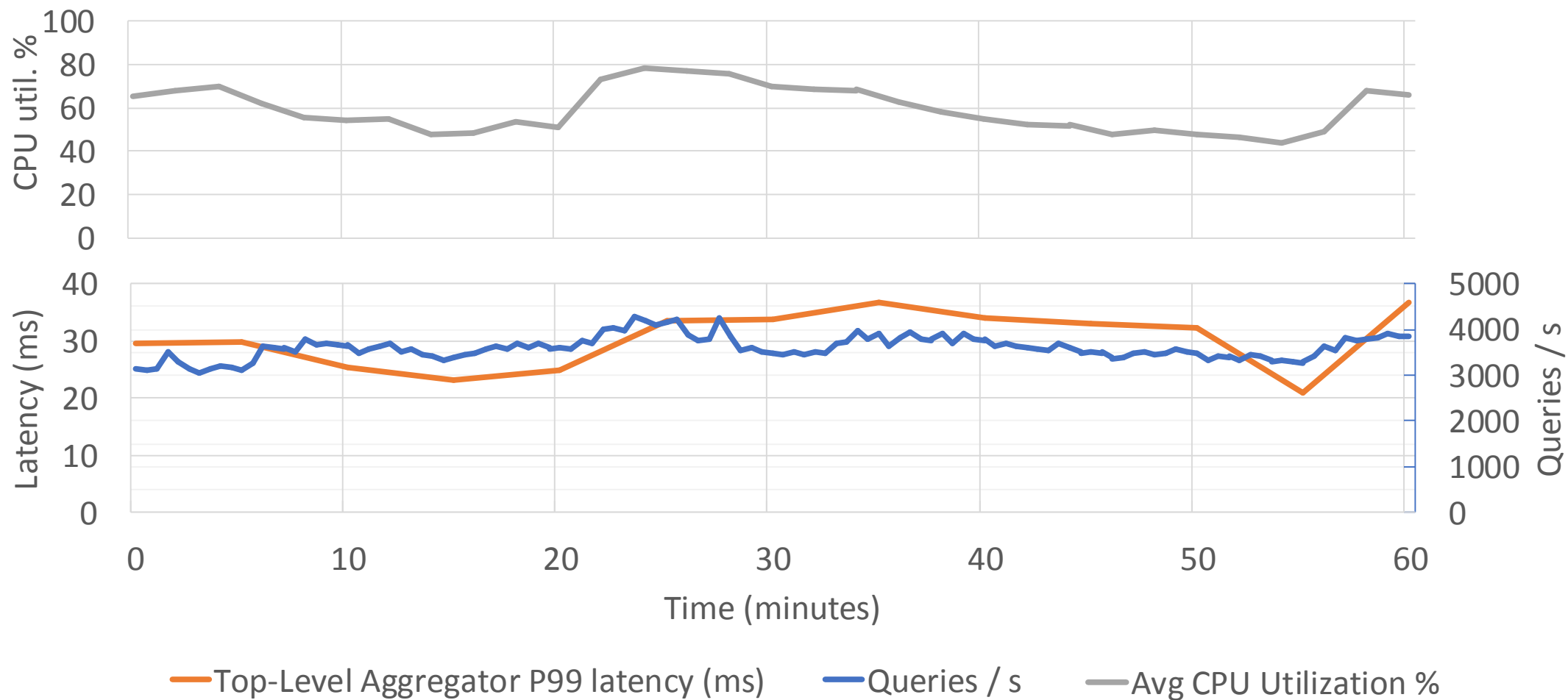
Restricting CPU cores does not work

Secondary: CPU-intensive micro-benchmark



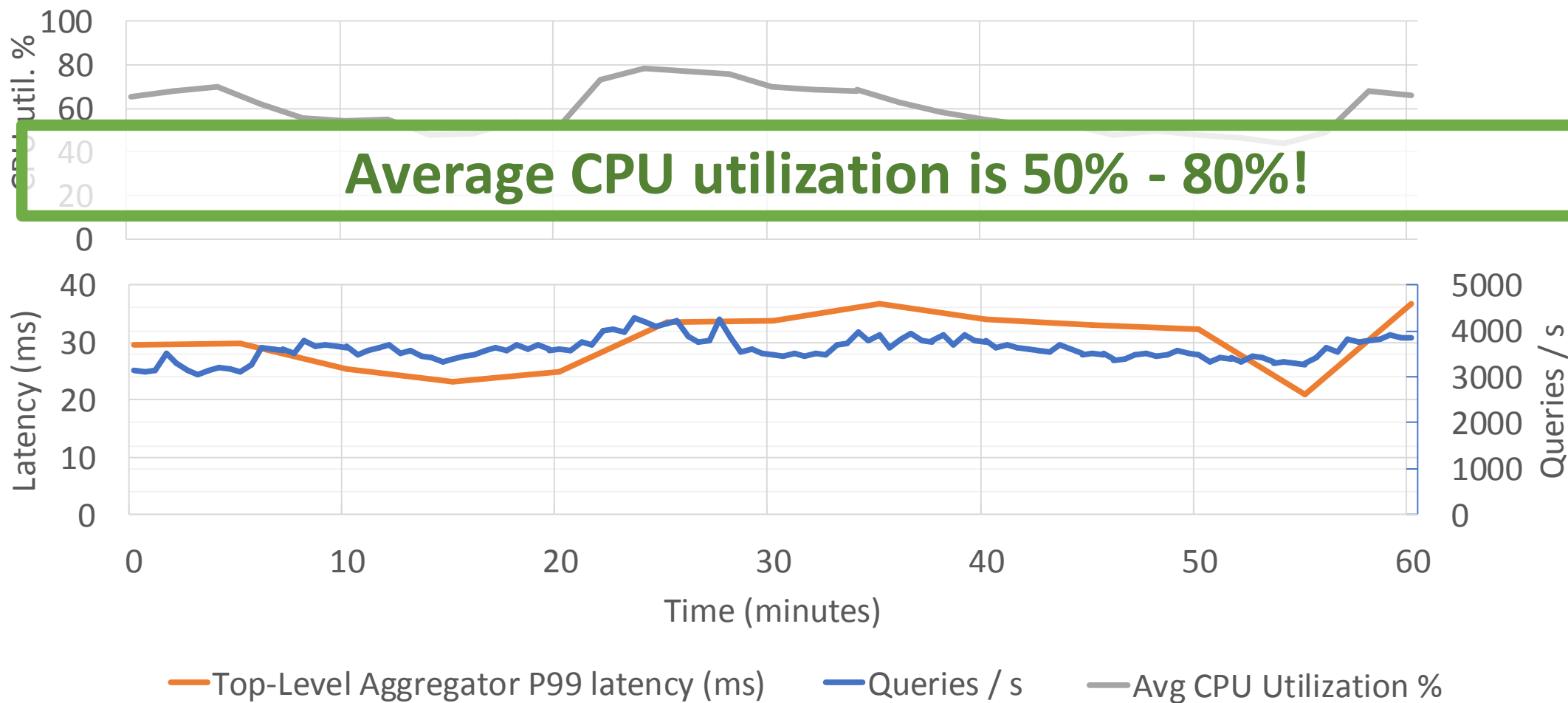
1-hour run of 650 machine cluster

Secondary: Machine-Learning computation



1-hour run of 650 machine cluster

Secondary: Machine-Learning computation



Interesting details in the paper

- Effectiveness of static CPU isolation methods
 - Restricting CPU cycles
 - Restricting CPU cores
- Comparison of state-of-the-art techniques
- Managing disk, memory, and network

PerfIso: Performance Isolation for Commercial Latency-Sensitive Services

Călin Iorgulescu* EPFL	Reza Azimi* Brown University	Youngjin Kwon* U. Texas at Austin	Sameh Elnikety Microsoft Research
Manoj Syamala Microsoft Research	Vivek Narasayya Microsoft Research	Herodotos Herodotou* Cyprus University of Technology	
Paulo Tomita Microsoft Bing	Alex Chen Microsoft Bing	Jack Zhang Microsoft Bing	Junhua Wang Microsoft Bing

Abstract

Large commercial latency-sensitive services, such as web search, run on dedicated clusters provisioned for peak load to ensure responsiveness and tolerate data center outages. As a result, the average load is far lower than the peak load used for provisioning, leading to resource under-utilization. The idle resources can be used to run batch jobs, completing useful work and reducing overall data center provisioning costs. However, this is challenging in practice due to the complexity and stringent tail-latency requirements of latency-sensitive services. Left unmanaged, the competition for machine resources can lead to severe response-time degradation and unmet service-level objectives (SLOs).

This work describes *PerfIso*, a performance isolation framework which has been used for nearly three years in Microsoft Bing, a major search engine, to colocate batch jobs with production latency-sensitive services on over 90,000 servers. We discuss the design and implementation of *PerfIso*, and conduct an experimental evaluation in a production environment. We show that colocating CPU-intensive jobs with latency-sensitive services increases average CPU utilization from 21% to 66% for off-peak load without impacting tail latency.

1 Introduction

New server acquisition contributes to over half of the total cost of ownership (TCO) of modern data centers [8]. However, server utilization is low in data centers hosting large latency-sensitive services for two main reasons: First, latency-sensitive services are typically provisioned for the peak load, which occurs only for a fraction of the total running time [18]. Second, business-continuity plans dictate tolerating multiple major data center outages, such as tolerating the failure of two data centers

* Work done while authors were at Microsoft Research.



Figure 1: Architecture of index serving system of Web search engine with two aggregation levels (MLA and TLA). The user query is processed on index servers, which send responses to MLAs, which send aggregated responses to TLA.

out of three data centers within a continent while remaining capable of processing peak load. The high degree of over-provisioning is imperative: a livensite incident causing brief downtime results in lost revenue and frustrated users, while an extended downtime comes with negative headline news and irreparable business damage. Even slightly higher response times decrease user satisfaction and impact revenues [29, 10, 17].

Over-provisioning means that resource utilization is low, offering the opportunity to colocate batch jobs alongside latency-sensitive services [32, 18]. Colocation must be managed carefully lest it degrades performance due to competition on machine resources. Our main goal is to ensure that the end-to-end service-level objectives (SLOs) are met while increasing the work done by batch jobs. The main technical challenges arise from maintaining short tail latency (e.g., the 99th latency percentile also called P99 latency) for the latency-sensitive services coupled with the complexity of commercial software and large deployments.

Oftentimes the service-level-objectives are not known explicitly for each individual component. For example, large commercial search engines contain tens of plat-

***PerfIso*: colocate batch jobs with online services**

PerfIso: colocate batch jobs with online services

- **Black-box**: do not tailor to one specific service

PerfIso: colocate batch jobs with online services

- **Black-box**: do not tailor to one specific service
- **Robustness**: favor user-mode over kernel implementation

PerfIso: colocate batch jobs with online services

- **Black-box**: do not tailor to one specific service
- **Robustness**: favor user-mode over kernel implementation
- **Headroom**: some core-slack makes Primary behave like standalone

PerfIso: colocate batch jobs with online services

- **Black-box**: do not tailor to one specific service
- **Robustness**: favor user-mode over kernel implementation
- **Headroom**: some core-slack makes Primary behave like standalone
- CPU Blind Isolation → colocation without impacting service performance