# Granular Resource Demand Heterogeneity

Yizhuo Liang
University of Southern California
Los Angeles, USA

Ramesh Govindan
University of Southern California
Los Angeles, USA

Seo Jin Park
University of Southern California
Los Angeles, USA

## Abstract

Granular resource heterogeneity refers to the phenomenon in which small computational units within or across applications exhibit distinct resource usage patterns. Traditional resource management in shared clusters lumps monolithic applications into coarse categories, overlooking smaller execution phases that differ in resource demands.

We introduce `hiresperf`, an extensible profiler that investigates resource usage at 10-microsecond intervals attributed to each function invocation, with a low overhead. We show that using `hiresperf` for either offline analysis or real-time monitoring can exhibit the granular resource heterogeneity without manually decomposing applications for profiling. Armed with the fine-grained insights, resource managers, batch schedulers, and serverless runtimes can proactively schedule and migrate tasks to minimize interference and boost utilization at the same time.

## CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Scheduling**.

## Keywords

profiling, granular computing, distributed systems, serverless, cloud computing

**ACM Reference Format:**
Yizhuo Liang, Ramesh Govindan, and Seo Jin Park. 2025. Granular Resource Demand Heterogeneity. In *Workshop on Hot Topics in Operating Systems (HOTOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730379

## 1 Introduction

Modern applications are increasingly deployed in shared clusters. These are managed by infrastructure providers, who often co-locate multiple programs on the same machine. Cluster resource management involves reserving virtual CPUs (vCPUs) and memory capacity for each application. Beyond these allocations, applications also utilize shared system resources (*e.g.*, DRAM and I/O bandwidths). Different applications may have widely-varying demands on these shared resources; we call this *resource demand heterogeneity* [1].

Understanding resource demand heterogeneity can help minimize resource contention for co-located applications. To determine how programs contend for memory, L3-cache, I/O bandwidth, *etc.*, current techniques measure resource usage at runtime [8, 23, 37, 40]. Often, these approaches measure aggregated resource usage at coarse timescales (*e.g.*, one minute) [8], and feed this data to cluster schedulers that manage resource allocation at the granularity of monolithic applications or jobs.

In this paper, we assert that aggregated resource usage monitoring is a poor fit for efficient resource allocation in shared clusters of the future. Increasingly, these clusters offer computing frameworks such as serverless computing and resource disaggregation [20, 32, 35, 36, 41]. In these frameworks, applications are decomposed into granular computing tasks, which permits greater flexibility for task scheduling on shared resources, and thereby achieves higher utilization and scalability. In this setting, resource demand heterogeneity may exist not just across applications, but also across granular computing elements (*i.e.*, pieces of a program, such as individual functions, either of monolithic applications or native serverless tasks). Moreover, these individual functions may execute for short durations (milliseconds to a few tens or hundreds of microseconds).

In this paper, we explore *granular resource demand heterogeneity* — heterogeneity in resource requirements of an application's granular components at fine timescales. For applications exhibiting such heterogeneity, we ask: can we simultaneously achieve low interference and high utilization?

To demonstrate granular heterogeneity, consider a full-text search server built with a C++ port [42] of Apache Lucene [4]. The server indexes input text and holds it in memory, then serves search queries. One component of a Lucene `search` is the `createWeight` function. This parses the query to build an internal representation of rules for evaluating relevance. Then the `score` function iterates through the indexed documents to perform evaluations.

[1]In contrast, the diversity of resource offerings across different hardware is referred to as *hardware heterogeneity*.

ACM ISBN 979-8-4007-1475-7/2025/05
https://doi.org/10.1145/3713082.3730379

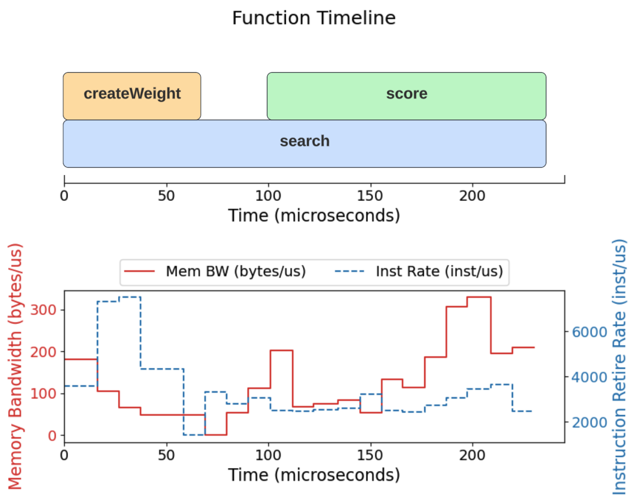

**Figure 1:** Hiresperf-generated timeline of a Lucene++ `search` invocation. Effective stack sampling interval about 5 $\mu$s, effective PMUs polling interval about 11 $\mu$s.

| Function | Time ($\mu s$) | MemBW (bytes/$\mu s$) | Inst per $\mu s$ |
|---|---|---|---|
| `createWeight` | 18.5 | 117 | 4395 |
| `score` | 71.1 | 218 | 2909 |

**Table 1:** Resource usage of Lucene++ functions profiled by hiresperf, averaged from an experiment of 460K query requests

During a search query, memory bandwidth and instruction pipeline usage change over time, as the CPU frequency is constant. Figure 1 demonstrates this for a `search` operation taken from a corpus of 40K searches. Table 1 shows the averaged resource usage metrics of these recorded functions. Each function executes for tens of microseconds, but has widely varying resource usage. In Table 1, `createWeight` exhibits a lower average DRAM access rate and higher instruction rate, indicating its job is more computation-heavy. Conversely, `score` uses roughly double the DRAM bandwidth and has 34% less instruction rate than `createWeight`. `score` executes fewer instructions per unit time because the chip's pipeline resources are less utilized partially due to more operations blocked by memory accesses. In fact, `score` incurs 33% more cycle stalls (not shown) than `createWeight` when there are pending memory operations [2]. This example demonstrates granular **intra-application heterogeneity** of resource demands, as distinct from **inter-application heterogeneity** that has long been exploited. Notably, obtaining resource usage at such fine timescales is itself a challenge; we discuss later how we obtain this without executing each function in isolation.

[2]Although this performance counter does not imply a causal relationship between memory access and stalls, it helps differentiate between computation-bound and memory-bound program pieces

Beyond monolithic applications, granular resource heterogeneity also naturally arises in serverless computing, where sharing instances among small tasks is commonplace. For instance, a video encoding task [10] and a 3D-printing heat control [31] task, both leveraging Function-as-a-Service (FaaS) platforms, have vastly different demands for network bandwidth. Specifically, the former needs to transmit video frames of up to tens of megabytes, whereas the latter only transmits tens of kilobytes of vectors after initialization.

Given this, we argue that to better control resource contention and increase utilization, cluster management should be *granular heterogeneity-aware*, *i.e.*, aware of the fine timescale resource needs of individual functions of monolithic or serverless applications. To achieve this, we must address two questions: (1) how do we effectively track and estimate granular resource usage and (2) how can cluster schedulers exploit knowledge of granular heterogeneity to control contention and improve utilization.

For the first question, we present `hiresperf`, a lightweight software profiler designed for measuring intra-application heterogeneity (Section 3). For the second, we discuss (Section 4) promising use cases and compare them with existing resource management approaches.

## 2 Background and Motivation

We begin by describing related work in interference-aware cluster management, then make the case for efficient techniques to measure granular heterogeneity.

### 2.1 Interference aware cluster management

In recent years, as the number of CPU cores on a single chip has continued to increase, shared compute platforms have increasingly begun co-locating applications. Orchestrating co-located applications usually involves reserving a pre-defined number of vCPUs and memory. Even with reserved resources, co-location can result in unpredictable performance degradation [24], due to contention for shared architectural resources and I/O bandwidth. In response to this, prior work has established the need for understanding heterogeneous resource demands among co-located applications. This line of work has extensively investigated techniques to characterize shared system resources required by different applications as well as to quantify how each is sensitive to contention. Scheduling and orchestration techniques can use these characterizations to minimize contention.

However, much of this work has focused on coarse-grain resource heterogeneity measurements for monolithic applications. For example, Paragon [8], estimates resource demand heterogeneity for programs aggregated over a minute or longer. In their setting, this was necessary since applications are multi-faceted in terms of resource consumption, and a longer profiling interval can cover more program phases or events. NuCore [40] has explored shorter timescales (*e.g.*, 50 milliseconds), but it still characterizes an application as a homogeneous entity. Even though these aggregated metrics ignore fine-grained fluctuations within the application, they were sufficient for earlier cluster schedulers, since individual resource usage spikes of individual threads were often statistically multiplexed across hundreds of concurrent threads, resulting in minimal system-wide impact. As such, these coarse-grained characterizations were sufficient for cluster management systems to decide which applications should share an instance to control the contention levels [8, 23].

More recent work has demonstrated the existence of demand heterogeneity at finer timescales *within* specific applications and explored ways to exploit it. For example, a DNN training task often alternates between computing and data loading. So a scheduler [39] can overlap one task's peak computing and memory capacity with another task's peak memory bandwidth. Similarly, Spark defines a high-level data processing workflow so that each Spark task can be transiently decomposed into computing, disk, and network IO phases, revealing scheduling opportunities to minimize resource contention [29]. However, to the best of our knowledge, such intra-application heterogeneity has not been exploited for scheduling generic computing tasks.

## 2.2 Granular and serverless computing frameworks

With the advent of FaaS, recent research has proposed ways to decompose monolithic applications into small pieces (stateless or stateful) that don't all reside on specific server or virtual machine (VM) instance [32, 33, 38]. These approaches allow instantaneous on-demand scaling, fast migration, and the ability to exploit "otherwise-stranded" resources (*e.g.*, the idle CPU cores on an instance whose memory is almost full) [32]. These features improve cluster-wide resource utilization and achieve high elasticity for the applications. High-performance computing researchers have considered similar ideas [35], in which tasks can dynamically scale at the granularity of threads and migrate across VMs shared by multiple applications.

However, the inherent diversity among serverless-style tasks has not received much attention [18]. Major state-of-the-art open-source FaaS platforms rely on simple vCPU and memory capacity quotas set by the user [3, 19, 27]. Despite these quotas, co-located FaaS applications can contend for shared resources (*e.g.* memory bandwidth), resulting in degraded performance. This remains one of the biggest obstacles to QoS-centric FaaS use cases, next to cold-start latency [13, 36, 45].

## 2.3 The need for novel profilers for granular heterogeneity

Characterizing granular heterogeneity, especially intra-application heterogeneity, remains a challenge. Potentially, one could decompose and micro-benchmark individual components to obtain fine-grained characterization, but this approach is usually cost-prohibitive at scale and requires significant manual effort. As such, only extremely popular applications use this approach because their use justifies the cost [12]. Moreover, this approach can be inaccurate since it may not capture the impact of synchronization events (*e.g.*, lock waits). An ideal approach would characterize granular heterogeneity after an application is deployed, and with real workloads. Such an approach would *directly profile* monolithic applications with low overhead.

This *granular heterogeneity profiler* should satisfy three objectives. It should:

O1: Collect resource usage data at fine time-scales (down to microseconds),
O2: Attribute each data point to one or more program pieces invoked by specific corresponding threads,
O3: Incur low-enough overhead to *not* perturb program behavior, and permit real-time monitoring in production.

Today, state-of-the-art profilers like Intel VTune [15], Linux perf, and OProfile [28] rely on hardware performance monitoring units (PMUs) [16] to reveal the utilization of architectural resources. However, these profilers do not satisfy the requirements listed above. Linux perf's statistical sampling cannot distinguish different invocations of the same function, so it cannot detect demand heterogeneity within a single function called during different program stages (Section 3.2). Thus, perf fails to satisfy O2. VTune does support timeline analysis, but it incurs high overhead (Figure 4) for short sampling intervals, thus failing O3. Moreover, VTune is proprietary software and may not be easy to integrate into larger resource management systems. Shim [43] achieves ultra-fine resolution with negligible program perturbation. However, Shim is designed for hand-tuning program performance rather than estimating demand heterogeneity and occupies all hyper-thread siblings for polling core-specific performance counters, which is prohibitively expensive for profiling deployed workloads.

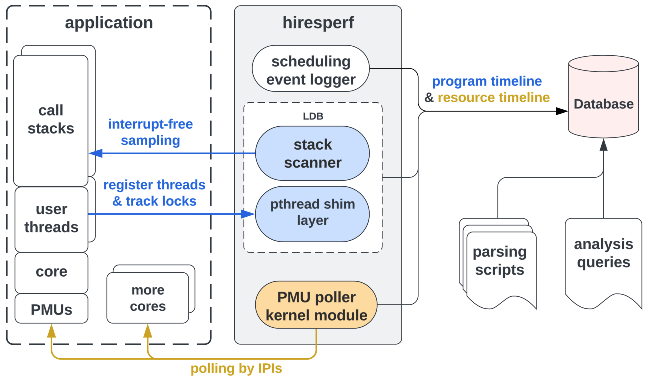

**Figure 2:** Hiresperf overview. Resource monitoring is in yellow and program tracing is in blue.

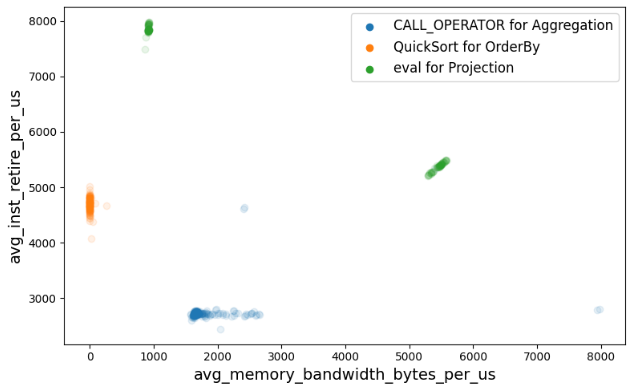

**Figure 3:** Averaged resource usage metrics for every function invocation of a velox program. All records of `QuickSort` and `eval` are shown. `CALL_OPERATOR` data points are randomly sampled from 15K recorded invocations.

## 3 Hiresperf: A Granular Resource Profiler

In this section, we describe `hiresperf`, a profiler specialized for measuring granular resource heterogeneity. Hiresperf satisfies the objectives listed in Section 2.3. Currently, it monitors each function invocation's memory bandwidth, instruction retiring rate, and CPU frequency. Hiresperf can be extended to monitor caches and I/O usage, but we have left this to future work.

### 3.1 Hiresperf design

Like VTune [15], Hiresperf also uses hardware PMUs to monitor resource usage. To ensure extensibility, permit both offline or real-time monitoring, and to support multiple programming languages, we decouple its resource monitoring and program tracing components, as shown in Figure 2.

For resource data, we use a dedicated Linux kernel module that periodically launches inter-processor interrupts (IPIs) to the monitored set of CPU cores. Each IPI triggers an interrupt handler, which reads the core's PMU counters and writes the timestamped results to a per-core buffer. On all Intel Xeon CPUs since Skylake, the handler can poll last-level cache misses and software prefetches to estimate [3] the core's DRAM bandwidth usage. On Sapphire Rapids and most newer microarchitectures, hiresperf can be configured to poll offcore counters [4] for better accuracy. In practice, this gives less than 5% error compared to Intel PCM [14]'s results.

For tracking control flows, we adopt the *stack sampling* mechanism from LDB [7]. The stack scanner of LDB can unwind each thread's stack without interrupts. Compiled with LDB-instrumented Clang, C/C++ programs write a canary and a generation number in each stack frame, for avoiding data races and distinguishing between consecutive occurrences of the same stack frame. Carefully designed to minimize cache-thrashing and to avoid data races, stack sampling effectively constructs a wall-clock timeline of every single function invocation. By consulting the scheduling history, hiresperf can annotate this per-thread timeline with the information of the core(s) a thread runs on.

We also developed a toolkit to efficiently consolidate the timelines and facilitate various analyses. Furthermore, hiresperf is modular. The PMU polling component generally works for all Intel platforms and is agnostic to the program language. The stack sampling component can be replaced by other language-specific tracing systems or even a tracer integrated into a serverless runtime, for example.

### 3.2 Hiresperf in Action

To demonstrate how hiresperf reveals granular heterogeneity (achieving O1 and O2), we use a data analysis example program built on the velox [30] execution engine. The program involves aggregations, projections, and OrderBy operations running on different threads, and hiresperf records resource metrics associated with the functions performing these operations. For simplicity, threads are pinned to physical cores and the small number of cores does not saturate memory bandwidth. As shown in Figure 3, one data point stands for one function invocation, and functions of different operations form several clusters in the memory vs. compute space. These clusters show how operations differ from each other in terms of resource utilization. Notably, there are two clusters of the function `eval`, which is used for two distinct projection queries, and one of them consumes about 5× more

[3]The accuracy varies on different CPUs. In our experiments, the last-level cache miss counters on Intel(R) Xeon(R) Gold 5420+ provided very accurate estimations, but Intel(R) Xeon(R) Platinum 8380 (Chameleon's compute_icelake_r750 instance) leaded to values that largely deviated from the PCM collected results, despite the profiled program did not change.
[4]Namely, we use `OCR.READS_TO_CORE.DRAM` for reads and `OCR.MODIFIED_WRITE.ANY_RESPONSE` for writes.

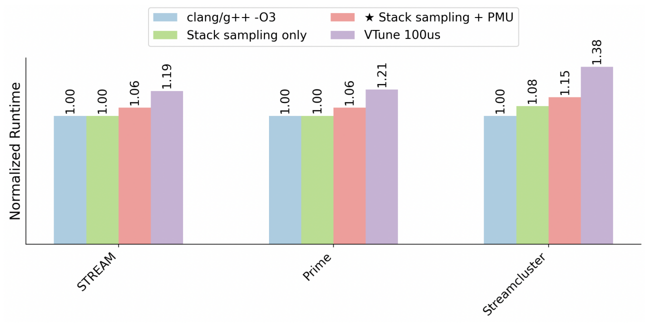

**Figure 4:** Hiresperf overheads on different apps. STREAM benchmark [25] is memory bandwidth aggressive, the prime checker is pure computing, and Streamcluster is in the middle. Note that VTune's sampling interval is set to 100us, an order of magnitude longer than hiresperf's. Also, VTune 2024.3.0 command line cannot customize performance event selection.

DRAM bandwidth than the other. This illustrates the need for O2.

When running this test, hiresperf has effective stack sampling interval of 5 $\mu$s and PMU polling interval of 11 $\mu$s. With this frequency, it can easily capture the behaviors of short-lived functions like `eval`, whose average runtime is only 179 $\mu$s.

## 3.3 Towards Greater Efficiency

We evaluate the overhead of hiresperf, with the same measurement intervals as the above example. Figure 4 shows overheads imposed on three different programs, which range from 6-15%. In contrast, VTune overheads with a periodicity of 100 $\mu$s are between 19-38%. Of these three programs, Streamcluster [5] experiences more overhead from stack sampling, because stack scans are more frequent for a program with higher function churn. In general, for offline analysis, this overall overhead is still acceptable for this fine a profiling resolution.

To use hiresperf for real-time monitoring, it may be possible to reduce the overhead. In our current implementation, 99% of PMU polling overhead comes from interrupts alone. We can actually avoid interrupts if the program or the serverless runtime polls the PMUs from within user-space and voluntarily reports the results. One practical way to achieve such *interrupt-free profiling* is similar to the compiler instrumentation approach of stack sampling: reading performance counters upon function call/return, and then writing the values within the stack frames so that the stack scanning component can collect these metrics while tracing function calls. Initial experiments show that, voluntarily reporting PMUs generates no measurable overhead beyond stack sampling alone.

## 4 Future Directions for Exploiting Granular Resource Heterogeneity

Hiresperf's granular resource monitoring can enable at least two use-cases, which we sketch below.

### 4.1 Resource management and scheduling

**Resource allocation for QoS-centric systems.** User-facing, latency-critical (LC) applications, especially ones with varying workloads, can be prone to interference. Reserving extra resources to minimize interference hurts overall utilization. Thus, systems like Caladan [11] and Heracles [21] dynamically allocate resources to LC and best-effort (BE) tasks. Seeking both utilization and QoS attainment, their resource schedulers penalize the BE tasks when LC applications incur increased latency. However, such *reactive* approaches have limitations, as they rely on the premise that a sufficient number of applications are BE, and throttling resource allocations during contention periods does not prevent the resource from being underutilized at other times.

Using hiresperf's information about granular resource heterogeneity, a better approach is to *proactively* schedule tasks that demand complementary resources together with the LC applications so that they are unlikely to contend. If a resource scheduler detects higher response latencies due to a workload spike, it can trigger migrations of the granular programs [32, 33] to fit into a new set of resources with residual capacity. We believe such near-real-time, predictive decisions are feasible, since Caladan has demonstrated decision times of less than 20 $\mu$s when the scheduler runs on a dedicated core. The challenge lies more in synchronizing the resource availability among local schedulers in a timely manner.

**Scheduling for batch processing systems.** Compute-intensive workloads consisting of thousands of small jobs are often deployed on batch processing platforms, including cloud services like Azure/AWS Batch [1, 26] and on-premise clusters managed by HPC schedulers like Slurm [34]. Also, big-data processing frameworks [2, 44] typically have dedicated clusters. Prior work has identified contention due to job co-location [6, 22], and developed methods to predict the interference to guide the job placement. However, such placements remain static until a job finishes, although the prediction models are trained with the awareness of resource utilization varying over time. With hiresperf's granular per-task measurements, schedulers can rearrange placement of in-flight tasks to minimize contention. For example, a recently-proposed batch processing runtime, GRANNY [35], can support such migration operations at synchronization barriers within a job.

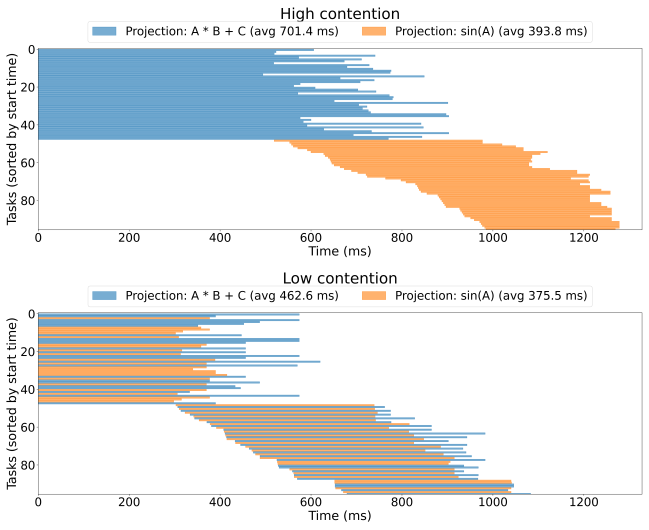

**Figure 5:** Simulating two scheduling decisions of heterogeneous velox tasks.

If migrations are infeasible due to large amounts of data associated with the jobs, we still have some flexibility to schedule the tasks on a machine, as batch jobs have less stringent latency requirements. As an example, Figure 5 shows two schedules consisting of two velox operations: the $A * B + C$ projection operation which is 5× more memory bandwidth intensive than the $sin()$ projection. In our example, each operation type has 48 tasks to run. Compared to scheduling one operation completely after the other, overlapping tasks with heterogeneous resource demands reduces the average runtime of the DRAM-intensive task by more than 34% and the makespan by more than 15%. Of course, other factors of scheduling (*e.g.*, starvation-proofing) should be considered in practice, along with contention-avoidance strategies.

Fine-grained scheduling has also been studied without profiling. Monotasks [29] demonstrate the benefits of partitioning a large task into pieces, with each piece consuming a single resource type, to achieve performance clarity. Monotasks have been applied to Spark to better expose resource usage and sometimes reduce contention. However, applying this to general applications is challenging and involves significant programmer effort. Fine-grained profilers like hiresperf may help quantify the bottlenecks of each task automatically to simplify the process of designing monotasks for an application.

### 4.2 Profiler-aided granular disaggregation

To enjoy the benefits mentioned above, monolithic applications need to be partitioned in the first place. This is made possible by granular computing platforms like Quicksand [32]. Typically, the first factor considered for such partitioning is how much data (memory) different components share. For example, the most bulky data of our Lucene server, the per-document term frequencies, is accessed by `score` but not `createWeight`. This implies no excessive state replication or network traffic will be introduced due to separating these two components.

Aside from shared state, differences in resource demands are also important factors to consider when partitioning apps, especially in a large cluster with heterogeneous hardware. The rationale is simple: if the components are bottlenecked on different resource types, they may run faster on hardware that best fits their needs. For example, a memory-intensive component can run faster on an instance with more memory channels, while another component that iterates over data with high cache-hit rates can run faster on an instance with higher CPU frequency, *etc.* Similarly, depending on the program's potential for exploiting out-of-order execution, some components can benefit more from more CPU pipeline resources. Granular heterogeneity profilers like hiresperf can identify whether components are memory-, cache-, or pipeline-intensive. Collectively, such partitioning can improve performance, cost efficiency, and energy efficiency in the future.

## 5 Conclusion

Granular resource heterogeneity offers new avenues for responsive, cost-effective, and scalable computing. Our prototype `hiresperf`[5] demonstrates how to explore microsecond-level resource demands. We also discussed how such insights can guide scheduling, dynamic resource sharing, and modular disaggregation. Future endeavors can harness these techniques for more efficient and powerful platforms.

## Acknowledgments

We would like to thank Yibo Yan, Inho Cho, and HotOS 2025 reviewers for valuable discussions and feedback. We appreciate CloudLab [9] and Chameleon [17] for providing the experiment platform.

[5]GitHub: https://github.com/yizhuoliang/hiresperf

## References

[1] Amazon Web Services. 2025. Efficient Batch Computing – AWS Batch. https://aws.amazon.com/batch/. Accessed: 2025-01-16.

[2] Apache Software Foundation. 2025. Apache Hadoop. https://hadoop.apache.org/. Accessed: 2025-01-13.

[3] Apache Software Foundation. 2025. Apache OpenWhisk Is a Serverless, Open Source Cloud Platform. https://openwhisk.apache.org/. Accessed: 2025-01-09.

[4] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. 2012. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*. 17.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128 https://doi.org/10.1145/1454115.1454128.

[6] David Buchaca, Joan Marcual, Josep Lluis Berral, and David Carrera. 2020. Sequence-to-Sequence Models for Workload Interference Prediction on Batch Processing Datacenters. *Future Generation Computer Systems* 110 (Sept. 2020), 155–166. https://doi.org/10.1016/j.future.2020.03.058 https://www.sciencedirect.com/science/article/pii/S0167739X19310921.

[7] Inho Cho, Seo Jin Park, Ahmed Saeed, Mohammad Alizadeh, and Adam Belay. 2024. {LDB}: An Efficient Latency Profiling Tool for Multithreaded Applications. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1497–1510. https://www.usenix.org/conference/nsdi24/presentation/cho.

[8] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. (2013).

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[10] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Anirudh Sivaraman, and George Porter. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. (2017).

[11] Joshua Fried, Zhenyuan Ruan, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. (2020).

[12] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. 2023. Profiling Hyperscale Big Data Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 47, 16 pages. https://doi.org/10.1145/3579371.3589082

[13] M. Reza HoseinyFarahabady, Albert Y. Zomaya, and Zahir Tari. 2018. A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform. *IEEE Transactions on Parallel and Distributed Systems* 29, 7 (July 2018), 1442–1455. https://doi.org/10.1109/TPDS.2017.2779502 https://ieeexplore.ieee.org/document/8126823/?arnumber=8126823.

[14] Intel Corporation. 2025. Intel PCM – Performance Counter Monitor. https://github.com/intel/pcm. Accessed: 2025-01-11.

[15] Intel Corporation. 2025. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. Accessed: 2025-01-08.

[16] Intel Corporation. 2025. PerfMon Events. https://perfmon-events.intel.com/. Accessed: 2025-01-10.

[17] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[18] Jeongchul Kim and Kyungyong Lee. 2020. I/O Resource Isolation of Public Cloud Serverless Function Runtimes for Data-Intensive Applications. *Cluster Computing* 23, 3 (Sept. 2020), 2249–2259. https://doi.org/10.1007/s10586-020-03103-4 https://link.springer.com/10.1007/s10586-020-03103-4.

[19] Knative Project. 2025. Home – Knative. https://knative.dev/docs/. Accessed: 2025-01-09.

[20] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 149–154. https://doi.org/10.1145/3317550.3321447 https://dl.acm.org/doi/10.1145/3317550.3321447.

[21] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, Portland Oregon, 450–462. https://doi.org/10.1145/2749469.2749475 https://dl.acm.org/doi/10.1145/2749469.2749475.

[22] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. 2017. Improving Spark Application Throughput via Memory Aware Task Co-Location: A Mixture of Experts Approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 95–108. https://doi.org/10.1145/3135974.3135984 https://doi.org/10.1145/3135974.3135984.

[23] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Porto Alegre Brazil, 248–259. https://doi.org/10.1145/2155620.2155650 https://dl.acm.org/doi/10.1145/2155620.2155650.

[24] Jason Mars, Lingjia Tang, and Mary Lou Soffa. 2011. Directly Characterizing Cross Core Interference through Contention Synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, Heraklion Greece, 167–176. https://doi.org/10.1145/1944862.1944887 https://dl.acm.org/doi/10.1145/1944862.1944887.

[25] John D McCalpin. 1995. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what* 22, 7 (1995).

[26] Microsoft Azure. 2025. Azure Batch – Compute Job Scheduling Service. https://azure.microsoft.com/en-us/products/batch. Accessed: 2025-01-13.

[27] OpenFaaS Ltd. 2025. OpenFaaS/FaaS-Netes. https://github.com/openfaas/faas-netes. Accessed: 2025-01-09.

[28] OProfile Project. 2025. OProfile – A System Profiler for Linux (News). https://oprofile.sourceforge.io/news/. Accessed: 2025-01-10.

[29] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, Shanghai China, 184–200. https://doi.org/10.1145/3132747.3132766 https://dl.acm.org/doi/10.1145/3132747.3132766.

[30] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022.

Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3372–3384. https://doi.org/10.14778/3554821.3554829 https://doi.org/10.14778/3554821.3554829.

[31] Keval S. Ramani, Chuan He, Yueh-Lin Tsai, Chinedum E. Okwudire, and Ehsan Malekipour. 2022. SmartScan: An Intelligent Scanning Approach for Uniform Thermal Distribution, Reduced Residual Stresses and Deformations in PBF Additive Manufacturing. *Additive Manufacturing* 52 (April 2022), 102643. https://doi.org/10.1016/j.addma.2022.102643 https://www.sciencedirect.com/science/article/pii/S2214860422000495.

[32] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. 2025. Leave Nothing Idle: Filling Datacenter Resource Utilization Gaps with Quicksand. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)* (Philadelphia, PA, USA). USENIX Association, Philadelphia, PA, USA.

[33] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving {Microsecond-Scale} Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan.

[34] SchedMD LLC. 2025. Slurm Workload Manager – Overview. https://slurm.schedmd.com/overview.html. Accessed: 2025-01-13.

[35] Carlos Segarra, Simon Shillaker, Guo Li, Eleftheria Mappoura, Rodrigo Bruno, Lluís Vilanova, and Peter Pietzuch. 2025. GRANNY: Granular Management of Compute-Intensive Applications in the Cloud. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)* (Philadelphia, PA, USA). USENIX Association, Philadelphia, PA, USA.

[36] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. (2020).

[37] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. 2022. Profile-Driven Memory Bandwidth Management for Accelerators and CPUs in QoS-enabled Platforms. *Real-Time Systems* 58, 3 (Sept. 2022), 235–274. https://doi.org/10.1007/s11241-022-09382-x https://doi.org/10.1007/s11241-022-09382-x.

[38] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. 2024. Unifying Serverless and Microservice Workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 385–402. https://doi.org/10.1145/3694715.3695947 https://dl.acm.org/doi/10.1145/3694715.3695947.

[39] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. (2021).

[40] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2016. Predicting the Memory Bandwidth and Optimal Core Allocations for Multi-Threaded Applications on Large-Scale NUMA Machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 419–431. https://doi.org/10.1109/HPCA.2016.7446083 https://ieeexplore.ieee.org/document/7446083/?arnumber=7446083.

[41] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the Planet of Serverless Computing: A Systematic Review. *ACM Trans. Softw. Eng. Methodol.* 32, 5 (July 2023), 131:1–131:61. https://doi.org/10.1145/3579643 https://doi.org/10.1145/3579643.

[42] Alan Wright. 2024. Luceneplusplus/LucenePlusPlus. https://github.com/luceneplusplus/LucenePlusPlus.

[43] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 170–184. https://doi.org/10.1145/2749469.2750401 https://dl.acm.org/doi/10.1145/2749469.2750401.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. (2010).

[45] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ACM, Vancouver BC Canada, 1–14. https://doi.org/10.1145/3567955.3567960 https://dl.acm.org/doi/10.1145/3567955.3567960.