

THE UNIVERSITY OF CHICAGO

SCHEDULING WITHOUT FORESIGHT: REACTIVE, OBSERVATION-BASED
RUNTIME SCHEDULING SYSTEMS FOR UNPREDICTABLE WORKLOADS

A DISSERTATION PROPOSAL SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
JIANRU DING

CHICAGO, ILLINOIS

AUGUST 2026

TABLE OF CONTENTS

LIST OF FIGURES	vi
LIST OF TABLES	viii
ABSTRACT	ix
ACKNOWLEDGMENTS	xi
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Scheduling hybrid in-memory and streaming workloads	3
1.3 Power management for overprovisioned systems	4
1.4 Serving multi-turn agentic LLM workloads	5
1.5 Thesis statement and contributions	6
2 ROBIN-HOOD: SCALABLE RUNTIME SCHEDULING FOR STREAMING AND IN-MEMORY TASKS	9
2.1 Introduction	9
2.2 Related Work and Motivation	11
2.2.1 Load Balancing in In-Memory Systems	12
2.2.2 Load Balancing in Streaming Systems	13
2.2.3 Comparison with Prior Systems	15
2.3 Load-Balancing with Robin-Hood	17
2.3.1 Design Principles	17
2.3.2 Runtime Architecture Overview	18
2.3.3 Unified-Work and Unified-Work-Queue Abstractions	20
2.3.4 Unified-Work-Queue Internals	21
2.3.5 Load-Balancing Strategy	22
2.3.6 Runtime Threads	23
2.3.7 Optimizations for Fine-Grained Streaming	24
2.4 Robin-Hood Implementation	25
2.4.1 Implementation Platform: UpDown System	25
2.4.2 Runtime Thread Mapping	26
2.4.3 Unified-Work-Queue Implementation	28
2.4.4 Static Analysis for Fine-Grained Tasks	29
2.5 Robin-Hood Evaluation	29
2.5.1 Experimental Setup	30
2.5.2 Q1: Performance and Load Balance	32
2.5.3 Q2: Scalability	33
2.5.4 Q3: Design Contributions and Overhead	33
2.5.5 Summary	35

3	DPS: ADAPTIVE POWER MANAGEMENT FOR OVERPROVISIONED SYSTEMS	43
3.1	Introduction	43
3.2	Related work	48
3.2.1	Constant Allocation Systems	49
3.2.2	Model-based Systems	50
3.2.3	Stateless Model-free Systems	51
3.3	Power Management with Dynamics	52
3.3.1	Fluctuating Power Demands	53
3.3.2	Power Management Challenges	55
3.3.3	Power Dynamics	56
3.4	DPS Design	56
3.4.1	Design Principles	57
3.4.2	Hardware Support: RAPL in DPS	58
3.4.3	DPS Control System	58
3.4.4	Power Dynamics Address Challenges	65
3.5	Experimental Setup	66
3.5.1	Hardware Evaluation Platform	66
3.5.2	Data Analytics and HPC Benchmarks	66
3.6	Results and Evaluations	67
3.6.1	Performance: Spark Low Utility Workloads	68
3.6.2	Performance: Spark High Utility Workloads	69
3.6.3	Performance: Spark & NPB Workloads	70
3.6.4	Fairness Analysis	72
3.6.5	Overhead Analysis	72
3.6.6	Summary of Key Results	74
3.7	Conclusion	74
4	OBSERVATION, NOT PREDICTION: CONVERSATION-LEVEL DISAGGREGATED SCHEDULING FOR AGENTIC SERVING	77
4.1	Introduction	77
4.2	Background	81
4.2.1	LLM Serving Techniques	81
4.2.2	Multi-turn Agentic Conversations	82
4.2.3	Serving with Heterogeneous GPUs	83
4.3	Characterization of Agentic Workloads	84
4.3.1	Prefill Phase	85
4.3.2	Decode Phase	87
4.3.3	Heterogeneous Serving	91
4.4	ConServe	92
4.4.1	Instance Configuration	95
4.4.2	Reactive Scheduling	95
4.4.3	Heterogeneous GPU Mapping	96
4.5	Evaluation	97

4.5.1	Experiment Setup	97
4.5.2	Q1: Agentic Conversation Performance	99
4.5.3	Q2: Serving Throughput and Quality	101
4.5.4	Q3: Pitfalls of Per-Turn Routing	102
4.5.5	Q4: Serving with Heterogeneous GPUs	103
4.6	Related Work	104
4.7	Conclusion	106
5	FUTURE WORK AND CONCLUSIONS	111
5.1	Future Work	111
5.2	Conclusion	112
	REFERENCES	114

LIST OF FIGURES

2.1	Robin-Hood Design. In-memory tasks are pre-assigned into the In-memory Queue and streaming tasks are <i>inserted</i> into the Streaming Hashmap by CEs. Load balancing is operated at two levels: Intra-Neighborhood: CEs <i>claim</i> tasks from local Unified-Work-Queue to execute; Inter-Neighborhood: work stealing between Unified-Work-Queues by <i>claiming</i>	17
2.2	Unified-Work-Queue state machine	36
2.3	Entry design of Streaming hashmap.	36
2.4	Geo-mean performance speedup of Robin-Hood and Work stealing normalized over Non-load-balancing. The error bars represent the minimum and maximum speedups. Robin-Hood includes results of whether NLBStream is on or off and the dot marker indicates which choice is made by the static analysis.	40
2.5	Performance speedup of Robin-Hood and Work stealing normalized over Non-load-balancing.	41
2.6	The lines show log latency for Non-load-balancing. (a) We extrapolate Freescale by 64 times so that it is large enough to have meaningful load to balance at 2^{17} parallelism and evaluate CSR SPMV. (b) We extrapolate live-journal by 32 times and evaluate TC.	42
3.1	Motivational Example: power management systems	46
3.2	Power phases for different applications.	54
3.3	DPS control system design	58
3.4	Hmean performance gain of the Spark low utility group normalized to under constant 110W cap.	68
3.5	Normalized hmean performance gain of the Spark high utility group.	69
3.6	Normalized hmean application performance gain of the Spark NPB group.	70
3.7	Fairness of the two High-utility workload groups.	73
4.1	Input/Output token distribution of the first 10 turns from agentic traces. For the appended input, turn-1 is the input prompt, turn-2+ is the tool call response.	84
4.2	Characterization of TTFT under large inputs with and without caching. Uncached agentic prompts ($\geq 10^4$) incur quadratic latency overheads but remain predictable as shown in the label ($R^2 = 1.0$).	86
4.3	KV-transfer overhead is constant at short input lengths (< 1024) and scales linearly at long inputs (≥ 1024). KV-transfer overhead between GPUs is significant when input is short but is dominated by Prefill latency in agentic workloads where inputs are tens of thousands of tokens long.	88
4.4	Heat map of mean time between tokens (TBT) across a range of batch sizes and context lengths. The dashed white line highlights the boundary between unsaturated (lower left) and saturated (upper right) memory bandwidth regions. Given the long context length of agentic tasks, agentic workloads lay in the saturated memory bandwidth domain.	89

4.5	Iteration latency of colocated prefill and decode stages. (a) Latency of a prefill request arriving after 58 – 92 decode iterations across a range decoder batch sizes. (b) Latency of a prefill request arriving after a range of decoding iterations with a fixed batch size (8). Prefix-caching significantly improves collocation overhead.	90
4.6	High variance in TBT throughout a long decode	91
4.7	TTFT difference between uncapped GPUs and GPUS capped at 2/3 TDP. Power capping impacts the prefill stage heavily but has a marginal effect on prefix caching.	92
4.8	Heat map of mean TBT difference between uncapped GPUs and GPUS capped at 2/3 TDP across a range of batch sizes and context lengths. The dashed black line highlights the boundary between unsaturated (lower left) and saturated (upper right) memory bandwidth regions. Power capping has a marginal effect in the saturated memory bandwidth domain.	93
4.9	ConServe System Architecture.	94
4.10	Normalized agentic performance results over multiple request arrival rates (lower is better). AMPD has 10% wrong prediction rate.	107
4.11	Overall TTFT and TBT distribution at RPS 1.634.	108
4.12	Comparison between ConServe and AMPD over different wrong prediction rate.	109
4.13	P95 agentic performance and energy efficiency on heterogeneous GPUs.	110

LIST OF TABLES

2.1	Comparison of prior systems by load-balancing structure and support for in-memory and streaming workloads.	15
2.2	Roles of Robin-Hood runtime threads.	23
2.3	Benchmark applications	38
2.4	Benchmark baseline performance	38
2.5	Sparse matrix datasets	39
2.6	Graph datasets for PR and TC	39
2.7	Graph datasets for Jaccard Similarity and GCN	39
3.1	Terminology.	55
3.2	Spark benchmark workloads.	76
3.3	Spark benchmark computing Resources.	76
3.4	NAS Parallel Benchmark applications.	76

ABSTRACT

Modern computing systems increasingly face workloads whose resource demands cannot be known in advance. Workloads themselves are becoming more heterogeneous and irregular, the pipelines that compose them mix fundamentally different execution models, and growing power constraints force schedulers to operate within tight resource envelopes rather than simply provisioning for peak demand. The emergence of generative AI services such as LLM inference compounds this further, as the compute cost is inherently unpredictable. Existing systems either assume that future demands can be predicted from past behavior—an assumption that ranges from brittle to fundamentally impossible depending on the workload—or discard readily observable information and forgo adaptation entirely.

This thesis argues for an alternative to prediction-based scheduling: rather than modeling future demand from historical data, each system reacts to a carefully chosen signal observed at runtime, which provides sufficient structure to schedule effectively even when full prediction is difficult or impractical. We demonstrate this across three systems.

Robin-Hood addresses load balancing for hybrid workloads that combine in-memory and streaming tasks. By encapsulating both task types into a shared abstraction, Robin-Hood decouples scheduling from data source, enabling a single two-level hierarchical scheduler that combines local atomic claiming with locality-aware work stealing across both modalities.

In overprovisioned distributed systems, a cluster-wide power budget must be divided among nodes whose demands fluctuate unpredictably. DPS manages this without application profiling or learned models. Instead, it tracks power dynamics—the first derivative and frequency of recent power changes—to anticipate near-future demand and distribute the budget fairly, while guaranteeing no worse performance than a static equal allocation.

For multi-turn LLM agentic workloads, total compute is unknowable at arrival, yet most of it is decode that should remain on a single replica to reuse the KV cache. ConServe turns this apparent obstacle into a scheduling advantage by disaggregating only the first-turn

prefill, the one expensive and predictable piece, from everything else. This decomposition renders the scheduling problem fully observable: prefiller utilization determines queueing time and decoder utilization determines end-to-end latency, both known when a request arrives. Where prior serving systems formulate scheduling as a constrained optimization that requires predicting conversation length and output size, ConServe chooses a decomposition that eliminates the need for such prediction. The resulting architecture maps naturally onto heterogeneous GPU clusters, assigning first-turn prefill to high-power GPUs and the rest to lower-power GPUs with comparable memory bandwidth, achieving improved performance with better power scaling.

Robin-Hood achieves up to $3.22\times$ speedup over static scheduling, DPS outperforms SLURM’s power manager by a mean 8%, and ConServe reduces job completion latency by up to 25% at saturation. Together, these results demonstrate that across scheduling, power management, and inference serving, the right abstraction often matters more than the right prediction. In each domain, a single well-chosen runtime signal recovers much of the performance gap between naive baselines and oracle systems, suggesting that this lightweight, signal-driven approach offers a broadly applicable design principle for systems facing unpredictable demands.

ACKNOWLEDGMENTS

Gratefulness to my parents, who have been supporting me the whole way. There were a lot of difficult times, so difficult that I wouldn't have been able to finish this 22nd grade and write this down if it weren't for my parents. Also huge regards to my best friend and family, Tongyu, who has been my motivation and support since I started.

Next I have to thank Christopher Stewart. Chris was my research advisor during undergrad, and he enlightened me when I was at my most confusing times. He offered me remarkable opportunities and brought me into the world of system research. And among anyone, I enjoyed playing basketball with him most.

Hank Hoffmann is my PhD advisor. My gratitude is beyond words. He is more than a wise, patient, humorous, honest, upright, principled mentor to me. I learned how to be a better person by looking at how he behaves. I hope I can learn how to be a better boxer by looking at how he fights.

Lastly, I have the honor of getting to know several of the most brilliant people. Yuan, with whom I share life. Zijie and Hanchen, my go-to friends for any fun time. Mingyuan, Miguel, Ziyi, the "four heavenly kings". My group SEEC, Pouya, Ryien, Trenton, Anwasha, and Tejas. Ahsan, my best and forever friend from the group. Yi, a good friend and leader on my academic pathway. Tianchi, Ruiqi, and Jiya, the little squad with whom we drilled on the most difficult problems together.

Just as Hank said in his thesis, I could never have done this by myself. It was only through the support of others that I could make it to this point. But I'm sure I will have more to offer than listing names in this document.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Scheduling assigns limited resources to competing work overtime [Pinedo, 2022]. How well a scheduler can do is bounded by what it knows about that work, primarily the processing time, release time, and dependencies of each job. Classical scheduling theory makes this dependence precise by distinguishing *clairvoyant* scheduling, in which these job characteristics are known in advance, from *non-clairvoyant* scheduling, in which a job’s cost is revealed only as it executes [Motwani et al., 1993]. The separation is not cosmetic: no deterministic non-clairvoyant algorithm matches the clairvoyant optimum in general, and the achievable competitive ratio is bounded away from it, which makes precise the intuition that information about the workload is what enables good scheduling. Yet real systems rarely supply this information. The clairvoyant assumption already fails in time-sharing operating systems, where the scheduler serves processes whose future behavior it cannot observe [Motwani et al., 1993], and many workloads now common in practice push the gap wider still. Confronted with work it cannot fully characterize before acting, a scheduler can take one of two stances: estimate the missing information and schedule against that estimate, or schedule on only what it can observe at runtime.

One stance is to predict the missing characteristics. A scheduler given learned estimates of its jobs’ parameters can approach clairvoyant performance when those estimates are accurate, and remains competitive when they are not [Kumar et al., 2018, Lattanzi et al., 2020]. The same idea recurs wherever the governing quantity is hidden at decision time. Model-based power managers learn how a node’s behavior maps to its power-performance tradeoff and predict the allocation that maximizes performance under a budget [Zhang and Hoffmann, 2019, Whiteside et al., 2017]. LLM serving systems route and reorder requests by predicted

output length, the quantity that fixes how long a request will hold a replica [Fu et al., 2024, Jain et al., 2025]. In each case the decision rests on an estimate of something the work has not yet produced: the job’s runtime, the node’s coming power draw, or the request’s output length. This is a coherent and often effective design, but it rests on one structural dependence: the schedule is only as good as the estimate [Mitzenmacher and Vassilvitskii, 2020]. A model fit to one workload or one generation of hardware can mispredict once either drifts [Wang et al., 2022b], and the quantity it predicts is, by construction, unmeasured at the moment of decision. The schedule degrades with whatever error remains.

The other stance avoids the estimate entirely. Rather than guess an undetermined quantity, a reactive scheduler conditions on quantities already determined when it acts, and so pays none of prediction’s price: no model to train, no accuracy to maintain, nothing to mispredict when the workload shifts. Classical non-clairvoyant scheduling shows this can be enough. Round-robin serves the unfinished jobs in equal measure, never estimating how long any will run, yet for total completion time it stays within a factor of two of the clairvoyant optimum, the best any deterministic algorithm achieves [Motwani et al., 1993]. Work-stealing carries the idea into load balancing, where a worker that empties its queue reacts by stealing from another, conditioning on observed queue occupancy rather than a model of how work will spread [Blumofe et al., 1995a]; its foil, static partitioning, fixes the assignment in advance and cannot adjust when task costs prove data-dependent [Zaharia et al., 2010]. Power management splits the same way, with stateless managers capping each node from its instantaneous draw alone [Yoo et al., 2003].

Reactive scheduling has a limitation of its own. Where prediction turns on the accuracy of its estimate, the reactive approach turns on the richness of its signal. A scheduler that observes the present instant but not the trend cannot act on a demand the recent past already implies. The two stances thus fail in opposite directions, one by trusting an estimate it cannot verify, the other by discarding structure it could have observed. This dissertation

stands between them, staying on the reactive side while treating the choice of signal as the central design problem rather than an afterthought. We develop this position across three systems, each identifying what that signal is, showing why it suffices, and constructing a runtime system that reacts to it.

1.2 Scheduling hybrid in-memory and streaming workloads

In-memory and streaming systems make incompatible assumptions about the work they schedule. In-memory frameworks [Zaharia et al., 2010] operate on bounded, fully materialized data and represent load balancing as the partitioning and migration of tasks whose data footprints are known in advance. This presumes that all work is countable and relocatable, a form of foreknowledge of the workload that admits static task graphs and partition-based assignment [Dean and Ghemawat, 2008, Zaharia et al., 2010, Aslam and Shah, 2015]. Streaming systems [Margara et al., 2023] make the opposite assumption: data arrives continuously over the network, the total volume of work cannot be enumerated, and tasks must be routed by key as they appear [Carbone et al., 2015, Narkhede et al., 2017, Liu and Buyya, 2020a]. Applications that mix both modalities, increasingly common in data lakes, hybrid transactional-analytical engines, and in-situ scientific analytics, fall into the gap [Schmitz et al., 2025, Yang et al., 2025, Makreshanski et al., 2017, Do et al., 2022]. Techniques that assume enumerable work cannot accommodate unbounded streams, and techniques built for streams provide no mechanism for redistributing large memory-resident structures.

Robin-Hood resolves this by making the scheduling decision react to observed local state rather than to an enumerated global plan. The system encapsulates both task types in a single abstraction, Unified-Work, that decouples scheduling from the data source, and stores them in a shared queue that a compute element drains by atomic claiming. When a neighborhood exhausts its local queue, it reacts by stealing work from remote neighborhoods using a locality-aware policy. The signal driving every decision is local queue occupancy,

observable without coordination, which is precisely the quantity that survives the transition from bounded to unbounded work. The unified abstraction lets a single reactive substrate balance both modalities at the scale of millions of compute elements.

1.3 Power management for overprovisioned systems

An overprovisioned cluster can draw more power than its budget permits and must therefore cap individual nodes, dividing a fixed cluster-wide budget among nodes whose demands fluctuate across application phases [Patki et al., 2013]. Prior managers fall cleanly onto the prediction-reaction axis [Meisner et al., 2011, Whiteside et al., 2017, Zhang and Hoffmann, 2019]. Model-based systems learn the relationship between system behavior and the power-performance tradeoff, then predict performance-optimal allocations. They achieve near-optimal performance when their training data covers the operating regime, but they incur substantial deployment cost on each new architecture and lose their guarantees, occasionally including the guarantee of respecting the budget, when the workload leaves the training distribution. Stateless managers, exemplified by the widely deployed SLURM power plugin [Yoo et al., 2003], occupy the opposite pole: they reallocate budget using only instantaneous power. This makes them robust and immediately deployable, but blind to trend. A stateless manager cannot tell a node that is ramping up from one that has plateaued, and as a result it can be driven into allocations that perform worse than a static equal split.

DPS shows that the useful middle ground is reactive but not memoryless. The system retains a small amount of state, the recent power dynamics of each node, captured as the first derivative and frequency of its power changes. From this signal it anticipates near-future demand and prioritizes nodes that are rising or oscillating quickly, without constructing any model of the application. Because power changes with inertia, the recent trajectory is locally predictive of the near future, and tracking it is enough to recover much of the benefit a model would provide. Retaining state also lets DPS detect when power is fluctuating too quickly

to manage actively and fall back to a static allocation, guaranteeing it never performs worse than an equal split. The signal here is recent power behavior; the reaction is a priority that modifies an otherwise stateless allocation.

1.4 Serving multi-turn agentic LLM workloads

A multi-turn agent [Yao et al., 2023] resolves a user task through many turns of dependent inference interleaved with tool calls, so the total compute a conversation will demand is unknown when it arrives: both its output length and its remaining turn count are decided only as it runs. Existing serving systems keep the turn as the unit of scheduling, deciding turn by turn whether to disaggregate prefill from decode [He et al., 2026, Li et al., 2026]. That decision depends on the turn’s decode length, tool behavior, and KV-cache growth, none of which is observable when the scheduler must act, so the system is forced to estimate them, and any estimate eventually misjudges a turn and routes it wrongly. The dependence on prediction here is not a property of the workload but of the scheduling unit: splitting a conversation into independently scheduled turns discards the very structure that would have made placement deterministic.

ConServe removes that dependence by raising the scheduling unit from the turn to the conversation. At turn granularity, decode lengths, tool calls, and KV growth are irregular and hard to infer; at conversation granularity, the irregularity collapses into a stable two-phase structure: a compute-bound first-turn prefill that encodes the task, followed by a long, memory-bound tail in which each later turn appends only a short tool response and decodes over an accumulating KV cache. This recovers the familiar prefill-decode structure [Patel et al., 2024], now at the scale of a whole conversation. Once the conversation is the unit, placement reduces to two quantities that are both observable: the first-turn input length, which fixes prefill cost through a latency curve profiled offline, and the per-decoder KV-cache occupancy. Neither is a forecast of what the workload will do; both are measurements of

state the system already holds.

The system that follows is deliberately minimal. ConServe routes the first-turn prefill to a high-throughput prefiller, transfers the resulting KV cache exactly once, and pins the conversation to a single decoder for its entire tail, where every subsequent turn runs with full cache reuse. It makes one placement decision per conversation and predicts no decode-side quantity at any point. Decoder replicas are overprovisioned so the prefiller saturates first, which places the throughput ceiling on the one tier whose load is governed by an observable, deterministic quantity, the incoming input-token rate; admission and scale-out then follow from state the scheduler already reads. The same two-phase split maps onto heterogeneous hardware by construction: the compute-bound prefill takes the highest-power GPU available, while the memory-bound tail, gated by memory bandwidth rather than compute, runs on lower-power GPUs of comparable bandwidth, so energy efficiency follows as a consequence. Where prior systems estimate quantities the workload has not yet produced and pay for the misjudgments, ConServe chooses a unit of scheduling under which the signal it needs is already determined when it acts.

1.5 Thesis statement and contributions

Taken together, the three systems support a single claim: across load balancing, power management, and agentic LLM inference serving, a scheduler should base decisions on observed runtime state and react to demand as it arrives, so that its performance does not depend on forecasting future load, the one quantity that unpredictable workloads make unpredictable. The signal differs by domain, local queue occupancy, recent power dynamics, and first-turn input length, but the design pattern is the same. Each system identifies the quantity that remains observable when prediction breaks down, chooses a scheduling unit or abstraction that makes that quantity sufficient, and reacts to it.

This thesis makes the following contributions:

- **A framing of scheduling under unpredictability.** We cast scheduling for unpredictable workloads as a choice between prediction-based and reactive policies, identify the choice of observable signal as the central design question on the reactive side, and show across three domains that a signal observable at decision time can carry enough structure to schedule well without forecasting the undetermined. The three domains span the range from prediction being impractical to fundamentally impossible.
- **Robin-Hood: unified load balancing for hybrid workloads.** We introduce Unified-Work and the Unified-Work-Queue, a task abstraction and shared queue that decouple scheduling from data source, together with a two-level scheduler that combines local atomic claiming with locality-aware work stealing. Conditioning only on observed queue occupancy, Robin-Hood balances in-memory and streaming tasks within one runtime and, on a cycle-accurate simulator scaling to one million compute elements, achieves a geometric-mean $3.22\times$ speedup over static scheduling and $1.54\times$ over work stealing.
- **DPS: model-free power management from power dynamics.** We introduce model-free but stateful power manager that conditions on recent power dynamics, the first derivative and frequency of each node’s power, rather than a learned model. We show this signal anticipates demand well enough to redistribute a cluster-wide budget while guaranteeing no worse performance than constant allocation, and on a public cloud running Spark and HPC workloads, DPS outperforms the SLURM power plugin by a mean of 8% and up to 21.3%.
- **ConServe: conversation-level scheduling for agentic serving.** We raise the scheduling unit from the turn to the conversation, collapsing turn-level irregularity into a two-phase structure and replacing per-turn prediction with two observable signals: first-turn input length and per-decoder KV occupancy. We introduce time-to-

first-effective-token and last-turn time-between-tokens as the conversation-level metrics agentic serving actually targets, show that per-turn prediction degrades linearly with its error rate while conversation-level placement holds constant, and map the two-phase split onto heterogeneous GPUs. ConServe reduces p95 TTFET by 51% over per-turn routing and improves energy efficiency, with a further 22.75% on heterogeneous hardware.

The remainder of this dissertation develops each system in turn. Chapter 2 presents Robin-Hood and its unified abstraction for hybrid in-memory and streaming load balancing. Chapter 3 presents DPS and the use of power dynamics for model-free power management. Chapter 4 presents ConServe and conversation-level scheduling for agentic LLM serving. Chapter 5 concludes the reactive, observation-based design principle and discusses future work.

CHAPTER 2

ROBIN-HOOD: SCALABLE RUNTIME SCHEDULING FOR STREAMING AND IN-MEMORY TASKS

2.1 Introduction

Modern large-scale systems often execute irregular workloads that require careful *load balancing* to achieve scalability and high performance [Jiang, 2016b]. Two major execution models dominate: *in-memory systems*, which operate on bounded datasets resident in memory or disk (e.g., Spark [Zaharia et al., 2010]); and *streaming workloads*, which process unbounded data as it arrives over the network [Margara et al., 2023]. Each model adopts distinct scheduling strategies and runtime abstractions, optimized for its data flow patterns. As a result, load-balancing mechanisms built for one are typically incompatible with the other.

However, many modern applications increasingly require *hybrid workloads* that combine both in-memory and streaming tasks within a single execution pipeline. Examples arise in data lakes that mix batch and streaming ingestion [Schmitz et al., 2025], real-time analytics systems that support both continuous queries and stored relational processing [Yang et al., 2025], hybrid transactional-analytical engines that integrate real-time updates with analytical queries [Makreshanski et al., 2017], and scientific simulations that stream data to in situ analytics [Do et al., 2022, Levy et al., 2016].

Unfortunately, in-memory and streaming systems make fundamentally different assumptions about data availability and task scheduling. In-memory systems operate on bounded data that is fully materialized (i.e., memory-resident), enabling optimizations like partition-based load balancing and static task graphs [Dean and Ghemawat, 2008, Zaharia et al., 2010, Aslam and Shah, 2015]. Streaming systems, on the contrary, operate continuously to process dynamic data that arrives over the network, using event-driven schedulers and key-based routing [Carbone et al., 2015, Narkhede et al., 2017, Liu and Buyya, 2020a].

These models are difficult to reconcile: techniques like global barriers, task reformatting, or runtime coordination are often required to bridge the two. As a result, applications with hybrid workloads face significant challenges in integrating both paradigms without incurring performance bottlenecks, redundant logic, or increased system complexity.

This paper introduces **Robin-Hood**, a runtime system that provides *unified load balancing for hybrid workloads* composed of both in-memory and streaming tasks. Robin-Hood introduces two key abstractions: **Unified-Work**, a unified representation of a task and its associated data, and **Unified-Work-Queue**, a local task queue that stores both streaming and in-memory tasks. These abstractions decouple task scheduling from data source assumptions, allowing the runtime to balance heterogeneous workloads using a shared scheduling interface. Internally, the Unified-Work-Queue optimizes data access for each modality: in-memory tasks are stored as pre-materialized objects, while streaming tasks are represented as a data source that will forward data to its eventual consumer, but the consumer is not bound until the task is ready for execution. This design eliminates the need for runtime coordination between disjoint execution engines or global barriers to transition between operating on different data types.

To achieve scalability, Robin-Hood adopts a two-level load-balancing scheme that groups compute elements (CEs) into *Neighborhoods*—sets of nodes that share fast local memory. Each Neighborhood maintains a private Unified-Work-Queue and services tasks locally when possible, preserving data locality and reducing synchronization overhead. When a Neighborhood runs out of work, its CEs perform *locality-aware, round-robin work-stealing* across other Neighborhoods to acquire new tasks. This structure enables efficient, decentralized load balancing that scales to systems with millions of CEs, while maintaining responsiveness to fine-grained streaming tasks.

We implement Robin-Hood by extending a MapReduce-style framework [Wang et al., 2023] to support streaming execution in the Reduce stage, using our unified scheduling ab-

straction. To evaluate its scalability, we deploy Robin-Hood on a cycle-accurate simulator of the UpDown architecture [Rajasukumar et al., 2024a], a distributed computing system capable of scaling to millions of compute elements. Across a suite of hybrid analytics workloads, including sparse matrix operations, graph algorithms, and sorting, we observe up to $3.22\times$ speedup over static scheduling and a $1.54\times$ improvement over classic work-stealing. We also show that Robin-Hood produces performance wins even when scaling to over 1 million CEs. These results demonstrate that Robin-Hood achieves both scalability and robust performance across irregular, hybrid workloads.

This paper makes the following contributions:

- We propose **Robin-Hood**, to our knowledge the first runtime that unifies load balancing across both in-memory and streaming tasks, enabling efficient execution of hybrid workloads without isolating phases or converting tasks.
- We introduce two key abstractions: **Unified-Work**, which encapsulates both task and data in a uniform interface, and **Unified-Work-Queue**, a shared queue that supports scheduling across data modalities.

Together, these contributions provide a scalable, unified runtime that eliminates the need for bridging logic between disjoint streaming and in-memory systems—simplifying hybrid application development while preserving performance.

2.2 Related Work and Motivation

Hybrid workloads that combine in-memory and streaming tasks are becoming increasingly common in both scientific and data analytics applications [Do et al., 2022, Levy et al., 2016, Schmitz et al., 2025, Yang et al., 2025, Makreshanski et al., 2017]. However, existing runtime systems treat these paradigms separately, providing either in-memory or streaming support but rarely both. This section reviews load balancing techniques in each model and shows

that while each has been extensively studied, prior work fails to provide a unified abstraction or scheduler structure capable of handling both simultaneously.

2.2.1 Load Balancing in In-Memory Systems

Data and Task Representation. In-memory systems operate on bounded datasets that are fully materialized in advance, such as those in Spark [Zaharia et al., 2010] or Hadoop [Dean and Ghemawat, 2008]. Here, tasks are typically represented as a function and a data reference, and load balance is achieved by partitioning the data and migrating it as computation progresses [Dhakal et al., 2007, Ghodsi et al., 2011, Frigo et al., 1998]. Even highly dynamic HPC frameworks, which support dynamic task graphs and asynchronous execution, such as Charm++ [Kale and Krishnan, 1993b], HPX [Kaiser et al., 2020], and Legion [Chapin et al., 1999], retain this basic assumption, although they add support for fine-grained parallelism, locality-aware scheduling, and some dynamic task migration. These models work well for fully materialized data but do not easily adapt to dynamically arriving workloads without costly global coordination or task conversion.

Scheduler structure. In-memory systems have explored a wide spectrum of load-balancing strategies. Early work focused on *static schemes* that schedule tasks based on prior knowledge of the workload or hardware configuration, such as the number of CEs or available memory capacity [Chen et al., 2013, Li et al., 2014, Etminani et al., 2009, Thiyeb and Alhomdy, 2020, Jiang, 2016a]. These methods are fast but inflexible, and cannot adapt to changing loads at runtime. Dynamic methods instead assign tasks according to the current system state, using information such as CE utilization or the distribution of unexecuted tasks [Ivanisenko and Radivilova, 2015, Elsakaan and Amroun, 2024a]. Fully decentralized approaches based on work-stealing further improve scalability by avoiding a single point of coordination [He et al., 2020, Freitas et al., 2021, Gutierrez-Garcia et al., 2023, Lifflander et al., 2012], while hierarchical schemes combine groups of CEs to balance load locally and

then exchange work globally [Elsakaan and Amroun, 2024b, Fu et al., 2023, Sakellariou and Zhao, 2004, Kumar et al., 2019, Kruekaew and Kimpan, 2022]. Although these mechanisms work well for bounded, pre-materialized data, they assume that all work is countable and relocatable, making them difficult to apply directly to streaming or dynamically generated workloads.

2.2.2 Load Balancing in Streaming Systems

Data and Task Representation. Streaming systems process unbounded, real-time data that arrives continuously over the network, making the total volume of work inherently unpredictable and difficult to enumerate. Unlike in-memory models, where data resides in known memory locations, streaming data is transient and must be processed on the fly. To enable parallelism, incoming data is typically segmented into *shards* or *windows*, which are tagged with *keys* to express data affinity or operator semantics [Röger and Mayer, 2019]. These keys are then grouped with user-defined functions to form executable tasks. Load balancing in this context involves reassigning keys (and thus their associated shards) to different compute elements to maintain even processing rates [Fragkoulis et al., 2023]. This representation makes fine-grained rebalancing more difficult, as both the volume and timing of data arrivals are unpredictable.

Scheduler Structure. Streaming runtimes employ a variety of centralized and decentralized load-balancing strategies. Centralized schedulers, like those used in Apache Storm or Spark Streaming, may assign tasks in a round-robin fashion or use pool-based fairness policies. These approaches are simple but struggle to scale due to the overhead of tracking fine-grained, real-time data flow. To address this, many systems adopt decentralized or heuristic methods for dynamic key reassignment [Aniello et al., 2013, Xu et al., 2014, Chatzistergiou and Viglas, 2014, Rizou et al., 2011, Sun et al., 2015, Peng et al., 2015]. More sophisticated techniques frame scheduling as a constraint-satisfaction problem [Cardellini

et al., 2016, Jiang et al., 2018, Schneider and Wu, 2017, Zhang et al., 2016, Buddhika et al., 2017], attempting to balance load while preserving key locality and minimizing data movement.

Because central coordination is expensive, several systems pursue fully decentralized rebalancing. Many of these are inspired by the Vivaldi algorithm [Dabek et al., 2004b], predicting communication latency and selecting optimal partners for key migration [Cardellini et al., 2015, Pietzuch et al., 2006, Rizou et al., 2010]. Others use adaptive partner selection strategies based on load probes or traffic statistics [Repantis et al., 2006a, Chaturvedi et al., 2017, Zhou et al., 2006, Sun et al., 2023, Fuerst and Sharma, 2022]. These methods prioritize local information, enabling responsive task migration without global knowledge, but also complicating integration with in-memory runtimes that assume global task visibility and countability.

Batch-Stream Unification. Several modern dataflow systems mentioned above explicitly attempt to unify batch and stream processing. Apache Flink treats batch as a special case of streaming and exposes both through a common dataflow API [Carbone et al., 2015]. Google’s Dataflow model and the Apache Beam SDK [Akidau et al., 2015] similarly provide unified semantics for bounded (batch) and unbounded (streaming) data, using watermarks and triggers to reconcile the two. Naiad [Murray et al., 2013] introduced the timely dataflow model, supporting low-latency streaming, iterative, and batch-style workloads within a single execution engine. Spark Structured Streaming [Armbrust et al., 2018] incrementally maintains results for relational queries, allowing the same SQL/DataFrame API to run in either batch or streaming mode. Trill [Chandramouli et al., 2016] offers a temporal data model supporting both offline and online analytics within a single library.

These systems focus on unification at the level of *programming models, operators, and semantics*. They provide a consistent API to developers but rely on operator partitioning, key-based routing, or micro-batching for execution. Put simply, they force in-memory tasks

to look like streaming (or vice versa). In contrast, Robin-Hood unifies *runtime load balancing itself* across both in-memory and streaming tasks. By introducing a single task abstraction and a single scheduling substrate, Robin-Hood addresses hybrid workloads at scale, rather than unifying only the developer-facing interface.

Table 2.1: Comparison of prior systems by load-balancing structure and support for in-memory and streaming workloads.

System	Load-Balancing Structure	In-Mem.	Stream.
Cilk [Blumofe et al., 1995b]	Decentralized work-stealing	✓	✗
Apache Spark [Zaharia et al., 2010]	Centralized scheduling, static partitioning	✓	✗
Apache Flink [Carbone et al., 2015]	Key-based routing, operator parallelism	✗	✓
Charm++ [Kale and Krishnan, 1993b]	Migrating over-decomposed objects	✓	✗
HPX [Kaiser et al., 2020]	Futures, global addressing for tasks	✓	✗
Legion [Chapin et al., 1999]	Region-based hierarchical scheduling	✓	✗
Naiad [Murray et al., 2013]	Timestamped coordination, operator-local routing	✗	✓
Robin-Hood	Hybrid hierarchical, unified task queue	✓	✓

2.2.3 Comparison with Prior Systems

Table 2.1 summarizes several representative runtime systems, highlighting their scheduling structures and support for in-memory versus streaming workloads. While many in-memory frameworks (e.g., Spark, Charm++, HPX) and streaming engines (e.g., Flink, Naiad) have well-developed load balancing techniques, **each is limited to a single data modality**. None of these systems provides a unified abstraction or shared runtime capable of simultaneously balancing hybrid workloads.

In-memory frameworks such as Charm++, HPX, and Legion assume that all tasks are resident in memory and can be migrated or redistributed with complete knowledge of their data footprint. These abstractions are effective for static or over-decomposed workloads, but

break down when tasks are driven by unpredictable streaming arrivals, where inputs cannot be fully enumerated ahead of time. Conversely, streaming systems such as Flink and Naiad are designed around online dataflow: they rely on key-based routing and operator-local scheduling to distribute shards as they arrive. While this is well-suited for continuous input streams, it provides no mechanism for redistributing large, memory-resident data structures or for coordinating with tasks whose inputs are static.

As a result, hybrid workloads expose a semantic gap between these two paradigms. Techniques designed for one modality fail when applications simultaneously require both pre-partitioned, memory-resident data and dynamic, unpredictable streams. This explains why existing systems cannot be trivially extended to support hybrid workloads. We also note that it is a real limitation to existing MapReduce frameworks—the Map stage is naturally in-memory and the reduce stage is naturally streaming, but we are not aware of any large-scale implementation that takes advantage of this. In fact, to our knowledge, no prior runtime provides a unified abstraction and scalable scheduling framework that spans both domains.

Robin-Hood fills this gap by combining three key features:

- A unified task abstraction (Unified-Work) that decouples execution from data source,
- A shared scheduling queue (Unified-Work-Queue) capable of storing both in-memory and streaming tasks, and
- A scalable, hierarchical scheduler combining local coordination and global work-stealing.

To our knowledge, Robin-Hood is the first runtime that natively supports scalable, low-overhead load balancing across both memory-resident and dynamically streamed data sources.

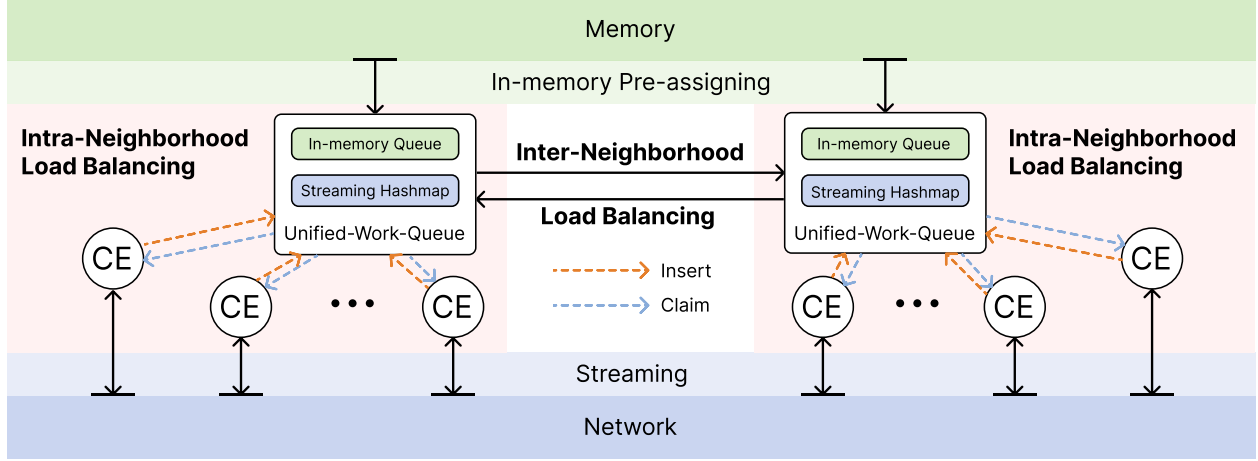


Figure 2.1: Robin-Hood Design. In-memory tasks are pre-assigned into the In-memory Queue and streaming tasks are *inserted* into the Streaming Hashmap by CEs. Load balancing is operated at two levels: Intra-Neighborhood: CEs *claim* tasks from local Unified-Work-Queue to execute; Inter-Neighborhood: work stealing between Unified-Work-Queues by *claiming*.

2.3 Load-Balancing with Robin-Hood

Robin-Hood is a runtime system that performs scalable, unified load balancing for hybrid workloads composed of both in-memory and streaming tasks. The core design insight is that while these workloads differ in data origin, they can be treated uniformly at the scheduling level through a shared abstraction. This enables a single runtime to balance both task types efficiently at extreme scale.

This section presents the Robin-Hood runtime, beginning with the guiding principles, followed by its architectural abstraction, runtime components, and the hybrid load-balancing algorithm that unifies in-memory and streaming execution.

2.3.1 Design Principles

Robin-Hood is guided by four key principles to support hybrid workloads that mix streaming and in-memory tasks, while scaling efficiently across large-scale systems. These principles are not generic maxims; they reflect specific responses to challenges uncovered in prior work

(§2.2) and the demands of fine-grained, high-throughput workloads at scale.

- **Principle 1: Abstraction-Driven Scheduling.** The scheduler must operate independently of a task’s data source, whether materialized in memory or arriving via stream. This ensures that hybrid applications can be scheduled uniformly and efficiently without duplicating logic.
- **Principle 2: Implementation-Aware Execution.** While the scheduling logic is unified, the runtime must optimize data structures for each task type. Streaming data, in particular, must avoid unnecessary DRAM materialization and use fast memory or direct forwarding.
- **Principle 3: Locality-Aware Hierarchical Balancing.** Efficient scheduling must reflect the machine’s physical structure. Load balancing should occur first within fast, *Neighborhoods*, and only escalate to cross-Neighborhood work exchange when necessary.
- **Principle 4: Fine-Grained Scalability.** To maintain high throughput even for very fine-grained tasks, the runtime must minimize overhead. Metadata, synchronization, and routing must remain lightweight and localized.

2.3.2 Runtime Architecture Overview

Figure 2.1 shows Robin-Hood’s high-level architecture. Robin-Hood organizes compute elements (CEs) into logical groups called *Neighborhoods*, which represent units of hardware locality; e.g., consider a core and a core cluster on a manycore chip, or a node and nodes in a NUMA domain, as CEs and Neighborhoods, respectively. Each Neighborhood shares a single Unified-Work-Queue.

All tasks are encapsulated as Unified-Work and each task contains a function pointer and a data reference. The scheduler interacts only with this abstraction, decoupling its logic from data placement or arrival modality (Principle 1).

Internally, the Unified-Work-Queue implementation includes two optimized structures: (1) An **In-memory Queue**, optimized for atomic pointer increments; and (2) a **Streaming Hashmap**, supporting efficient data forwarding and minimal buffering while optimized for atomic iteration (Principle 2).

Local scheduling occurs within each Neighborhood: CEs use fast atomic `claim()` operations to take tasks from the local queue. If the local queue is empty, a CE initiates *inter-Neighborhood work stealing* using a `hunt()` protocol. Stealing uses a *locality-aware round-robin* protocol: CEs remember neighborhoods with work and prioritize them. This avoids redundant probing of idle neighbors and reduces network traffic under skew (Principle 3).

Importantly, the `claim()` operation has minimal overhead, supporting ultra-fine-grained tasks. In-memory tasks only incur atomic pointer operations; streaming tasks include lightweight bookkeeping to ensure correct routing. All such overheads are bounded and localized (Principle 4).

Mapping to Design Principles:

- **P1 (Abstraction-Driven Scheduling):** Scheduling logic operates solely on Unified-Work.
- **P2 (Implementation-Aware Execution):** Specialized queues (linear and hashmap) back the unified interface.
- **P3 (Locality-Aware Balancing):** Neighborhoods exploit fast local access before remote stealing.
- **P4 (Fine-Grained Scalability):** Claiming is lightweight and skips inactive queues or stale states.

Together, this architecture supports scalable, unified execution across hybrid workloads, enabling efficient task routing and execution at the scale of millions of compute elements.

2.3.3 Unified-Work and Unified-Work-Queue Abstractions

The Unified-Work and Unified-Work-Queue abstractions are central to Robin-Hood’s ability to schedule hybrid workloads at scale. These abstractions unify the treatment of in-memory and streaming tasks, enabling all scheduling logic to operate on a common API, `insert()` and `claim()`, regardless of the data modality. Each Unified-Work instance is tagged with a key; both streaming records and memory-resident task partitions are scheduled through the same queue interface, which guarantees generality regarding task type and scalable load balancing.

Figure 2.2 presents the Unified-Work-Queue’s state machine. The four states—Dynamic, Static, Exhausted, and Terminated, govern transitions between input arrival, task completion, and application termination.

- **Initialization as Dynamic.** Both the in-memory queue and the streaming hashmap start in the Dynamic state. When *claim* is called, the Unified-Work-Queue will try to pop tasks from either data structure based on the priority between in-memory and streaming tasks. In the example of Figure 2.2, in-memory tasks have higher priority.
- **Transition to Static.** During application execution, the runtime will intercept signals indicating if new in-memory or streaming tasks stop being generated and dispatch signals to all CEs. When the Unified-Work-Queue receives such a signal, it will set the state of the corresponding data structure to Static.
- **Transition to Exhausted.** Whenever a data structure is at the Static state and depleted, its state transitions to Exhausted, and the *claim* function will skip checking this data structure to reduce overhead.
- **Transition to Termination.** If the states of both data structures are set to Exhausted, the Unified-Work-Queue will set its termination state to Unified-Work-Queue

Exhausted, and a *claim* call will do nothing but return a failure signal. The runtime will aggregate the termination state of all Unified-Work-Queue. When all are exhausted, Robin-Hood will enter its termination sequence.

Critical to scalability, the queue’s logic ensures minimal contention: insertions, claims, and state transitions are all handled atomically and locally. This unified interface is central to realizing abstraction-driven scheduling and implementation-aware execution, and enables seamless progression through both data modalities without manual intervention or system reconfiguration.

2.3.4 *Unified-Work-Queue Internals*

We implement the Unified-Work-Queue using separate internal structures specialized for in-memory and streaming tasks, providing scalable, low-overhead support for both:

- *In-Memory Queue*: A fixed-size array with head and tail pointers in fast local memory, enabling low-latency atomic claim and insert operations. The queue supports atomic advancement for concurrent claimers and marks exhaustion or termination as described in Figure 2.2.
- *Streaming Hashmap*: A lightweight, linked hashmap buffer (see Fig. 2.3) implements streaming task claims and routes records efficiently to compute elements. Entries are stored in a queue that supports atomic iteration for concurrent claimers, while the linked-list structure supports fast entry addition and deletion. Each entry holds a claim bit, destination, record count, and cache offset for data buffering, as well as a pointer for chaining. Claim is executed with the atomic bit set; if the key is unclaimed, data is buffered, otherwise it is directly routed to the destination CE that claimed the data stream.

Unified-Work-Queue states are stored as a single integer cached in local memory, with

head/tail pointers and entry fields also locally cached. The `claim()` and `insert()` methods use atomic operations on pointers or hashmap entry fields. This structure minimizes synchronization overhead, enabling scalable throughput for both statically known and dynamically arriving tasks.

2.3.5 Load-Balancing Strategy

Robin-Hood implements a two-level, hybrid load-balancing strategy to support highly parallel and irregular workloads with minimal coordination overhead, aligning with Principles P3 (locality-aware balancing) and P4 (fine-grained scalability).

Intra-Neighborhood (Local) Balancing

Within each Neighborhood, CEs share a local Unified-Work-Queue in fast memory. Worker threads claim work from this queue using atomic operations, skipping inactive queues thanks to cached state flags. This enables rapid, low-synchronization task scheduling without message passing.

Inter-Neighborhood (Global) Balancing

When a Neighborhood depletes its local queue, Worker threads initiate the `hunt()` protocol, a form of decentralized, locality-aware work stealing. Each CE selects targets using a round-robin policy, tracking previous success via the `next_dest` field. If a remote queue holds claimable work, a Receiver replies with Unified-Work (i.e., a task to execute); otherwise, the CE retries another Neighborhood. This design avoids global coordination, emphasizing local information and reducing contention.

Hybrid Load Balancing in Action

By combining centralized local balancing and decentralized global stealing, Robin-Hood achieves:

- Fast local decisions when a Neighborhood has work.
- Scalable global adaptation when load imbalance occurs.
- Seamless balancing of both streaming and in-memory tasks through unified `claim()` and `insert()` calls.

For detailed message protocol and control flow, see the Worker and Receiver thread roles in the next subsection.

2.3.6 Runtime Threads

Robin-Hood use three cooperating thread types, summarized in Table 2.2 and discussed in more detail below.

Table 2.2: Roles of Robin-Hood runtime threads.

Thread Type	Responsibilities and Interfaces
Controller	Initializes queues, spawns threads, coordinates task generation and termination. Interfaces: <code>start()</code> , <code>terminate</code> signals.
Worker	Claims and executes tasks; initiates stealing using locality-aware <code>hunt()</code> . Interfaces: <code>claim()</code> , <code>hunt()</code> .
Receiver	Handles incoming records, signals, and remote steal requests. Routes or buffers data. Interfaces: <code>insert()</code> , <code>hunt_reply()</code> .

Controller Threads

Controllers coordinate execution hierarchically. A global controller partitions input data and launches the application, node-level controllers allocate resources, and neighborhood controllers manage local Unified-Work-Queues and runtime threads. Each controller tracks

the Unified-Work-Queue state machine (Figure 2.2), marking queues as static, exhausted, or terminated as tasks complete. Termination proceeds bottom-up: neighborhood completions propagate upward until the global controller halts execution.

Worker Threads

Each compute element (CE) runs a Worker thread responsible for claiming, executing, and balancing tasks. Workers first attempt local `claim()` operations; if none are available, they issue a locality-aware `hunt()` request to remote neighborhoods. Returned tasks are executed as if local, ensuring decentralization and avoiding bottlenecks. Once all queues in a neighborhood are terminated, Workers signal completion and exit.

Receiver Threads

Receivers decouple communication from computation by handling asynchronous events. For streaming data, they check whether the key is unclaimed (buffering values) or already assigned (forwarding records directly to a CE’s cache). Receivers also process remote `hunt()` requests, returning claimable tasks, and manage control signals (`streaming_end`, `terminate`). Offloading these responsibilities enables overlap of communication and computation, maintaining responsiveness under high load.

2.3.7 Optimizations for Fine-Grained Streaming

To reduce overhead for ultra-fine-grained streaming tasks, Robin-Hood applies static analysis before execution to select one of two specialized modes (supporting Principle 4, Fine-Grained Scalability):

- **NHMStream (Non-hashmap Mode):** Used when key grouping is unnecessary. The streaming hashmap is replaced with a simple queue, lowering per-task overhead.

- **NLBStream (No Load Balancing):** Used when tasks are extremely short (fewer than ~ 40 instructions). Such tasks are executed immediately on arrival without insertion into a queue. The 40-instruction threshold corresponds to the update cost of the streaming hashmap. If a static bound cannot be determined (e.g., due to data dependence), Robin-Hood falls back to regular load balancing.

These modes ensure that Robin-Hood maintains performance even in pathological cases with very short streaming tasks, while preserving generality across diverse workloads.

2.4 Robin-Hood Implementation

We prototype Robin-Hood as a MapReduce-style framework that combines in-memory map tasks with streaming reduce tasks. While our prototype targets the UpDown architecture [Rajasukumar et al., 2024a,b], the abstractions and scheduling mechanisms are portable across other systems. The Unified-Work-Queue abstraction decouples task type from data source, and the hybrid local/global scheduling strategy applies to any architecture with hierarchical memory or decentralized compute units. Specifically, UpDown’s software stack already supports an efficient, scalable key-value store [Fell et al., 2025]. Since our Unified-Work-Queue abstraction is already designed to support keys, it maps naturally into this abstraction and it is easy to implement the Robin-Hood runtime on top of this abstraction. As a key-value store is a fairly common abstraction implemented across many different architectures, we would expect Robin-Hood’s abstractions to be easily portable.

2.4.1 Implementation Platform: UpDown System

We evaluate Robin-Hood using the UpDown system, a distributed architecture with cycle-accurate simulation infrastructure validated in prior work [Rajasukumar et al., 2024a,b, Chien et al., 2026]. UpDown enables experiments at extreme scale (up to one million compute

elements, CEs), which is infeasible on current hardware but essential for testing Robin-Hood’s scalability.

Each UpDown node consists of one CPU and 32 accelerators integrated into 8 high-bandwidth memory (HBM) stacks; each accelerator contains 64 CEs with 4MB local memory. We treat each accelerator as a Neighborhood abstract in Robin-Hood and each CE is an event-driven programmable lane within the accelerator, and Events are first-class primitives in the UpDown ISA and are grouped by threads with physical context, supporting fast context switch. Each CE supports up to 128 virtual threads. Nodes communicate through the PolarFly interconnect, a low-diameter network with microsecond-scale latency [Lakhotia et al., 2024].

2.4.2 Runtime Thread Mapping

Each UpDown accelerator is treated as a Neighborhood. The 64 CEs within an accelerator share a local Unified-Work-Queue, with both an in-memory queue and a streaming hashmap backed by DRAM. The runtime uses three thread roles (Table 2.2): hierarchical Controllers coordinating state transitioning, such as initialization and termination; one Worker per CE, responsible for task claiming and execution; and one Receiver per CE, which handles streaming records, `claim()` requests from CEs in other Neighborhoods, and system signals. This mapping preserves locality while distributing scheduling to avoid bottlenecks.

Controller threads are in a tree-like hierarchy and deliver information in either a fan-in or a fan-out fashion. We implement the controller threads in 4 levels: the global controller, node controller, accelerator controller, and CE controller. To initialize the application, the global controller partitions the data and distributes it down till the controller thread spawns the worker thread. During execution, the controller threads aggregate from workers and inform all receivers about state changes. When all workers reaches the termination state, the controller terminates worker threads in a fan-in fashion and returns control to the user.

Algorithm 1: Worker thread: runtime

```
Function run(claim_count):  
  global uwq_state, next_dest;  
  while uwq_state! = uwq_exhausted do  
    | tasks  $\leftarrow$  claim(claim_count);  
    | if !tasks then  
    |   | tasks, new_dest  $\leftarrow$  hunt(next_dest, claim_count);  
    | if tasks then  
    |   | execute(tasks);  
    | else  
    |   | next_dest  $\leftarrow$  new_dest;  
  send_to_controller(terminate);  
  terminate;
```

Worker threads, when alive, continuously claim tasks and execute locality-aware round-robin work-stealing. Algorithm 1 shows the pseudocode of the Worker runtime. Worker threads perform local load-balancing by *claiming* from the Unified-Work-Queue, and global load-balancing by sending the *hunt* messages. As discussed in Section 2.3.5, each Neighborhood (accelerator) keeps a *next_dest* word in local memory initialized as *current_accelerator_ID* + 1. When a *hunt* message is sent out, it returns a *new_dest* word. When the *hunt* message returns as failed, Worker threads will update the *next_dest* as the *new_dest*.

Receiver threads are implemented as single-event threads, i.e., there is no thread variable dependency between events and the thread lifetime is one event. Such an implementation strategy prevents the Receiver execution from being interrupted and violating the virtual thread limit with a large amount of incoming messages. The Receiver’s functionalities include receiving three types of messages: streaming tasks, hunting messages, and controller signals.

- **Streaming tasks.** Upon receiving a streaming task, the Receiver will first try to look up the key in the streaming hashmap. If the lookup fails, the Receiver spawns a new thread to call *insert*. Otherwise, the Receiver checks if the key is *claimed* and sends

the task to the recorded destination.

- **Hunting messages.** The *hunt* message asks for two objects in return: unexecuted tasks and this CE’s recorded *next_dest*. The Receiver executes the same code of *claim*, and returns the *claimed* tasks (or a flag indicating failing) and its recorded *next_dest* word.
- **Controller signals.** Two events are implemented to intercept and handle two Controller signals respectively: the streaming-end and application-termination signals. Upon receiving the streaming-end signal, the Receiver will set the *uwq_state* to *streaming_static*; Upon receiving the application-termination signal, the Receiver will message the Worker to terminate itself.

2.4.3 Unified-Work-Queue Implementation

Since we prototype Robin-Hood as a MapReduce-style framework, we follow the presumption that no new Map tasks are generated and the Reduce tasks are executed after Map tasks. Therefore, the In-memory queue needs only one state, and the Exhausted states can be combined with others. We thus aggregate and simplify the states, such that the Unified-Work-Queue maintains four states (*in_memory*, *streaming_dynamic*, *streaming_static*, *uwq_exhausted*), cached in local memory for fast access.

The in-memory queue is a flat DRAM array with local head/tail pointers (*in_mem_q.head* and *in_mem_q.tail*), while the streaming hashmap uses a linked-bucket design with its head/tail pointers (*stream_hm.head* and *stream_hm.tail*) and entries cached in fast memory, containing the pointer to data stored in pre-allocated DRAM slots (*stream_hm_dram_ptr*).

Two core operations support scheduling: *claim()*, which atomically advances the queue or hashmap head to allocate tasks, and *insert()*, which appends new streaming tasks by updating the hashmap and initializing the corresponding DRAM entry. Algorithm 2

shows the implementation of the *claim* protocol and how the states change the *claiming* functionality. The interface takes a *claim_count* parameter that indicates how many keys to be taken. The actual *claiming* operation is implemented as an atomic add to the head pointer of one of the two data structures. *Claiming* from the hashmap bears a few more instructions for updating the entry. In addition, the `hunt()` protocol allows Neighborhoods without work to send messages to remote neighborhoods requesting work. This separation allows Workers and Receivers to operate concurrently without centralized coordination.

2.4.4 Static Analysis for Fine-Grained Tasks

As discussed in Section 2.3.7, Robin-Hood applies compile-time analysis to avoid overhead on ultra-fine-grained streaming tasks. Two specialized modes are supported:

- **NHMStream:** Bypasses the streaming hashmap and uses a simple queue when key grouping is unnecessary, reducing pointer chasing overhead.
- **NLBStream:** Executes streaming tasks immediately upon arrival when they are <40 instructions, avoiding the cost of hashmap updates. If instruction count cannot be bounded statically, regular load balancing is used.

These optimizations preserve performance in edge cases while maintaining generality across workloads.

These implementation choices enable us to study whether Robin-Hood’s abstractions and scheduling mechanisms deliver scalable performance across diverse workloads, which we evaluate in the next section.

2.5 Robin-Hood Evaluation

Our evaluation addresses the following questions:

- **Q1:** Does Robin-Hood improve throughput and balance compared to baseline schedulers?
- **Q2:** How well does Robin-Hood scale with system size?
- **Q3:** How do Robin-Hood’s core design features contribute to performance and efficiency?

2.5.1 *Experimental Setup*

Platform. We use a cycle-accurate simulation of the UpDown architecture. Each UpDown accelerator is verified against an RTL implementation. Network and DRAM traffic to remote nodes are simulated as worst-case timing. Our baseline configuration is 8 nodes (16K CEs), with scaling results run on up to 512 nodes (1M CEs). At this size, this might be the largest scale study of work-stealing. As we are running on a simulator, we limit all experiments to a maximum of 48 hours.

Applications. We evaluate nine workloads, with both in-memory and streaming components:

- **Sparse Matrix-Vector Multiply (SPMV):** CSR (Compressed Sparse Row) and COO (Coordinate),
- **Graph Analytics:** PageRank (PR), Triangle Count (TC), Jaccard Similarity (JS),
- **Sorting:** Bucket QuickSort and Bucket Insertion Sort,
- **GCN:** A simplified Graph Convolutional Network layer.

Table 2.3 provides greater detail on each, including whether it is in-memory or streaming intensive, the input type, and how the map and reduce tasks are implemented. The table also shows the results of static analysis.

Most of our results evaluate speedup. To understand baseline performance, Table 2.4 compares the non-load-balancing UpDown implementation to reference numbers from the literature. For the GCN, we compare with a CPU implementation evaluated on a dual-socket AMD EPYC 7763 64-Core machine with Hyperboost enabled. Importantly, these results show that the baseline performance of the UpDown system is already competitive with other systems, so the speedup numbers for our load balancer are meaningful (i.e., we are load balancing computations that are already fast, which should be harder than load-balancing slow computations).

Datasets. Table 2.3 shows each application’s input type. Table 2.5 lists the sparse matrices with different skew selected from Suitesparse [Davis and Hu, 2011]; NNZ indicates the number of nonzeros. We evaluate two sets of graphs from Snap [Leskovec and Krevl, 2014], shown in Tables 2.6 and 2.7. We test PR and TC on 8 large graphs and JS and GCN with 8 small graphs. We evaluate sorting on 8 arrays, each of which is generated by randomly sampling from a distribution that is the sum of two normal distributions. The distribution is in the range of 0 and 10^9 . For each added distribution, the means of the two distributions are at $\frac{1}{3}$ and $\frac{2}{3}$ of the range respectively, and the standard deviations of the two distributions are from 1 to 37.

Baselines. We compare Robin-Hood to:

- **No Load Balancing (NLB):** Tasks are statically assigned. In-memory tasks are equally partitioned for all CEs. Streaming tasks are directed via a static hash function.
- **Work-Stealing (WS):** Based on NLB, tasks are stolen opportunistically, and all streaming tasks are materialized in memory for stealing.
- **Random Robin-Hood:** In this implementation of Robin-Hood, tasks are stolen randomly (inter-Neighborhood) instead of using the locality-aware round-robin algorithm from Section 2.3.5.

While the WS baseline converts streaming tasks to in-memory tasks, we do not include a baseline that converts in-memory tasks to streams, since this requires first extracting them from memory and injecting them into the network. This process is always slower than direct memory reference, adding overhead without scalability, so we omit it from these results.

2.5.2 Q1: Performance and Load Balance

Overall Results Figure 2.4 shows the geometric mean speedups for all baselines and applications. This figure shows that Robin-Hood consistently outperforms both NLB, WS, and Random across all workloads. We include both results of Robin-Hood with NLBStream On and Off; i.e., whether or not the streaming tasks are load balanced. Though static analysis always selects the implementation yielding the best performance. Specifically, we find that:

- Compared to **NLB**, Robin-Hood improves performance by a **geometric mean of $3.22\times$** , by dynamically redistributing tasks in response to workload skew.
- Compared to **Work Stealing**, Robin-Hood achieves a **$1.54\times$** geometric mean speedup by exploiting centralized local queues and avoiding indiscriminate steal attempts.

Robin-Hood handles both in-memory and streaming intensive applications more efficiently, especially under highly-skewed workloads like PageRank and Jaccard Similarity.

Individual Application Results Figure 2.5 shows the individual speedups for each application and input. As shown in Table 2.3, we divide the benchmarks by intensity; i.e., whether their in-memory or streaming tasks have more instructions. One benchmark (COO SpMV) is labeled as none because each task is a single floating point operation. Even on that benchmark (which should be the hardest to load-balance because each task is so short), Robin-Hood achieves a mean $1.04\times$ speedup.

The in-memory intensive benchmarks have imbalanced map tasks. Although the data is equally partitioned among all CEs, the execution lengths of map tasks are data-dependent,

resulting in the load imbalance. For Pagerank, Robin-Hood achieves a 2.84x speedup and out-performs WS by 2.38x. For Map Bucket Sort and CSR SPMV, Robin-Hood achieves 3.11x and 4.02x speedups respectively, and always outperforms WS.

Streaming intensive applications have reduce tasks with input-dependent length or more than 40 instructions on average, therefore Robin-Hood’s static analysis chooses to load balance them. Robin-Hood outperforms WS in all benchmarks with a speedup over NLB from 1.11x to 14.88x. We drop the result of *soc-Epinions1* for JS as it takes much more than 48 hours to finish executing. In the non-graph applications, Bucket Quick Sort and Bucket Insertion Sort, we progressively increase the skewness of the input data by increasing the standard deviation in the data generation model. We note that in Bucket Insertion Sort, performance decreases as the workload gets more skewed, because the largest bucket takes so long to sort that it hides the benefits of load balancing.

2.5.3 Q2: Scalability

Figure 2.6 shows Robin-Hood scales efficiently up to 1M CEs.

Due to the long simulation time, we evaluate with two applications representing in-memory- and streaming-intensive workloads, respectively. Figure 2.6 (a) shows the performance of CSR SPMV. The NLB baseline performance scales up linearly. WS and Robin-Hood scales up sub-linearly, but Robin-Hood constantly outperforms WS. Figure 2.6 (b) shows the performance of Triangle counting. In this case, NLB achieves no speedup beyond 2^{17} CEs, but both WS and Robin-Hood use the increased parallelism and achieve speedup from load-balancing. Again, Robin-Hood consistently outperforms WS by a large margin.

2.5.4 Q3: Design Contributions and Overhead

Robin-Hood’s design features directly improve efficiency:

Unified Abstraction (P1). All tasks are represented using the Unified-Work abstrac-

tion, enabling uniform scheduling for both in-memory and streaming data.

Specialized Queue Structures (P2). We find the Unified-Work-Queue to be low overhead in practice:

- In-memory `claim()` completes in 8–15 cycles.
- Streaming `insert()` and `claim()` operations cost 30–45 cycles depending on load and the size of streaming hashmap.
- Robin-Hood’s state-machine avoids `claim()` operations on exhausted queues reducing steal latency by up to 13%.
- The in-memory queue uses 16B in local memory; the streaming hashmap uses from 10.5 KB (ca-AstroPh, PR) to 600 KB (live-Journal, PR), with 129.8 KB on average. The space is considerably small compared to the 4MB local memory size per accelerator.

Locality-Aware Balancing (P3). CEs with shared local queues initiate stealing using a locality-aware round-robin strategy that avoids redundant probing. We find that the locality-aware round-robin outperforms random search by a geo-mean 10.7%.

Fine-Grained Scalability (P4). Robin-Hood uses compile-time analysis for streaming tasks. As shown above, this analysis always makes the right decision and provides significant speedup with no additional user burden. The control threads are virtual threads and they bear so simple a responsibility that each invocation spends around 10 cycles. The number of invocations depends on its level and the units the controller thread covers, e.g., in the 512 node setting (1M CE), the top-level controller thread is invoked 1027 times, taking about 5 μ s over the lifetime of around 3 hours, while the bottom-level controller threads are always invoked 5 times consistently. These optimizations ensure minimal runtime overheads even for highly irregular, fine-grained workloads.

In addition to these core mechanisms, Robin-Hood incorporates lightweight optimizations that address the edge case of ultra-fine-grained streaming tasks (Section 2.4.4). We examine

these here. For workloads with ultra-short streaming kernels (e.g., under 40 instructions), the NLBStream mode bypassed the streaming hashmap and executed tasks immediately, avoiding overhead that would otherwise dominate execution time. In such cases, Robin-Hood improved throughput by up to 3.90x compared to the baseline mode. Similarly, NHMStream avoided unnecessary hashmap maintenance when key grouping was irrelevant, yielding an up to 12.22x improvement. These results confirm that Robin-Hood preserves efficiency even in pathological fine-grained cases, consistent with Principle 4 (Fine-Grained Scalability).

2.5.5 *Summary*

Robin-Hood delivers high performance and scalability:

- Achieving significant yet consistent speedups across a range of hybrid applications with high skewness using its Unified-Work abstraction (Q1).
- Scaling across a large number of CEs, up to 1M CEs (Q2).
- Reducing overhead with queue design and static analysis (Q3).

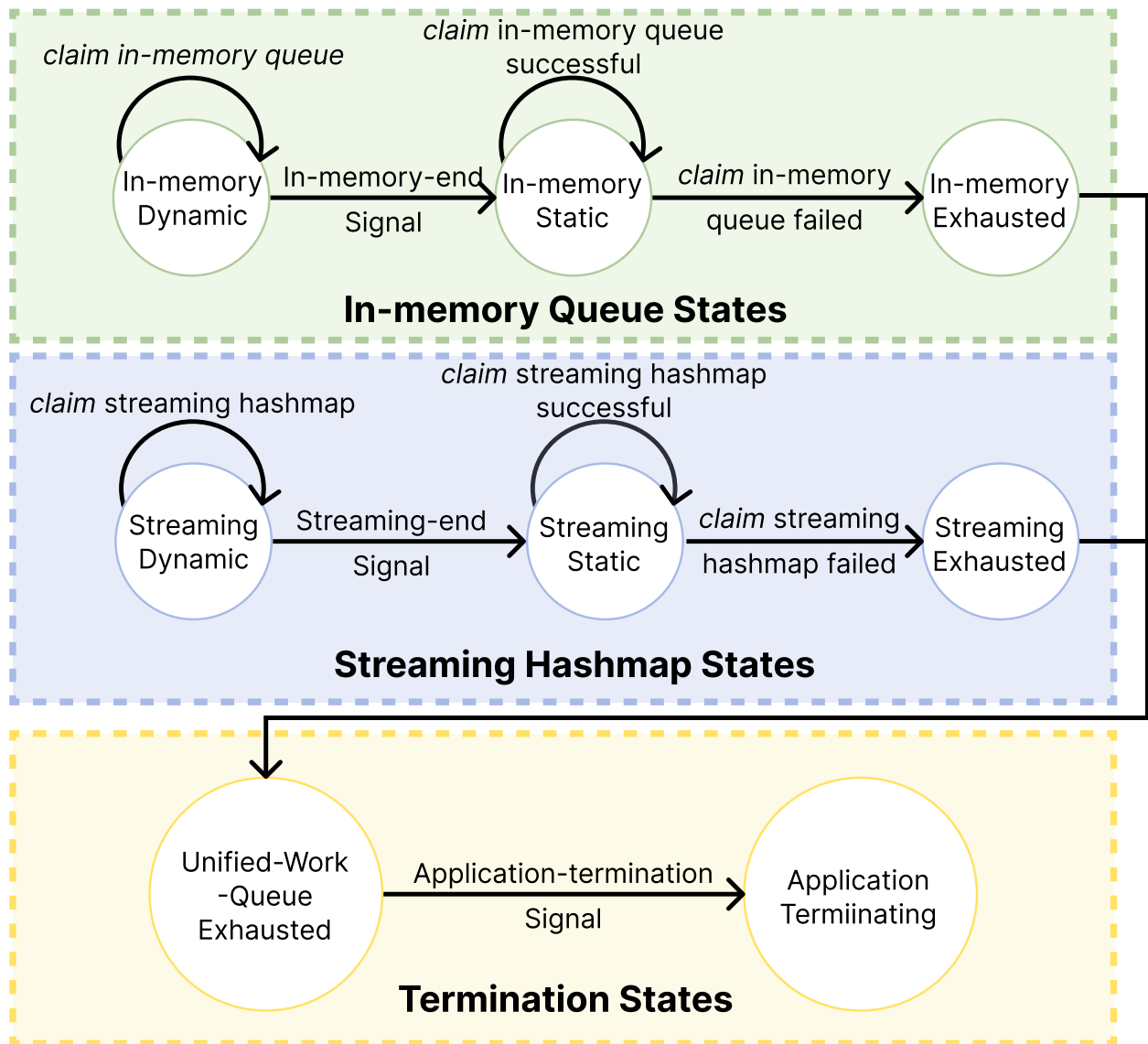


Figure 2.2: Unified-Work-Queue state machine

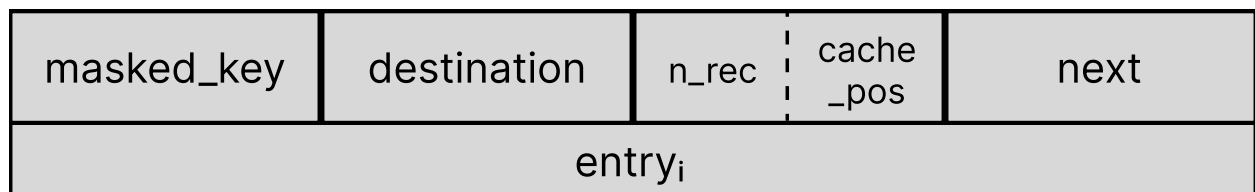


Figure 2.3: Entry design of Streaming hashmap.

Algorithm 2: Unified-Work-Queue: *claim*

States:  In-Memory  Streaming  Exhausted

Returns:  In-Memory tasks  Streaming tasks (Update Hashmap Entry)

```
1 Function claim(claim_count):
2   global in_mem_q, stream_hm, uwq_state;
3   if uwq_state == in_memory then
4     // Try claiming from Work Queue
5     if in_mem_q.head < in_mem_q.tail then
6       comp_and_swap(in_mem_q.head, in_mem_q.head +
7         claim_count);
8       for task ∈ [head, head + claim_count] do
9         | send_back(task);
10      if in_mem_q.head == in_mem_q.tail then
11        | uwq_state ← streaming_dynamic;
12        | // Synchronize empty-Work-Queue
13        | send_to_controller(uwq_state);
14   else if uwq_state != uwq_exhausted then
15     // Try claiming from Work Hashmap
16     if stream_hm.head < stream_hm.tail then
17       comp_and_swap(stream_hm.head, stream_hm.head +
18         claim_count);
19       for
20         entry ∈ [stream_hm.head, stream_hm.head +
21           claim_count] do
22         | entry.key ← mask(entry.key);
23         | dram_addr ← entry.destination;
24         | entry.destination ← this.execute;
25         | for i ∈ [0, entry.n_rec] do
26           | | send_back(dram_addr + i);
27   else if uwq_state == streaming_static then
28     // Work Hashmap empty, set uwq_state
29     uwq_state ← uwq_exhausted;
30     // Synchronize empty-Work-Hashmap
31     send_to_controller(uwq_state);
32     send_back(FAIL);
33   else
34     send_back(FAIL);
35   terminate;
```

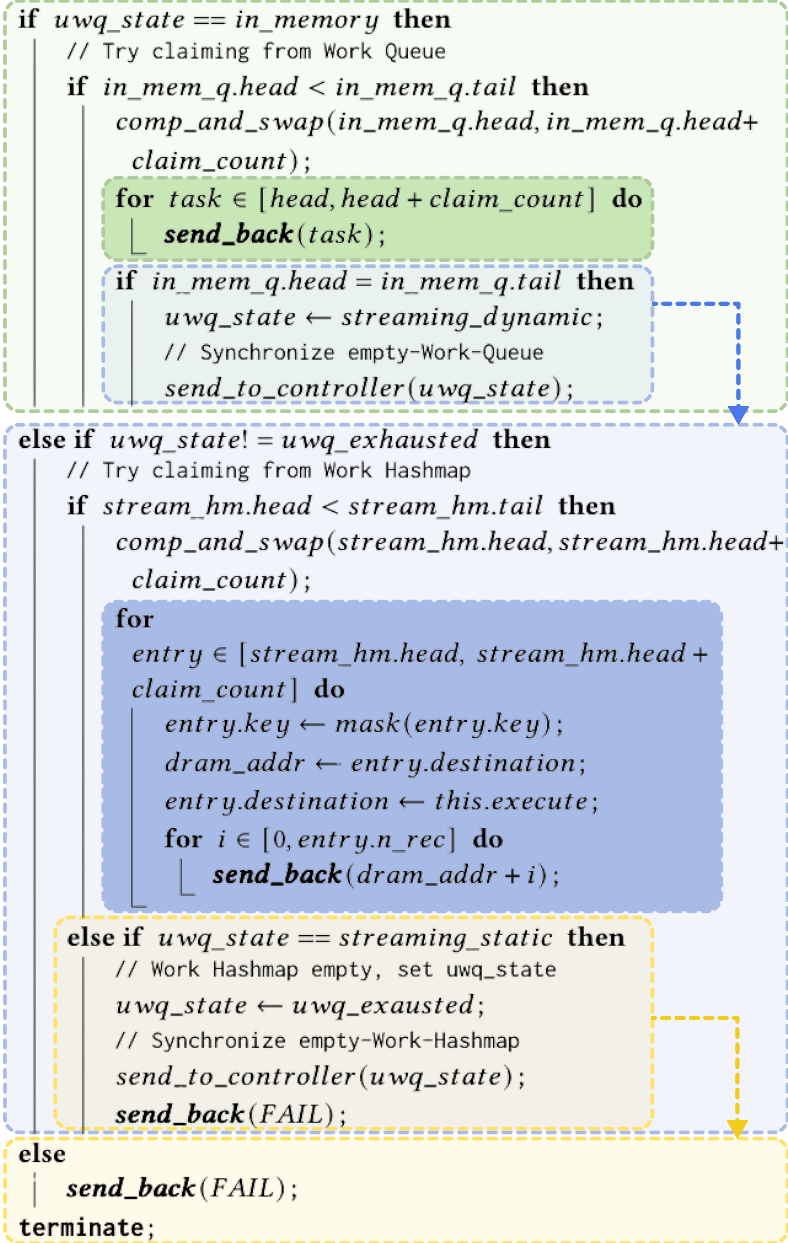


Table 2.3: Benchmark applications

Application	Intensive Phase	Input Type	Implementation			Static Analysis	
			Function	Map	Reduce	NHM Stream	NLB Stream
COO SPMV	None	Sparse matrix	$o = M * v$	$tmp[i][k] = M[i][j] * v[j]$	$o[i] += tmp[i][k]$	✗	✓
CSR SPMV	In-memory	Sparse matrix	$o = M * v$	$o[i] = M[i] * v$	NA	✗	✓
PR	In-memory	Graph	for u, v in edges : $score[u] += 1$	for v in $u.neighbor$: $reduce(u, v) \rightarrow$ $hash(v)$	$score[v] += 1$	✗	✓
TC	Streaming	Graph	$triangle(graph)$	for v in $u.neighbor$: if $(u > v)$: $reduce(u, v) \rightarrow$ $hash(u, v)$	$count_intersection$ $(u.neighbor,$ $v.neighbor)$	✓	✗
JS	Streaming	Graph	$js(graph)$	for v in $u.neighbor$: $reduce(u, v) \rightarrow$ $hash(u, v)$	$count_intersection$ $(u.neighbor,$ $v.neighbor)$	✓	✗
GCN	Streaming	Graph	for u, v in edges : $filtered_feat[u]$ $+= weight[u] * feat[v]$	$reduce(v, u) \rightarrow$ $hash(u)$	$filtered_feat[u]$ $+= weight[u] * feat[v]$	✗	✗
Bucket Quick Sort - Map	In-memory	Random array	$sort(array)$	$quicksort(bucket[i])$	NA	✗	✓
Bucket Quick Sort - Reduce	Streaming	Random array	$sort(array)$	$reduce(bucket[i]) \rightarrow$ $hash(bucket[i])$	$quicksort(bucket[i])$	✓	✗
Bucket Insertion Sort	Streaming	Random array	$sort(array)$	$reduce(n) \rightarrow$ $hash(bucket(n))$	$insert(n, bucket(n))$	✗	✗

Table 2.4: Benchmark baseline performance

Application	Reference Performance	Non-Load-Balancing Performance
COO SPMV	19.53 MFlops/core[Chu et al., 2023]	19.2 MFlops/core
CSR SPMV		11.3 MFlops/core
PR	0.35 mteps/core[Elmougy et al., 2023]	0.54 mteps/core
TC	2.84 Triangles/s/core[Mawhirter et al., 2021]	1.43 Triangles/s/core
JS	0.11 mteps/core[Elmougy et al., 2023]	0.05 mteps/core
GCN	33.36 kteps/core	81.68 kteps/core
Bucket Quick Sort - Map	0.55 Gb/min/core[Jiang et al., 2016]	3.02 Gb/min/core
Bucket Quick Sort - Reduce		1.12 Gb/min/core
Bucket Insertion Sort		0.82 Gb/min/core

Table 2.5: Sparse matrix datasets

Dataset	Dimension	NNZ	NNZ per Row		
			Avg	Max	Std
inline_1	503,712	36.8m	73.09	843	35.63
flickr	820,878	9.8m	11.98	10,272	87.73
HFE18_96_in	4,096	933k	393.48	1,201	319.18
c8_mat11	5,761	2.5m	539.89	1,255	427.66
audikw_1	943,695	77.7m	82.28	345	42.45
mycielskian20	786,431	2.7b	3446.42	393,215	7267.46
as-Skitter	1,696,415	22.2m	13.08	35,455	136.86
Freescall	3,428,755	17.1m	4.97	27	2.07
mip1	66,463	10.4m	155.77	66,395	350.74
torso1	116,158	8.5m	73.32	3,263	419.59
eu-2005	862,664	19.2m	22.30	6,985	29.33
NLR	4,163,763	25.0m	6.00	20	0.84

Table 2.6: Graph datasets for PR and TC

Dataset	Vertex	Edge	Degree		
			Avg	Max	Std
ca-AstroPh	18,772	396,160	21.10	504	30.57
com-youtube	1,134,890	5,975,248	5.27	28,754	50.75
cit-Patents	3,774,768	33,037,896	8.75	793	10.49
live-Journal	4,847,571	68,993,773	14.23	14,815	39.57
web-Google	875,713	8,644,102	9.87	6,332	39.67
flickrEdges	105,938	2,011,648	18.99	61,943	200.78
mico	96,638	2,160,310	22.35	1,359	37.54
com-orkut	3,072,441	234,370,166	76.28	33,313	154.78

Table 2.7: Graph datasets for Jaccard Similarity and GCN

Dataset	Vertex	Edge	Degree		
			Avg	Max	Std
ca-AstroPh	18,772	396,160	21.10	504	30.57
soc-Epinions1	75,879	508,837	6.71	1,801	26.02
musae_facebook	22,470	342,004	15.22	709	26.41
deezer_europe	28,281	185,504	6.56	172	7.95
email-Enron	36,692	367,662	10.02	1,383	36.10
cit-HepPh	34,546	843,156	24.41	846	30.87
ca-CondMat	23,133	186,994	8.08	279	10.63
ca-HepPh	12,008	237,042	19.74	491	46.64

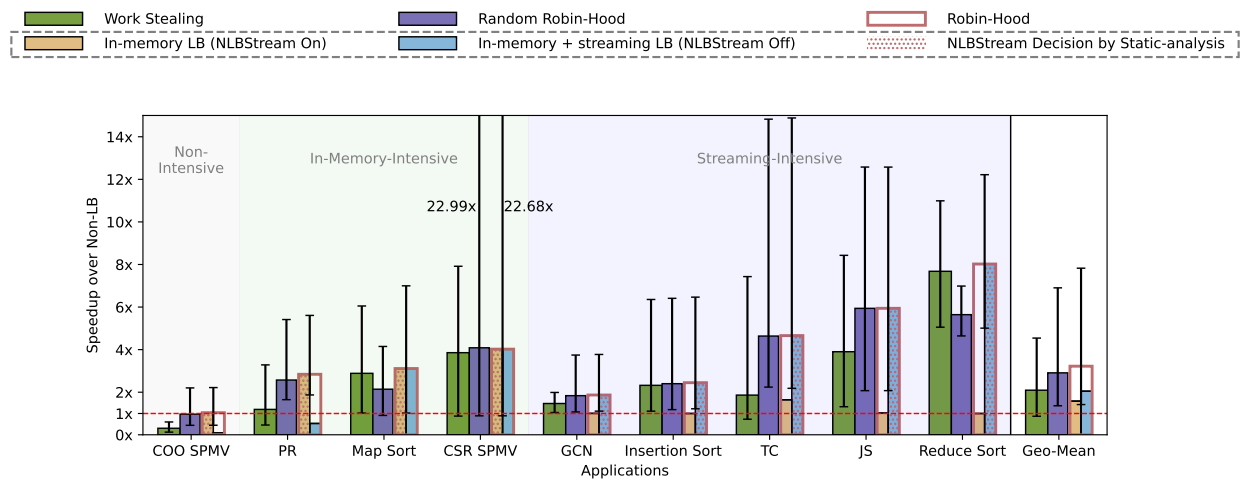


Figure 2.4: Geo-mean performance speedup of Robin-Hood and Work stealing normalized over Non-load-balancing. The error bars represent the minimum and maximum speedups. Robin-Hood includes results of whether NLBStream is on or off and the dot marker indicates which choice is made by the static analysis.



Figure 2.5: Performance speedup of Robin-Hood and Work stealing normalized over Non-load-balancing.

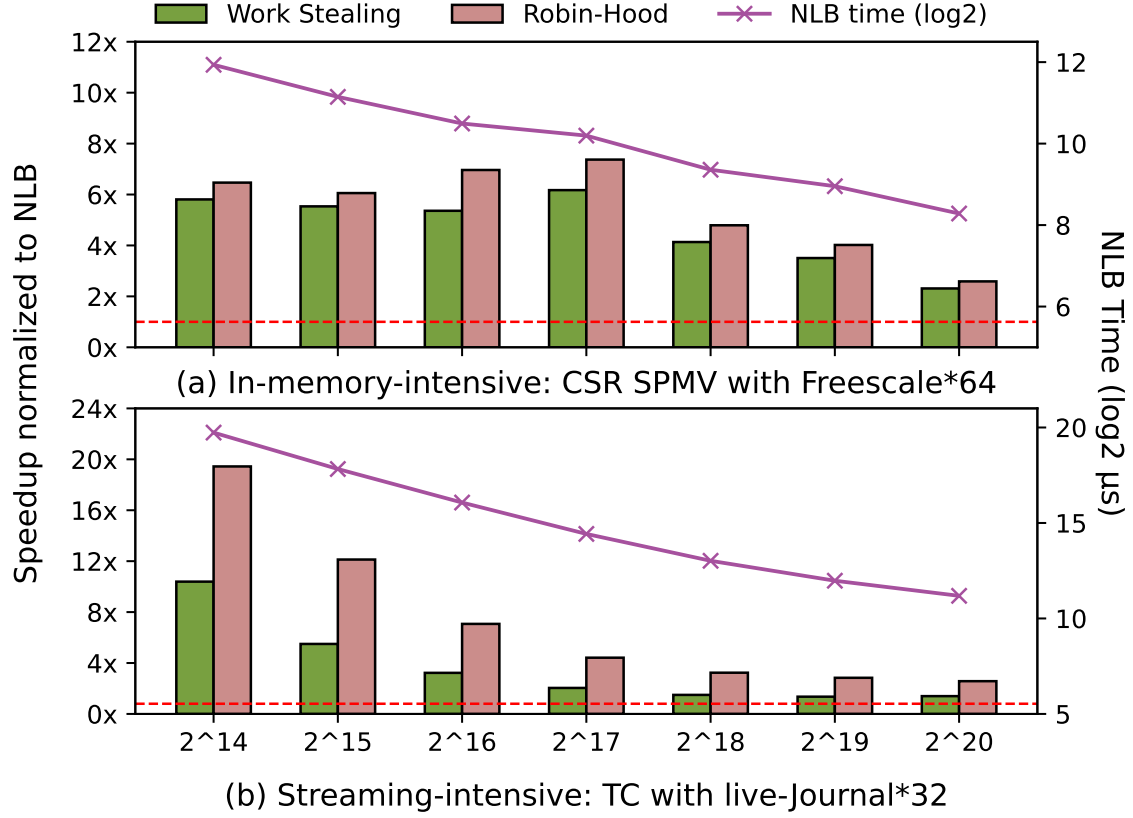


Figure 2.6: The lines show log latency for Non-load-balancing. **(a)** We extrapolate Freescale by 64 times so that it is large enough to have meaningful load to balance at 2^{17} parallelism and evaluate CSR SPMV. **(b)** We extrapolate live-journal by 32 times and evaluate TC.

CHAPTER 3

DPS: ADAPTIVE POWER MANAGEMENT FOR OVERPROVISIONED SYSTEMS

3.1 Introduction

As the scale of distributed systems increases, power becomes the bottleneck of the system design. The US Department of Energy (DoE)’s goal for Exascale Computing is to operate in a power envelope of 20–30 MW [Messina, 2017]. Overprovisioning—building a system which is capable of drawing power over its budget and then limiting individual nodes to respect the budget—has thus been proposed to increase scale while respecting a power budget [Patki et al., 2013]. To support this goal, modern processors include power-capping hardware. Since the Sandy Bridge architecture, Intel processors have supported the RAPL (Running Average Power Limit) interface, which takes a processor power limit and meets it by automatically managing voltage and frequency [David et al., 2010].

In real systems, it is rare that all machines operate at full power. For example, cloud servers typically operate between 10 to 50% utilization [Barroso and Hölzle, 2007]. Thus, unused power budget on low-power machines to be migrated to those operating at their power caps to increase performance. This creates the need for a *power management system* that assigns the cluster-wide power budget dynamically based on power usage such that system performance is maximized.

Prior work proposes three broad categories of power managers:

- **Constant allocation systems** assign an equal power budget to each node. This approach is simple to implement and clearly respects the cluster-wide power budget. However, it is rarely optimal as it cannot shift power dynamically based on demand.
- **Model-based systems** use machine learning or similar approaches [Meisner et al.,

2011, Whiteside et al., 2017, Zhang and Hoffmann, 2019] to construct models that capture the relationship between measurable system behavior and power-performance tradeoffs. These systems then use their models to find performance-optimal power assignments and can dynamically adapt to changes in the operating conditions or workload. But such models require a huge amount of data to train, which creates a non-negligible deployment overhead to migrate such systems to new architectures. They can also be brittle and subject to failure if the system enters an operating regime that is not captured in the training data [Wang et al., 2022a].

- **Model-free stateless systems** (later referred to as stateless systems) use no model, and in fact, keep no knowledge or reference to past operations. Instead, they make decisions based solely on instantaneous power consumption, shifting unused power from nodes operating below their caps, to nodes operating at their cap. This model-free approach works because it is assumed that nodes operating at their cap can achieve good performance with more power and nodes operating below their cap will not be hurt if their cap is lowered. A well-known example of the stateless system is the power management plugin of SLURM (later referred to solely as SLURM) which is currently world-widely deployed on many supercomputers and distributed systems [Yoo et al., 2003]. Such systems are lightweight and ready to operate once configured. They are also robust to different scenarios as they make decisions based solely on current power usage. However, lacking a model, they do not have the means to predict future power states and thus have limited ability to optimize power allocations.

To illustrate the differences between these approaches (and ultimately argue for a new approach), Figure 3.1 compares 4 different power management systems (rows 2 to 5) operating on a two-node (depicted in solid bars) overprovisioned system over 5 timesteps (shown on x-axis). The caps assigned at each timestep are depicted in dotted frames. In this example, both nodes will eventually execute at the maximum power (as shown in the top row with

infinite budget); however, Node 0 increases its power 2 timesteps before Node 1 (shown on the first row). The constant allocation (row 2) wastes part of the budget at T2 and T3 but evenly distributes power at T4. A Perfect Model-based system and a stateless system fully utilize the budget through time T3. At T4, a perfect model-based system reassigns part of Node 0’s budget to Node 1, but a stateless system sees both nodes operating at their power caps and continues with this disproportionate power allocation, starving node 1. This raises the question of whether there is a practically realizable power management system that can arrive at the same place as a perfect model-based system while looking only at power data.

In fact, this paper argues that there is an important, yet unexplored, middle ground between model-based systems and stateless model-free systems. Specifically, *power dynamics*—changes in power usage over time—reveal a great deal of information that can be used to allocate power budgets in a distributed system without reliance on complicated models. Consider again the example in Figure 3.1. From time T1 to T3, node 1’s power increases and it is likely that it will need more power in the future. By tracking these changes, power can be redistributed based on the system’s dynamics, without reliance on a complicated model.

Based on these observations, we present the Dynamic Power Scheduler (DPS), a model-free stateful power manager for overprovisioned systems that improves performance while maintaining a cluster-level power limit. Inheriting the low-overhead advantage of the model-free stateless approaches, DPS assumes no *a priori* knowledge of application workloads and monitors only power data. DPS decides the power cap for each node by first using a stateless approach and then updates this initial power distribution using *power dynamics*, which include an estimate of each node’s first derivative of power and its frequency of power changes. These power dynamics are then converted into a priority that modifies the initial power distribution to ensure that nodes that either (1) need power now or (2) will likely need power in the near future get that power. DPS also uses power dynamics to ensure that when power usage fluctuates too quickly to benefit from active management, it never produces

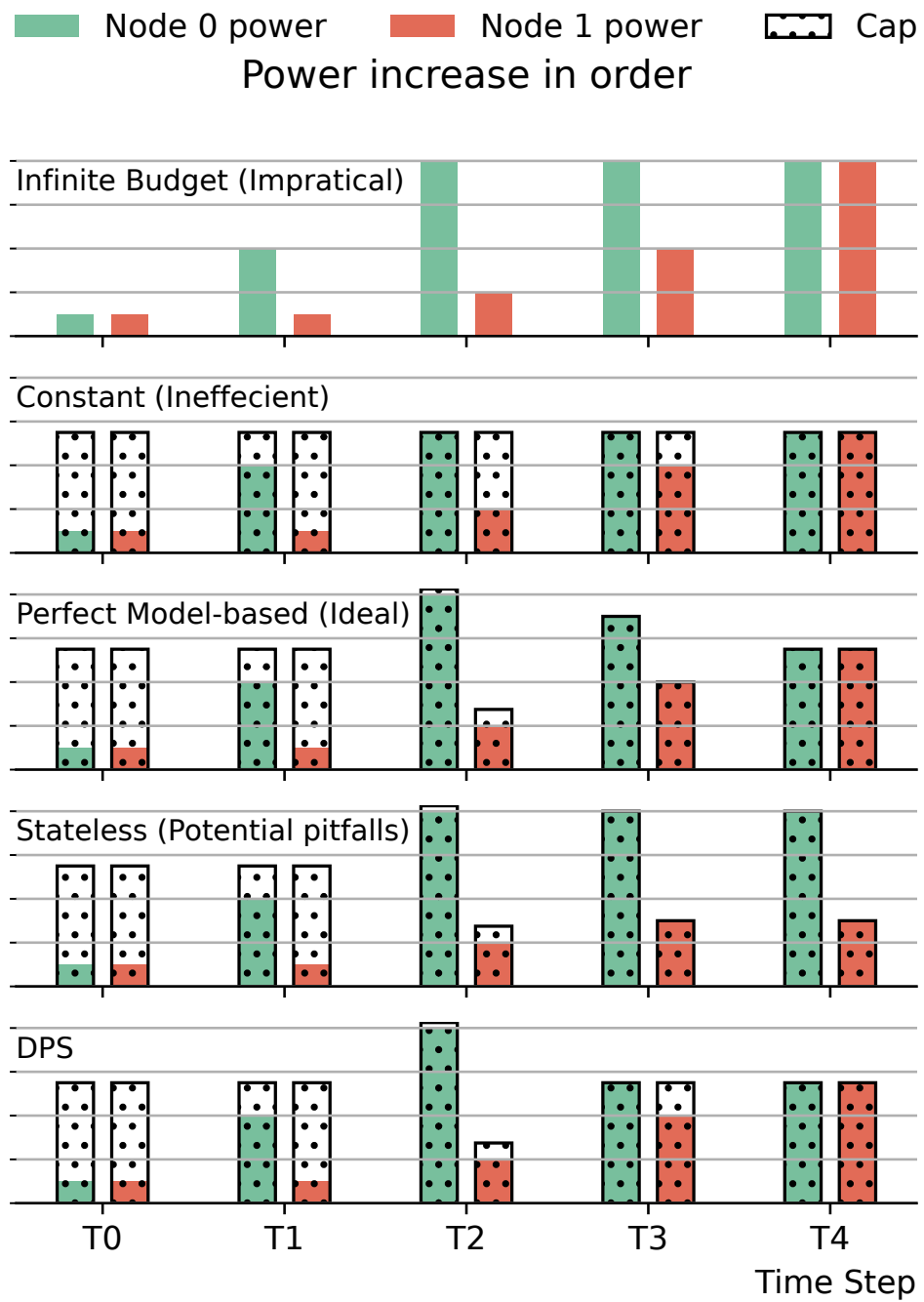


Figure 3.1: Motivational Example: power management systems

worse performance than a constant allocation.

Figure 3.1 includes DPS as the last row. DPS makes the same decision as the stateless system until T2. When the power needs exceed the power budget at T3, different from the stateless system, DPS detects the increasing trend of Node 1’s power and adjusts the caps to the anticipated power need. Then at T4, it is able to continue assigning balanced caps just as a perfect model-based system does.

We implement DPS and test it on an 11-node Linux/x86 server cluster with the machine learning workloads in the HiBench benchmark suite [Huang et al., 2012] and an HPC benchmark suite, the NAS Parallel Benchmarks (NPB) [Bailey et al., 1995]. The Spark applications demonstrate distinct phases in power changes while the HPC applications continuously require high power. Machine learning applications are implemented and executed in Apache Spark, and categorized into two groups based on the application’s power usage. We test in three experimental setups where two clusters are run in parallel to reflect a real-world cloud service utility. In the first setup where mid-power Spark applications are paired with low-power Spark applications, our results show that DPS achieves similar mean performance to prior stateless systems, but with a maximum improvement in application performance of up to 17.5%. In the second setup where high-power and mid-power Spark workloads are paired together, DPS outperforms the power-management plugin in the wide-used resource manager, SLURM by a mean 5.4%. In the third setup where Spark applications are paired with NPB applications, DPS out-performs SLURM, by a mean 8.0% and up to 21.3%.

We acknowledge that a highly constrained set of application workloads (and inputs) could benefit from 1-time training in a model-based approach, which, however, bares a non-negligible overhead beyond such a constrained scenario, as we believe HPC (and cloud) workloads are becoming increasingly diverse and dynamic. On the contrary, DPS can be deployed on any cloud system, including commercial cloud systems like elastic clouds. DPS can also be deployed in systems where the applications are black-boxes where Model-based

systems are essentially unable to operate.

To our best knowledge, this paper proposes the first model-free stateful power manager and makes the following contributions:

- Developing a model-free stateful power management system to maximize performance under a power cap. The state is simply the recent power usage changes, which we refer to as power dynamics.
- Proposing a new methodology for designing power management systems based only on recent power behavior.
- Making all scripts, code, and data available as open source, so others can test or extend these results.

3.2 Related work

Power and energy have become first-order concerns for computer system design. Therefore, power management has been proposed at both node and cluster-level. We first briefly cover node-level designs before reviewing cluster-level solutions (the topic of this paper). Cluster-level solutions, of course, require some node-level support [Bianchini and Rajamony, 2004]. Several OS projects add support for node-level power monitoring and energy allocation [Fonseca et al., 2008, Roy et al., 2011, Snowdon et al., 2009, Shen et al., 2013b, Weissel et al., 2002, Yuan and Nahrstedt, 2003, Vardhan et al., 2009, Hoffmann and Maggio, 2014, Imes et al., 2019]. Several studies profile applications or hardware-level metrics to improve energy efficiency [Hoffmann, 2015, Mishra et al., 2015]. To support setting power limits, Intel’s SandyBridge and later processors provide power management in hardware [David et al., 2010]. RAPL provides an interface for specifying a power limit and then keeps the processor at or below that limit.

As we enter the exascale era, the United States Department of Energy (DoE) is anticipating a cluster-level power envelope of 10s of Megawatts (MW)[Messina, 2017]. In this scenario, it is anticipated that large-scale systems for supercomputing and datacenters will be *overprovisioned*; that is, it will not be possible to power each server at its TDP (Thermal Design Power) while still respecting the cluster-level budget. For example, prior work proposes improving performance for HPC systems by overprovisioning and using RAPL to set node-level power limits below TDP [Patki et al., 2013]. In industry, Google has deployed overprovisioned systems worldwide for several years [Sakalkar et al., 2020]. The overprovisioned system has become a practical and critical approach to addressing tight cluster-level power limits in current and future cloud clusters. As introduced in Section 3.1, prior works on cluster-level power management systems can be generally separated into three categories and this section covers each.

3.2.1 *Constant Allocation Systems*

Some early works in cluster-level power management explore constant allocation systems that set a static power cap for each node in the system [Ge et al., 2005, Rountree et al., 2009]. Patki et al. [2013] first explored application scalability in a strictly overprovisioned system with a constant allocation system using RAPL to set node-level power limits. Sarood et al. [2013] extended this system to include power limits on DRAM. Constant allocation systems assure cluster-level power limits and are easy to implement (given RAPL’s hardware support), but using the same peak power limit for all nodes leads to sub-optimal application performance, as nodes with compute-heavy workloads run at the limits, and other nodes run below their limits, making poor use of the available power.

3.2.2 *Model-based Systems*

Researchers quickly realized the limits of constant allocation and turned to solutions that dynamically allocate power. A prominent class of model-based systems uses machine learning to make decisions on how to set node-level caps to meet cluster-wide budgets. For example, Lee et al. [2016] models I/O behavior to dynamically adjust per-node power budgets by shifting power from I/O bound nodes to compute bound ones. The idea of power management via a workload characterization model has been studied within various types of services, including online data-intensive services [Meisner et al., 2011], parallel workloads [Huang and Feng, 2009], non-interactive workloads [Feng et al., 2005], visualization workloads [Brink et al., 2021], and microservices [Hou et al., 2020]. Several works build job performance models to manage an isolated component or coordinate multiple components [Sarood et al., 2014, Georgiou et al., 2014, Labasan et al., 2017, Gholkar et al., 2016, Marathe et al., 2015]. Machine learning models are also widely studied for power management. The PowerShift project models coupled applications offline to make power cap decisions based on its model’s predictions [Zhang and Hoffmann, 2018]. The PoDD project upgrades this idea by building machine learning models online [Zhang and Hoffmann, 2019]. The PANN project uses neural networks to dynamically allocate power in overprovisioned systems [Whiteside et al., 2017]. Some other works build feedback-based models to improve power efficiency. Wang et al. [2008] establishes a feedback control framework to improve power efficiency with DVFS. Several feedback-based systems are proposed to adaptively apply power capping and maximize performance [Wang and Chen, 2008, Zhang and Hoffmann, 2016b, Gholkar et al., 2018, Marincic et al., 2020]. Model-based systems generally achieve near-optimal performance under the assumption that their models have sufficient training data. Once trained, their optimality (and sometimes even ability to meet the power budgets) is dependent on the assumption that runtime workloads are drawn from the same distribution as their training data. If the architecture or workloads change significantly from this training set, the models

will no longer maximize performance, and in some cases might fail to maintain the power budget.

3.2.3 *Stateless Model-free Systems*

Stateless model-free systems eliminate the dependence on a well-trained model. SLURM is a widely used cluster job and resource management system that incorporates a power management plugin. SLURM’s power management system is a stateless system that maintains a cluster-wide power limit by setting node-level power caps using only the current power measurements [Yoo et al., 2003]. A framework based on SLURM provides safe hardware overprovisioning on a 965-node cluster at Kyushu University [Sakamoto et al., 2017]. The Argo project incorporates a conclave-node two-level stateless power management system [Ellsworth et al., 2015a,b, Perarnau et al., 2015, Ellsworth et al., 2016]. The Fukagu supercomputer incorporates a core retention mode to turn off idle nodes based on current processes and improve energy efficiency [Kodama et al., 2020]. These systems demonstrate the practical benefits of approaches that eschew models. However, stateless model-free systems make power assignments using only current power usage, forcing a greedy optimization strategy. They lack the ability to predict future power usage and escape local optima, providing sub-optimal performance, especially when workloads transition through phases of high- and low-power needs.

To the best of our knowledge, only model-free systems are deployed in production clusters. As of June 2022, SLURM is the default resource and job manager for 5 supercomputers in the top 10 of the Top 500 list, including Frontier, LUMI, Sunway TaihuLight, Perlmutter, and Tianhe-2A [top, 2022]. Fukagu still remains as the second top supercomputer in the world. Therefore this paper takes the position that a power management system that can be applied in real-world systems should not rely on a model that introduces a high deployment cost and a reliance on well-calibrated training data. Thus, this paper explores a model-free

power management system that can escape local optima by incorporating *power dynamics*.

3.3 Power Management with Dynamics

Overprovisioned systems must divide a cluster-wide power budget among individual nodes. We refer to the maximum power that a node could draw as its *power demand*. Generally, meeting a node’s power demand will result in higher performance; compute-bound nodes tend to have high power demands, while memory- and IO-bound nodes tend to have lower demands. So, a power manager should meet a node’s demands whenever possible. However, sometimes there is not sufficient budget to meet all nodes’ demands simultaneously. In this case we would like to ensure each node is equally penalized so the power budget is fairly distributed according to demands. We note that this is a novel definition of fairness—instead of evenly distributing power among units, we seek to distribute power proportional to each node’s demands. In other words, we define fairness based on a node’s power demand rather than its absolute power.

Although it is difficult to quantify how much a single node’s power demand is capped at a certain time, we can define a more general term of how a node’s power demand is satisfied throughout the lifetime of a workload, as the node’s *satisfaction*. Equation 3.1 shows the definition of *satisfaction* of Node n . Then we can further define the fairness of allocating the power budget to two nodes. As shown in Equation 3.2, the *fairness* of Node i and j is unity minus the absolute difference between the satisfaction of i and j . The *fairness* is between 0 and 1, and the higher the *fairness* is, the two nodes are getting closer percentages of the power budget they demand. As will be discussed later in section 3.6, a higher *fairness* will generally result in a higher harmonic mean performance of different workloads.

$$satisfaction(n) = \frac{\text{average power under current cap}}{\text{average power under no cap}} \quad (3.1)$$

$$fairness(i, j) = 1 - |satisfaction(i) - satisfaction(j)| \quad (3.2)$$

While conceptually simple, meeting power demands fairly is challenging in practice for two reasons: (1) the true demand cannot be measured directly as the system might be capped and (2) the demands vary dynamically, so even once the demand is known at some point, it is not clear what it will be in the future. Because different machines may support different power management scales (cores, sockets, or nodes), in the following text we refer to each part of a machine that supports power capping individually as a unit.

3.3.1 *Fluctuating Power Demands*

Power demand is continually changing. At any time, a unit’s power demand depends on its application workload and even different phases within an application. Figure 3.2 shows three Apache Spark applications—LDA, Bayes, and LR—and their power over time, executed separately without power limit. We refer to the power consumption that an application would exhibit without a cap as its *power demand*. From the figures we summarize three observations about these applications’ power dynamics; i.e., their changes in power demand over time.

- **Power phase duration is diverse.** Applications put different computing loads on the system at different times, resulting in power phases. In Figure 3.2a, LDA has a long phase ranging from second 0 to 125. Figure 3.2c shows LR has many phases shorter than 10 seconds, for example from second 140 to 149. Figure 3.2b shows phase durations in between these extremes, but each phase has a different length. One longer duration ranges from second 50 to 75, but a shorter one lasts only 13 seconds, from second 235 to 248.

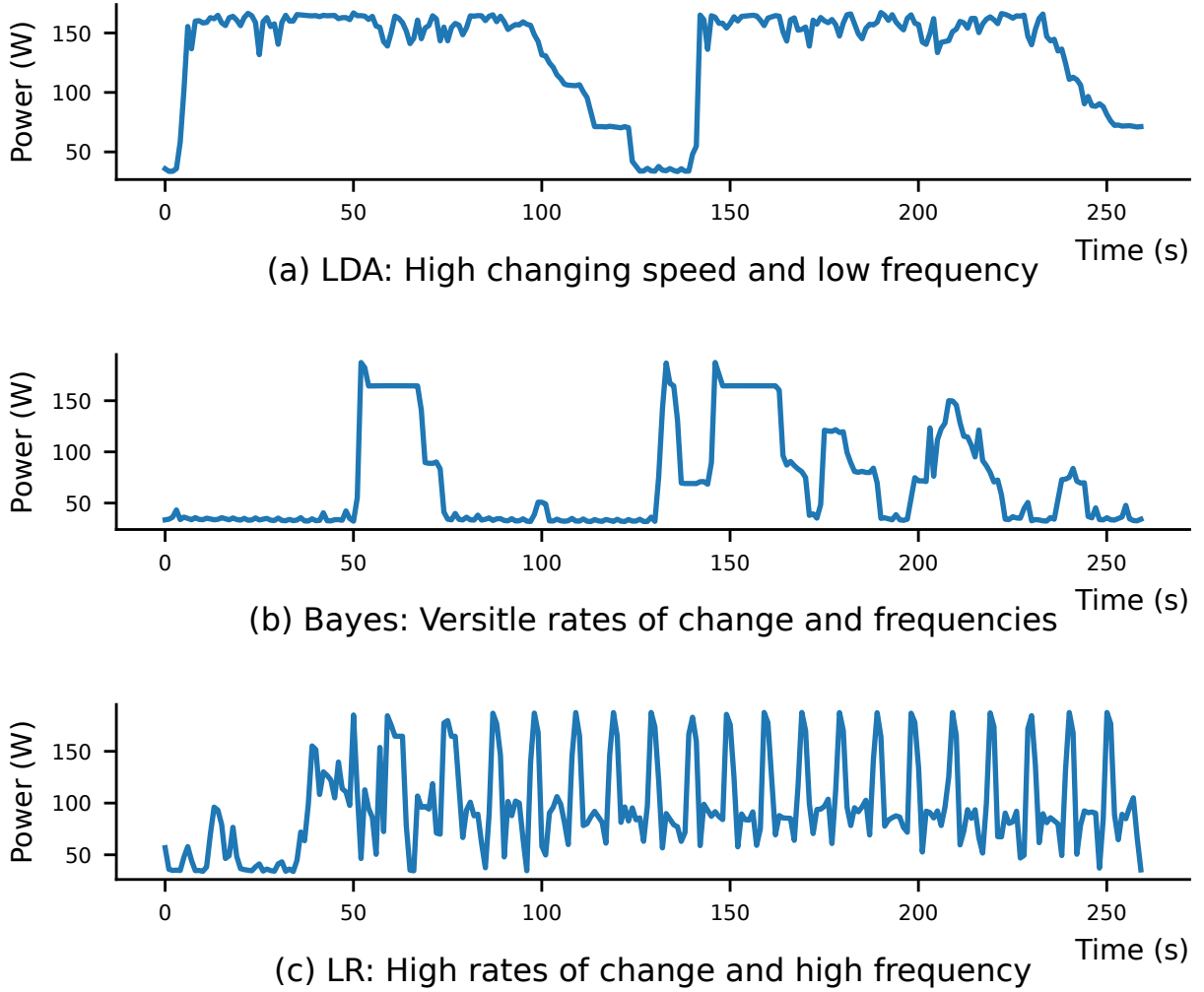


Figure 3.2: Power phases for different applications.

- **Peak power at each phase is diverse.** In Figure 3.2b, Bayes's peak power is different for different phases. For example, from second 50 to 75, power increases to 165W, but from second 175 to 192, the peak power is only 110W.
- **The first derivative of power is diverse.** When power changes, its first derivative also differs based on the current computing loads. In Figure 3.2a, power increases from 20W to 160W in 3 seconds starting from second 3, but decreases slowly from 160W to 70W in 20 seconds starting from second 97. In Figure 3.2b, changing speeds are different in different phases: from second 50 to 75, power both increases and decreases

Table 3.1: Terminology.

Variable	Description
t	Timestep t
T_t	Time at t
dT	The granularity of timesteps
E_t	Energy consumption till t
P_t	Power consumption at t
$\frac{dP_t}{dT}$	Derivative at t
$E_t(P)$	Estimated power at t
C_t	Power cap at t

quickly, but from second 195 to 225, power both increases and decreases slowly.

3.3.2 Power Management Challenges

Under a system-wide power budget, the power dynamics observed above lead to two power management challenges:

- **Matching the power demands and the budget through application phases.**

To respect all units' demands fairly, the power management system should not over or under allocate the budget to any unit. Under-allocating limits that unit's performance, while over-allocating means less budget is available for other units. Yet the diversity in both peak power and power derivatives makes it hard to match the allocation to demands fairly.

- **Handling high-frequency power changes.** Power management decisions take time.

But the diverse power phase duration can be too short for the system to react. For example, in Figure 3.2c the periods of high and low power usage are very short and the power manager must determine if it can reallocate power fast enough to improve performance. If it cannot, there is a risk that overall performance is worse as the system reacts too slowly, providing extra power just as the application transitions to a low-power phase and taking power just as it transitions to high power.

3.3.3 Power Dynamics

A key observation in this work is that power changes with inertia; power changes are locally predictable, meaning we can estimate near-future power changes from recent power history. We define the components in the power history that are useful for this estimation as *power dynamics* and identify two key components: the first derivative and the frequency. Relevant symbols and terminology are included in Table 3.1.

The estimation of the first derivative is as defined in equation 3.3. It indicates the direction and magnitude of power changes. If it is positive, power is increasing, and vice versa. The absolute value of the first derivative indicates the speed of change. Though it does not allow us to predict the exact power consumption, it implies the direction that the power is most likely to go. If power is increasing quickly, power will likely either increase or stay high in the near term. Similarly, if power is decreasing quickly, we can predict the unit will not need more power budget in the near term.

$$dP_t = \frac{P_t - P_{t-1}}{dT} \quad (3.3)$$

But such prediction can be wrong when the frequency change is high. Therefore the frequency is another important element in the *power dynamics*. The frequency is inferred from the number of power peaks in the recent past. The frequency may be too fast for the power management system to keep up with. In this case, it is important to know the reaction speed of the power manager, and have a separate power allocation mechanism that handles the case when the power change frequency is faster than the power management system's reaction.

3.4 DPS Design

DPS is a model-free stateful power manager. It builds off of prior stateless approaches, but

distinguishes itself by taking *power dynamics* into consideration and attaching a priority to each power-allocation unit based on the observed dynamics.

3.4.1 Design Principles

DPS has two fundamental design goals: overhead minimization and performance maximization. Overhead includes both operation and deployment. The operating overhead is the cost of making decisions, including data collection, signal transition, power cap setting, etc. The deployment overhead is the preparation cost for the system to operate, for model-based systems, this can be large. In summary, the goals of low overhead (deployment and operational) and high performance lead to the following design principles:

- **Minimum load on applications.** The system should have minimal impact on application performance, including the cost of monitoring and setting power, communication, etc.
- **No initial profiling or data-driven model building.** A model of any kind that involves profiling or training data will dramatically increase overhead compared to a model-free system. Instead of relying on such a pre-trained model to predict the *power demand*, DPS incorporates *power dynamics* to make decisions on power caps.
- **Only power usage data is involved.** Making decisions based on any metric other than power usage requires additional operational and deployment overhead. DPS thus uses only unit-level power consumption data.
- **Ensures the same lower-bound performance as constant allocation.** For any power management system that makes dynamic decisions, there is a risk that it may make worse decisions than simply dividing the cluster-wide power budget equally for each unit, resulting in a lower *fairness* than what constant allocation guarantees. Indeed, prior work can reduce performance compared to constant allocation in certain

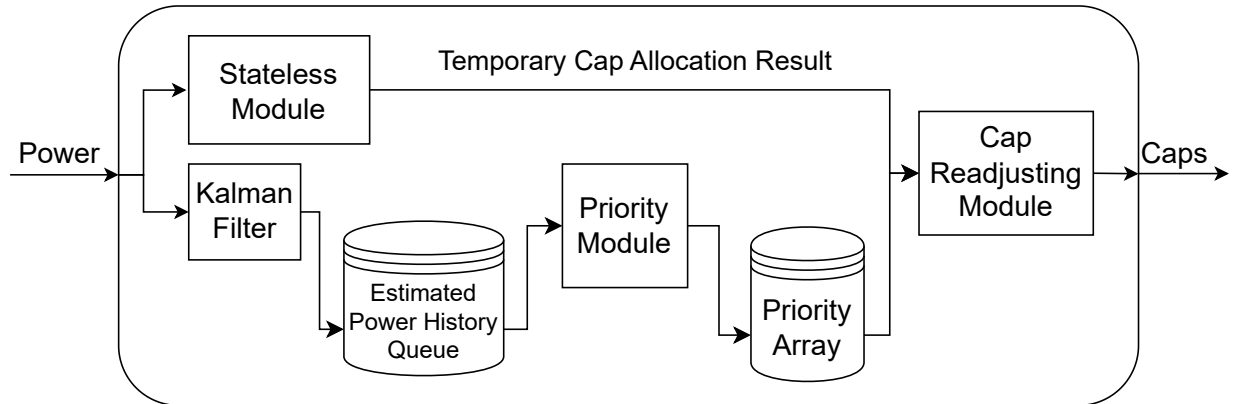


Figure 3.3: DPS control system design

scenarios (see Section 3.6). However, DPS ensures the same lower-bound performance as constant allocation because it incorporates *power dynamics*, so it knows when power behavior will be poorly suited to dynamic reallocation and it switches to a constant approach.

3.4.2 Hardware Support: RAPL in DPS

DPS interacts with the hardware in 2 ways, reading power usage data and setting power caps. Both abilities are supported by Intel’s RAPL system [David et al., 2010]. RAPL observes various low-level hardware events and estimates energy consumption based on event counters. Still, the accuracy of RAPL’s energy readings is shown to be high and the overhead is low [Khan et al., 2018].

Although DPS uses RAPL to read power and set the power caps, it is not tied to the RAPL interface. DPS only needs to interact with the hardware in these two ways and it can be implemented with any interface with these functionalities.

3.4.3 DPS Control System

DPS consists of a server on a central node and clients on each computing node (similar to

SLURM’s power management plugin). Each client is responsible for reading power, setting caps, and communicating with the server for all the power capping units on the node. The control system is excluded from the client to minimize operating overhead. The server keeps a list of all the power capping units, including their current power caps, estimated power histories, and priorities, which we will go into detail about later in this section. The control system on the server, as shown in Figure 3.3, consists of 4 modules and two important global data sets. At each time step, the control system receives current power usage data from all computing nodes. The stateless module takes in current power and outputs a temporary cap allocation decision. The priority module takes in the power history and updates the priorities attached to all units. Although RAPL has been verified by previous work to deliver reliably high accuracy, noise exists in power usage traces and we further assume pessimistically that RAPL bears certain measurement noise. Therefore we assume the exact power is not known, but is a hidden variable that must be estimated from these noisy measurements. To address such noisy measurements, DPS incorporates a Kalman Filter that takes the (potentially noisy) power measurements and updates the estimated power history, which stores the recent power dynamics for each unit. This power history is then fed into the priority module. The cap readjusting module finally takes the priorities and the caps produced by the stateless module and outputs the cap decisions, internally modifying the caps set by the stateless system based on the priorities. The control system then sends the new caps to all computing units. This section introduces each individual module and how the control system incorporates *power dynamics* to provide improved power cap allocation compared to a stateless system, without requiring any additional measurements or models.

Stateless Module

As indicated by the name, this module is a stateless power management system, which takes in the current power and decides the power caps only based on the current power. This cap

Algorithm 3: Stateless module

```
1 Function multp_inc_multp_dec(inc_threshold, dec_threshold, inc_percentile,  
   dec_percentile):  
   // power: list of current power consumption of all units  
   // cap: list of current power cap of all units  
   // set_flag: list of flags for all units of whether the cap is changed  
2   global power, cap, set_flag;  
   // Initialize set_flag to 0  
3   for u  $\in$  units do  
4      $\lfloor$  set_flag[u]  $\leftarrow$  0 ;  
   // First loop: decrease caps  
5   for u  $\in$  units do  
6     if power[u] < cap[u] * dec_threshold then  
7        $\lfloor$  cap[u]  $\leftarrow$   $\max(\text{power}[u], \text{cap}[u] * \text{dec\_percentile})$  ;  
8        $\lfloor$  set_flag[u]  $\leftarrow$  0;  
   // Second loop: increase caps in random order  
9   avail_budget  $\leftarrow$  total_budget -  $\text{sum}(\text{cap})$ ;  
10  for u  $\in$   $\text{random}(\text{units})$  do  
11    if power[u] > cap[u] * inc_threshold then  
12       $\lfloor$  tempt  $\leftarrow$   $\min(\text{cap}[u] * \text{inc\_percentile}, \text{avail\_budget})$ ;  
13       $\lfloor$  cap[u]  $\leftarrow$  tempt;  
14     $\lfloor$  avail_budget  $\leftarrow$  avail_budget - tempt;
```

result serves as a temporary basis to be readjusted by the cap resetting module. As shown in Algorithm 3, the stateless module is a Multiplicative-Increase-Multiplicative-Decrease (MIMD) based controller, inspired by SLURM’s power management system [Yoo et al., 2003]. It maintains two thresholds for increasing and decreasing the power cap respectively, set at percentages of the current cap. If a unit’s current power is below the decreasing threshold, its power cap will be decreased by a percentage or to its current power. If the current power of a unit is above the increasing threshold, its power cap will be increased by a percentage or by what is left in the cluster-wide budget. The cap-increasing loop is done in a random manner so that no unit has priority in increasing the cap over others.

Kalman Filter

The Kalman Filter (KF) produces an estimation of a joint probability distribution over a single measurement for each time frame by taking unknown noise and variance as other variables. We use the version of a 1-dimensional Kalman Filter in the standard formulation that renders minimum computing loads while providing reliable estimations [Welch et al., 1995]. More specifically in DPS, at each time step, the KF module calculates the Kalman gain based on the estimation uncertainty, which is updated after each estimation, and produces an estimation of current power. The estimation will be pushed into the global estimated power history queue.

Priority Module

The priority module estimates the two properties of power dynamics, the frequency and the first derivative, and attaches a priority of either high or low to each unit.

As shown in algorithm 4, the priority module keeps track of the changing frequency of units' power phases. At each time step, the module first identifies the high-frequency units and attaches high priorities to them. For those who have already been identified as high-frequency units, the module checks both the number of prominent peaks [Palshikar et al., 2009] and the standard deviation over the power history. If the two measures are both below their thresholds, they are identified as low-frequency units. The additional check on the standard deviation is because of the uncertainty in the power and the fixed threshold in calculating the number of prominent peaks. Sometimes the number of prominent peaks can fall below the threshold yet power is still changing with high frequency, the standard deviation identifies such scenarios.

After attaching the priority according to the frequency, the module estimates the first derivative for all units that are identified as low frequency. Two thresholds are involved in classifying the first derivative, one positive and one negative. If the first derivative is above

Algorithm 4: Priority Module

```
1 Function set_priority(inc_threshold, dec_threshold, std_threshold, pp_threshold,  
   direv_length):  
   // duration_history: list of duration of each power reading  
   // high_freq_flags: list of flags of whether the unit is demonstrating  
   // high-frequency power changes  
2 global power_history, duration_history, high_freq_flags, priority_arr  
3 for u  $\in$  units do  
4   pp_count  $\leftarrow$  count_prominent_peaks(power_history[u], inc_threshold)  
5   if not high_freq_flags[u] then  
6     if pp_count > pp_threshold then  
7       high_freq_flags[u]  $\leftarrow$  True  
8       priority_arr[u]  $\leftarrow$  True  
9       continue  
10  else  
11    if pp_count < pp_threshold and std(power_history[u]) < std_threshold  
12    then  
13      high_freq_flags[u]  $\leftarrow$  False  
14      priority_arr[u]  $\leftarrow$  False  
15      continue  
16    if not high_freq_flags[u] then  
17      avg_direv  $\leftarrow$   $\frac{\text{power\_history}[-1] - \text{power\_history}[-\text{direv\_length}]}{\text{sum}(\text{duration\_history}[u][- \text{direv\_length}:])}$   
18      if avg_direv > inc_threshold then  
19        priority_arr[u]  $\leftarrow$  True  
20        continue  
21      if avg_direv < dec_threshold then  
22        priority_arr[u]  $\leftarrow$  False  
23        continue
```

Algorithm 5: Cap Readjusting Module: Restore

```
1 Function restore(inc_threshold):  
2   global power, cap, set_flag  
   // Restore all caps to the initial cap if no unit is consuming high power  
3   restore_flag  $\leftarrow$  True  
4   for  $u \in \text{units}$  do  
5     if  $\text{power}[u] > \text{initial\_cap} * \text{inc\_threshold}$  then  
6       restore_flag  $\leftarrow$  False  
7       break  
8   if restore_flag then  
9     for  $u \in \text{units}$  do  
10       $\text{cap}[u] \leftarrow \text{initial\_cap}$   
11       $\text{set\_flag}[u] \leftarrow \text{True}$ 
```

the positive threshold, a high priority is attached. If it is below the negative threshold, a low priority is attached. If it is in between, the unit's current priority is not changed. The positive threshold detects fast power increases, and the negative threshold detects fast decreases. The reason behind using two thresholds is that after the power change, the unit's priority should be kept unchanged until the power changes again. For example, if a unit is set as high-priority when its power increases, it should be considered as high-priority until tasks are finished and power decreases.

Cap Readjusting Module

After the Stateless Module makes a temporary cap allocation decision, some of the cluster-wide power budget could be unassigned. The cap readjusting module allocates this unassigned budget to all the high-priority units. If there is no power budget left after the Stateless Module's adjustment, this module instead readjusts the caps of all the high-priority units to force a relatively high *fairness* between all units and fix any major unfairness due to the Stateless Module's random ordering. However, if there are no large loads in the whole system at all, and every unit's power demand is lower than the constant cap (power budget evenly

divided among all units), this module will ignore all previous decisions and set each unit's cap to the constant cap. Such restoration makes sure there is headroom for any unit's incoming tasks. Therefore the Cap Readjusting Module is separated into restoring and readjusting.

Algorithm 5 shows that the restoring part of this module checks if the current power usage of each unit is under a threshold and restores the cap of each unit if so. The boolean flag indicating whether restoration is made is passed to the readjusting module.

As shown in algorithm 6, the readjusting part first checks the boolean flag passed by the restoring part. If all caps are already restored, this part will be skipped at once. If not, then it calculates the remaining unassigned budget and readjusts the caps accordingly.

If there is some remaining unassigned budget, the module will assign the budget to all high-priority units in a way that units with lower caps currently will get allocated more additional budget. Such an allocation decision is made considering two aspects. First, high-priority units with lower caps will need more budget for them to reach peak power compared to those with higher caps. Second, if these units are increasing power in order, units with lower caps will eventually be penalized harder if they are not allocated more additional budget at this time step.

On the other hand, if there is no remaining unassigned budget, the module will readjust the caps of all high-priority units in case units increasing power in order but capped are not penalized equally. The module leaves low-priority units unchanged and equalizes the caps of all high-priority units. Such a decision not only limits all high-priority units to the same power cap, but also ensures that this power cap is no lower than the constant cap, because low-priority units cannot increase power and get allocated with additional budgets before this time step. Therefore, the Cap Readjusting Module ensures the same lower-bound performance as the constant allocation.

3.4.4 Power Dynamics Address Challenges

DPS includes all necessary mechanisms to tackle the two power management challenges identified in Section 3.3.2:

- **Matching the power demands and the budget through application phases.**

When a unit quickly increases power—i.e., a large positive first derivative—it is identified as high priority. By assigning extra power to such units, the Cap Readjusting Module ensures that they always get a fair allocation in anticipation of their future demands. When cluster-wide power demand exceeds budget, the Cap Readjusting Module restores all high-priority units’ power caps to the same level. Since it is impossible for a model-free system to predict the exact *fairness* at any moment, DPS assumes that all high-priority units demand the highest power possible. While it does not guarantee that the maximum *fairness* is achieved, it avoids the situation that a high-priority unit’s power budget is aggressively stripped by another one, resulting in low *satisfaction* for a high-priority unit, and forces a relatively high instantaneous *fairness*.

- **Handling high-frequency power changes.** DPS identifies high-frequency units as high priority. By assuming they are in need of extra power, DPS assures their performance does not suffer, even when it cannot react to the speed of their phase changes. This approach guarantees the same lower-bound performance as constant allocation, where prior work (that does not account for these high-frequency changes) can easily produce worse results than constant allocation (as we will see in the evaluation section).

3.5 Experimental Setup

3.5.1 Hardware Evaluation Platform

We use the Chameleon cloud [Keahey et al., 2020]. Each experiment runs on a Server node and ten client nodes. The client nodes include two clusters. Each node is a dual-socket system running Ubuntu 18.04 (GNU/Linux 5.4) with 2 Intel(R) Xeon(R) Gold 6240 processors, 192 GB of RAM, and hyperthreads. Each socket has 24 cores/48 threads and a 20 MB last-level cache. TurboBoost is turned on and the CPU frequency governor is set in performance mode. The servers support power capping at the granularity of sockets, with a Thermal Dynamic Power (TDP) of 165W per socket. For all the experiments in this paper, a cluster-wise 66.7% power limit is enforced.

3.5.2 Data Analytics and HPC Benchmarks

We test 7 widely-used machine learning applications (written in Apache Spark) from the HiBench Benchmark Suite [Huang et al., 2012], and 8 compute-intensive HPC benchmarks from the NAS Parallel Benchmark Suite (NPB) [Bailey et al., 1995]. We also use 4 micro applications from the HiBench Benchmark Suite that consume much less power. Spark workload specs are listed in Table 3.2 and the NPB workload specs are listed in Table 3.4. Spark workloads are labeled as mid-power if the time portion above 110W is above 10%, and they are labeled as high-power if more than 2/3 of the time power is above 110W. Low-power workloads are executed with 1 Spark executor with 8 cores. The mid-power and high-power workloads are executed with 48 Spark executors with 8 cores, taking up all cores (including virtual cores) on the worker nodes. All NPB workloads are considered as high-power as over 99% of the time power is above 110W. The thread setting for the NPB workloads is the maximum allowed.

To test the power manager’s ability to adapt to a wide range of situations, we create

several different workload profiles by co-executing applications with differing power demands. Spark workloads have distinct power phases while NPB workloads consistently consume high power. Therefore, we experiment with the following three benchmark setups:

- **Spark low utility:** mid-power and high-power paired with low-power workloads, representing total utility below 50%.
- **Spark high utility:** A high-power workload paired with mid-power workloads, with average utility above 50%.
- **Spark NPB:** mid-power and high-power Spark workloads paired with NPB workloads, representing high utility and tight power demands.

In the first group we compare DPS to constant allocation, SLURM’s power management plugin, and an oracle. In the second and third group no low-power workloads are included, and the cluster-wide power demands often exceed the power budget. Meanwhile, the variation in Spark makes the instantaneous power demand at a specific time uncertain. Altogether, implementing an oracle in such cases is extremely difficult. Therefore we only compare DPS to constant allocation and the SLURM plugin in this group.

3.6 Results and Evaluations

We report results for all scenarios described above. We compare a number of power managers based on their harmonic mean performance improvement over constant allocation. For reference, Tables 3.2 and 3.4 report the mean latency of each workload under the constant allocation scheme (each socket gets 110 W power cap). Although experiments with multiple power limits lower than the TDP can provide a more comprehensive evaluation of DPS, multiple runs of each benchmark workload are needed to diminish the performance variance and the benchmark workloads are time-consuming. We repeated at least 10 runs for each workload and spent over 1,000 hours collecting data for experiments with one power limit.

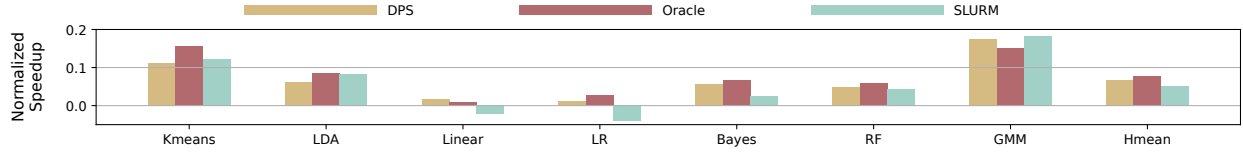


Figure 3.4: Hmean performance gain of the Spark low utility group normalized to under constant 110W cap.

We note that in all cases (and for all power managers) the power caps are respected. We therefore focus the evaluation solely on the difference in performance.

3.6.1 Performance: Spark Low Utility Workloads

When a mid-power workload is paired with a low-power workload, cluster-wide power demands rarely exceed the budget. As shown in Figure 3.4, both DPS and the oracle improve over constant allocation by 5% to 8% on average. The performance improvements differ between workloads, though. DPS provides a maximum performance improvement for GMM at 17.6%. SLURM also improves performance for 5 workloads to a similar extent as DPS and the oracle, except for Linear and LR, which both have high-frequency power phases. LR’s performance is decreased by 4.0% with SLURM. The performance difference between DPS and SLURM mainly results from DPS’s ability to identify and handle these high-frequency changes.

For two workloads, LDA and GMM, the oracle doesn’t provide maximum performance. In fact, the oracle doesn’t provide maximum performance at all times for any workload. The two reasons behind this are first, cluster-wide power demands rarely exceed the budget and DPS and SLURM are able to provide close-to-oracle performance as long as they match power caps to power demands fast enough, and second, the Spark workloads demonstrate such variable performance between different runs under the same execution condition that the average performance of DPS and SLURM may exceed that of the oracle.

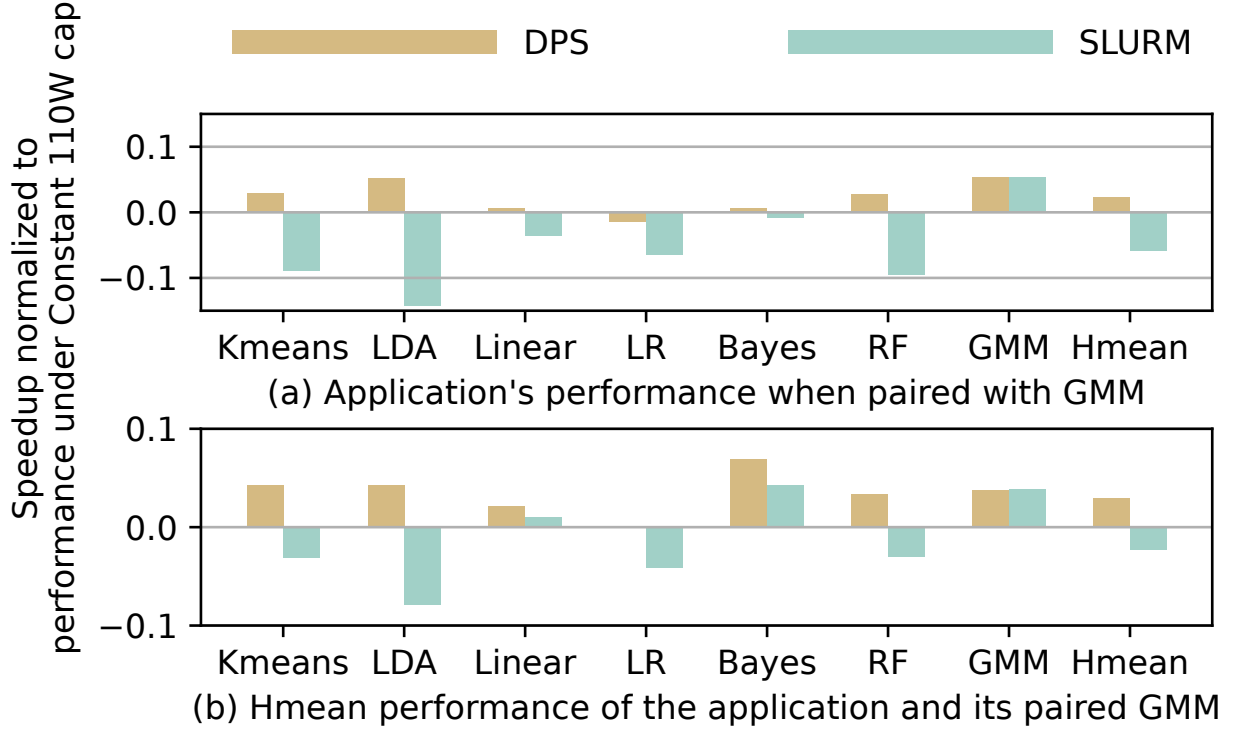


Figure 3.5: Normalized hmean performance gain of the Spark high utility group.

3.6.2 Performance: Spark High Utility Workloads

When pairing mid-power and high-power Spark workloads, the cluster-wide power demands often exceed the power budget. In such cases, the performance reflects not only how fast the power management system reacts to demand changes but how both workloads are penalized when the power budget is exceeded.

Figure 3.5 (a) shows the harmonic mean of each mid-power workload when it is paired with a high-power workload. DPS delivers either the same performance or improvements up to 5.2% compared to the constant allocation. SLURM penalizes all workloads except GMM. For workloads with long power phases, Kmeans, LDA, and RF, SLURM slows them down by from 8.9% to 14.3%. The workloads with high frequencies, Linear and LR, are penalized by up to 7.7%.

Figure 3.5 (b) shows the harmonic mean of the performance of each workload and its

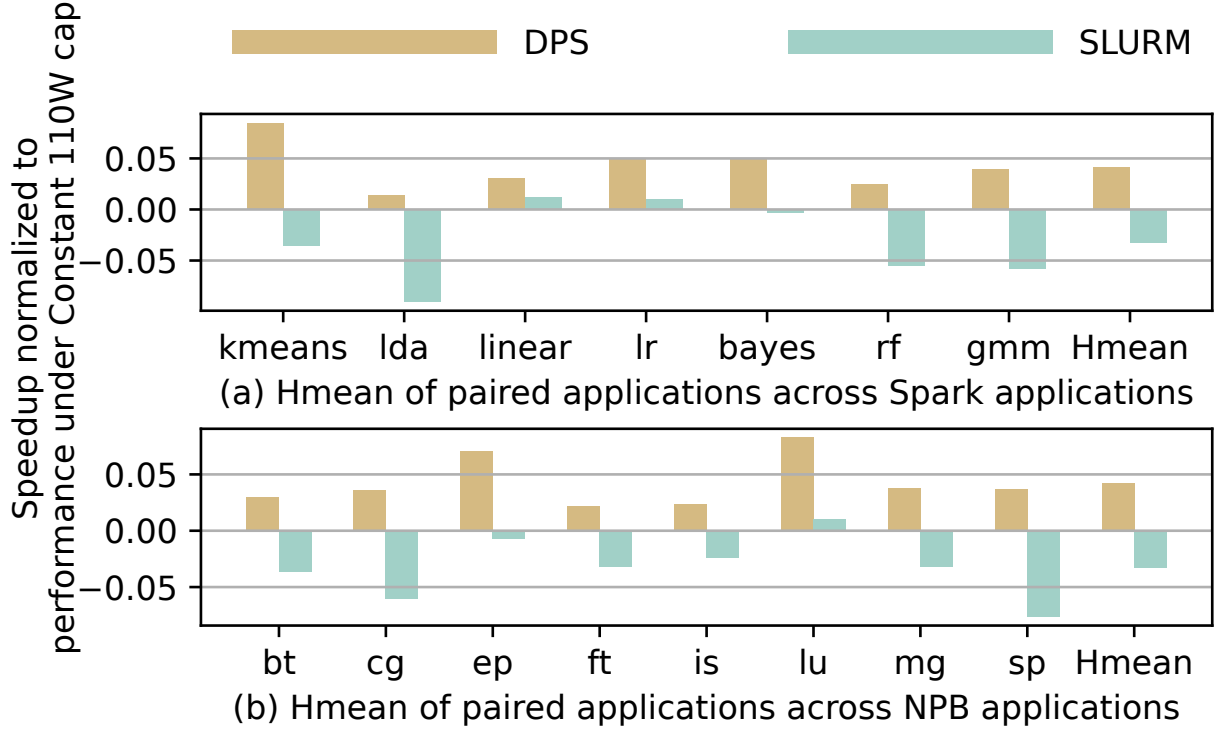


Figure 3.6: Normalized hmean application performance gain of the Spark NPB group.

paired GMM executions. DPS improves the mean performance of both workloads or ensures the lower-bound performance all the time, but the mean performance of SLURM is up to 8.1% below the constant allocation baseline. DPS outperforms SLURM by up to 22.76% at maximum for LDA. While SLURM always prioritizes power allocation to GMM and penalizes the other paired workload, DPS makes sure both workloads are equally penalized. By taking power from GMM and allocating it to the other workloads when necessary, DPS improves performance for both workloads and ensures the same lower-bound performance as constant allocation.

3.6.3 Performance: Spark & NPB Workloads

Spark workloads have phases but the NPB workloads always consume high power. That is to say, when a Spark workload is paired with an NPB workload, they compete for power

budget whenever the Spark workload is not idle. Therefore these performance results reflect how well the managers balance a limited power budget.

The Spark NPB group runs Spark workloads on one cluster and NPB workloads on another, including all exhaustive pairs of mid-power and high-power Spark workloads and the NPB Workloads. Among all the pairs, DPS always outperforms SLURM by a range from 1.7% to 21.3%. Figure 3.6 (a) shows the result grouped by the Spark workloads. For each workload in the figure, the two bars represent the harmonic mean performance gains of the paired workloads managed by DPS and SLURM respectively, normalized to the constant allocation at 110W per socket. While performance differs between different workloads, DPS improves the performance of all the workloads. On the contrary, SLURM decreases the performance of all workloads but Linear and LR. Since the data in the figure represents the harmonic mean between the Spark workload and NPB workloads, it shows that the performance improvement of the NPB workloads and the performance decline of the Spark workloads under SLURM leads to an overall performance drop. For example, for the pair of GMM and EP, SLURM results in a 30.3% performance decline for GMM and 27.8% performance increase for EP, altogether resulting a harmonic mean 9.8% performance decline. DPS, on the other hand, yields 3.9% and 5.2% performance increases for GMM and EP respectively, and a harmonic mean 4.6% performance improvement. DPS outperforms SLURM on Kmeans by a max 12.4% and a min 1.9% on Linear, and an overall mean 8.0%.

Figure 3.6 (b) shows the result grouped by the NPB workloads. Similar to grouping by Spark workloads, DPS improves performance for all NPB workloads, and SLURM decreases performance for all workloads except LU. The performance speedup also varies among NPB workloads. For short-duration NPB workloads, such as FT and EP, when paired with long-duration Spark workloads, such as Kmeans and GMM, multiple runs are in need to match one run of the Spark workload. Although NPB workloads consistently demand high power, the short period between workloads makes each execution look like a power phase. Given a

long range of time, the short-duration NPB workloads do not always demand high power. Therefore SLURM delivers better performance with short-duration NPB workloads. In general, DPS outperforms SLURM by a minimum of 4.9% on IS and a maximum 12.2% on SP.

3.6.4 Fairness Analysis

For the Spark Low Utility workloads, DPS, SLURM, and the oracle achieve similar *fairness*, close to 1. This is because there is generally sufficient power to cover all power demands. Yet for the Spark High Utility workloads, DPS achieves a mean 0.97 *fairness* while SLURM achieves only 0.75 mean *fairness*. Figure 3.7 shows the distribution of the workload *fairness* under DPS and SLURM. More specifically for each workload, DPS gets a *fairness* from 10.9% to 43.6% higher than that of SLURM. For the Spark & NPB workloads, where *fairness* has the most influence on the harmonic mean performance, DPS and SLURM achieve a mean 0.96% and 0.71% *fairness* respectively. Workload-wise, DPS always gets a higher *fairness* than SLURM, from 0.2% to 77.5%. We observe a general positive correlation between *fairness* and harmonic mean performance.

3.6.5 Overhead Analysis

All above reported performance data is real measurements on systems running either DPS or SLURM. The performance data includes the overhead of the power management systems themselves. DPS and SLURM are implemented using the same Internet communication protocol, while Constant Allocation and Oracle do not have overhead since they do not change the power caps.

As discussed in section 3.4, DPS is designed with multiple modules among which is a stateless module. The stateless module has theoretically the same overhead as SLURM. Apart from that, the overhead of the other three modules also scales by a constant cost. As

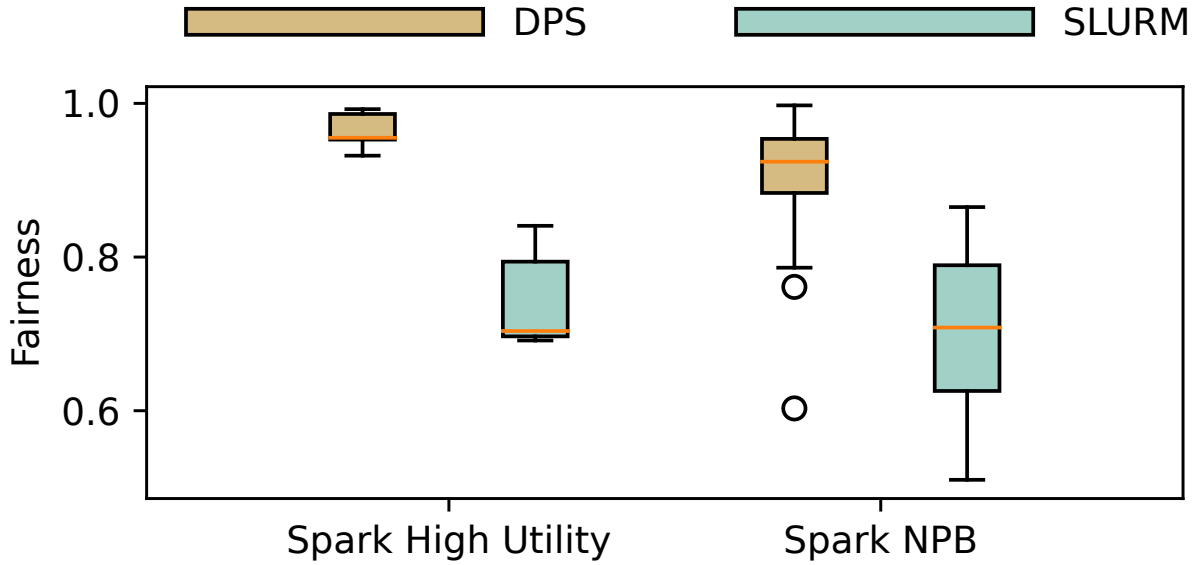


Figure 3.7: Fairness of the two High-utility workload groups.

DPS keeps a short range of estimated power history of each socket, default 20 time steps, the power history can easily fit in the last-level cache even scaled to tens of thousands of nodes, taking up several megabytes. Therefore IO is not bottle-necking the turnaround time of DPS and we observe that it incurs less than 0.5% average CPU usage on the controller node with 10 client nodes. In fact, we observe that communication using the BSD socket interface takes up most of the turnaround time. The network latency is of tens of microseconds. Scaling to 1,000 nodes would only incur a several millisecond latency, which is little compared to a one-second decision loop time. Since only 3 bytes are exchanged per request with each node, scaling to even 1M nodes, requiring a network traffic size of 3MB, would put a little burden on a network bandwidth in GB/s. Such overhead is shared among all centralized power management systems and consistent with SLURM and prior work [Zhang and Hoffmann, 2019, Srivastava et al., 2023]. Therefore we claim that DPS has negligibly more operating overhead than SLURM, a stateless system, and the controller could handle tens of thousands of nodes with no bottleneck in computation or network.

Different from model-based systems, DPS is ready to operate once deployed on a cluster.

It needs at most the time of the range of estimated power history to make desired decisions, which is defaulted at 20 seconds. SLURM, on the other hand, sets power caps based on instantaneous power, so it starts to operate functionally as soon as deployed on a cluster. However, considering 20 seconds are negligible compared to the lifetime of a cluster, we claim that DPS has negligibly more deployment overhead than SLURM.

3.6.6 Summary of Key Results

DPS ensures the same lower-bound performance as the Constant Allocation. In low-utility scenarios where cluster-wide power demands rarely exceed the power budget, DPS yields similar performance to SLURM and an oracle. In high-utility scenarios where cluster-wide power demands often, or even always exceed the power budget, DPS outperforms SLURM from 1.7% to 21.3%. Other work on model-based approaches reports a range of 14% to 22% improvement compared to SLURM [Zhang and Hoffmann, 2019]. DPS yields competitive performance improvement without requiring a model.

3.7 Conclusion

This paper presents DPS, a model-free stateful power management system that improves application performance in an overprovisioned system. DPS contributes a new methodology for designing a power management system by analyzing *power dynamics*. It explores a promising middle ground between prior work on model-based systems, which require an expensive and error-prone model but yields very high performance, and model-free stateless systems, which make decisions based solely on instantaneous power usage. We hope this inspires people to further explore the use of *power dynamics* and the temporal relationships residing in power usage data, and achieve performance even closer to the model-based systems.

Algorithm 6: Cap Readjusting Module: Readjust

```
1 Function readjust(restore_flag):
2   global power, cap, set_flag, priority
3   if restore_flag then return
4
5   avail_budget  $\leftarrow$  total_budget - sum(cap)
6   if avail_budget > 0 then
7     // Assign all the rest budge if any left
8     high_priority_ratios  $\leftarrow$  Dictionary()
9     for u  $\in$  units do
10      if priority[u] then
11        high_priority_ratios[u]  $\leftarrow$  cap[u]
12      budget_high  $\leftarrow$  sum(high_priority_ratios.values())
13      for u  $\in$  priority[u].keys() do
14        high_priority_ratios[u]  $\leftarrow$  budget_high/high_priority_ratios[u]
15      total  $\leftarrow$  sum(high_priority_ratios.values())
16      for u  $\in$  priority[u].keys() do
17        cap[u]  $\leftarrow$  min(spec_max_cap, avail_budget * high_priority_ratios[u]/total)
18        set_flag[u]  $\leftarrow$  True
19
20   else
21     // Readjust all high-priority units
22     budget_high  $\leftarrow$  0
23     count_high  $\leftarrow$  0
24     for u  $\in$  units do
25      if priority[u] then
26        budget_high  $\leftarrow$  budget_high + cap[u]
27        count_high  $\leftarrow$  count_high + 1
28     readjusted_cap  $\leftarrow$  budget_high/count_high
29     for u  $\in$  units do
30      if priority[u] then
31        cap[u]  $\leftarrow$  readjusted_cap
32        set_flag[u]  $\leftarrow$  True
```

Table 3.2: Spark benchmark workloads.

Workload	Data size	Duration	Power Type	Above 110W
Wordcount	3.1 GB	44.36s	low-power	0.18%
Sort	313.5 MB	38.48s	low-power	0.10%
Terasort	3.0 GB	54.53s	low-power	0.07%
Repartition	3.0 GB	44.92s	low-power	0.20%
Kmeans	224.4 GB	1467.08s	mid-power	47.58%
LDA	4.1 GB	1254.12s	mid-power	51.54%
Linear	745.1 GB	928.36s	mid-power	14.53%
LR	52.2 GB	499.37s	mid-power	16.69%
Bayes	70.1 GB	342.18s	mid-power	33.20%
RF	32.8 GB	415.71s	mid-power	35.78%
GMM	8.6 GB	2432.43s	high-power	68.96%

Table 3.3: Spark benchmark computing Resources.

Power Type	# Executors	Cores per executor
low-power	1	8
mid-power	48	8
high-power	48	8

Table 3.4: NAS Parallel Benchmark applications.

Workload	Data size	Duration	Threads
BT (Block Tri-diagonal solver)	247.1 GB	3509.29s	144
CG (Conjugate Gradient)	21.8 GB	1839.00s	128
EP (Embarrassingly Parallel)	4 TB	6019.07s	192
FT (Fourier Transform)	400.0 GB	152.83s	128
IS (Integer Sort)	128.0 GB	416.80s	128
LU (Lower-Upper Gauss-Seidel)	296.5 GB	1895.89s	192
MG (Multi-Grid communication)	400.0 GB	143.82s	128
SP (Scalar Penta-diagonal solver)	494.2 GB	3563.23s	144

CHAPTER 4

OBSERVATION, NOT PREDICTION: CONVERSATION-LEVEL DISAGGREGATED SCHEDULING FOR AGENTIC SERVING

4.1 Introduction

Traditional large language model (LLM) serving is built around a simple contract: a request arrives, the model prefills its prompt and decodes a response, and the request departs [Vaswani et al., 2017]. Agentic workloads break this contract. A user task is no longer resolved by a single model invocation but by a sequence of dependent inference calls interleaved with tool use, environmental feedback, and accumulating state [Yao et al., 2023]. The served object is therefore not a request but a *stateful, multi-turn program*, which, moreover, exhibits a characteristic temporal asymmetry: its first turn encodes a long task description (e.g., instructions, repository state, retrieved context) in a single compute-bound prefill, while later turns typically append only short tool outputs and observations, dominated by memory-bound decoding over a key-value (KV) cache that only grows. A conversation is therefore, typically, structured as one heavy prefill followed by a long, light, memory-bound tail.

This shift is not merely quantitative. Current generic LLM serving systems typically take the request (equivalently, a single prefill–decode turn) as the atomic unit of scheduling. For single-shot inference this approach is harmless, as the unit the system schedules coincides with the unit the user values. Agentic workloads break this coincidence. What the user values is the conversation’s final, externally meaningful output, yet most intermediate turns emit only tool calls that are never read by the user. The result is a *scheduling–value mismatch*: the system optimizes per-turn latency while the user waits on a conversation-level response. Consequently, existing serving systems that optimize each turn in isolation optimize the wrong objective. To name the value the user actually cares about, we introduce **time-**

to-first-effective-token (TTFET): the time until the first token of genuine, user-visible progress, as opposed to the internal tokens that drive tool calls. TTFET is thus a property of a conversation, not of any turn

Existing multi-turn serving systems do recognize that later turns differ from the first . Building on prefill–decode disaggregation, which separates the compute-bound prefill from latency-sensitive decoding onto distinct workers [Zhong et al., 2024], these systems add append-prefill routing, KV-cache retention, and tool-aware scheduling to avoid recomputing shared state across turns [Li et al., 2025, Abhyankar et al., 2024, Luo et al., 2025]. Yet these designs share a single design principle: the decision unit remains the turn . For each turn, such a system decides whether to disaggregate it (i.e., whether to pay a remote prefill and a KV transfer or process it locally) and reaches that decision by profiling or predicting the turn’s cost [He et al., 2026, Li et al., 2026].

In this work, we first argue that this per-turn scheduling decision is *structurally brittle*. The cost that drives the decision cannot be known in advance: a turn’s prefill is fixed by its input length and can be read directly, but other key behavior (e.g., how many tokens it will decode, whether it will call a tool, and how much it will enlarge the KV cache) is unobservable at the moment the scheduler must act. Yet such decode-side quantities are what determine whether disaggregating the turn pays off, so per-turn placement must predict them . Any predictor, however well-tuned, will eventually misjudge a turn and route it incorrectly (we characterize this prefill/decode predictability gap empirically in Section 4.3). We argue this prediction is not a requirement of the workload but of the scheduling unit: partitioning a conversation into independently scheduled turns discards precisely the structure that would have made placement deterministic. *Thus, in prior work, the unit, not the predictor, is the source of the fragility.*

Key Insight. We schedule the conversation, not the turn. At turn granularity, agentic serving is irregular: decode lengths, tool calls, append-prefill sizes, and KV growth are difficult

to infer when placement decisions are made. At conversation granularity, *this irregularity collapses into a stable two-phase structure*: an initial compute-bound prefill that materializes task state, followed by a memory-bound trajectory consisting of the first decode and all subsequent append-prefill/decode turns. Later turns may contain append-prefill work, but their input is small relative to the accumulated KV cache, so execution is dominated by KV movement and attention over existing state. This restores the original prefill–decode abstraction but at conversation-level granularity: one compute-bound phase, one memory-bound phase, and a KV transfer at the boundary.

Two consequences follow. First, scheduling need not predict turn-level outcomes such as future decode lengths or tool behavior; it can instead rely on the same kind of observable phase structure that classical serving systems already expose. Second, heterogeneous serving follows by construction: under heterogeneous cluster environments, the same phase split maps compute-intensive initial prefills to high-throughput GPUs and the long memory-bound tail to devices with sufficient memory bandwidth and capacity.

ConServe. We instantiate this principle in ConServe¹, a conversation-level disaggregation scheduler for agentic serving. ConServe mirrors the original prefill–decode abstraction at the conversation level: it routes the first-turn prefill as the compute-bound phase, transfers the resulting KV cache exactly once, and then runs the first-turn decode and every subsequent append-prefill/decode turn as a single memory-bound phase. Under ConServe, a conversation is therefore pinned for its entire memory-bound tail, avoiding repeated KV movement across turns. This paradigm shows that *placement should be reactive rather than predictive*. New conversations are assigned using observable quantities: input-token counts for the initial prefill and decoder KV-cache utilization for the memory-bound tail. ConServe overprovisions decode replicas to absorb variation in conversation length; when a prefiller is saturated, or when decoders approach saturation under bursts of long conversations, the

1. So named as to reflect two commitments: ConServe *serves* stateful, multi-turn agentic *conversations* and *conserves* energy by spending it only on the phase that requires it.

scheduler routes the *next* conversation elsewhere rather than migrating one already placed.

We evaluate ConServe on representative agentic workloads [Jimenez et al., 2024], using a Qwen3-0.6B model backbone and a NVIDIA A40 GPU cluster, against collocated, fully disaggregated, and per-turn prediction-based baselines. Compared to the per-turn baseline [He et al., 2026], ConServe reduces p95 TTFET by 51.08% and improves energy efficiency by 7.51% while maintaining similar last-turn time-between-tokens (TBT) and preserving service-level objectives (SLOs). It does so with a deliberately minimal policy: one disaggregation per conversation and no learned cost model. On heterogeneous hardware, ConServe preserves these latency results while improving energy efficiency by a further 22.75% over its homogeneous configuration. These gains show that the relevant abstraction is the conversation, not the turn. We therefore present conversation-level scheduling as a principled paradigm for agentic serving, and ConServe as a minimal instantiation of this principle. We make the following contributions:

- **Conversation-level scheduling.** We identify a *scheduling-value mismatch* in agentic serving: systems optimize per-turn latency, while users wait for conversation-level outcomes. We argue that the conversation is therefore the correct unit of scheduling, and introduce *time-to-first-effective-token* (TTFET) and *last-turn time-between-tokens* (TBT) as metrics that capture this objective. Raising the unit to the conversation replaces turn-level prediction with directly observable state, eliminating the source of brittleness in prior multi-turn schedulers.
- **ConServe.** We introduce a disaggregation scheduler that routes first-turn prefills to high-power GPUs, transfers KV state exactly once, and executes a conversation’s entire tail locally on a separate (possibly low-power) decoder devices. Committing to one placement per conversation from observable signals allows ConServe to work without a learned cost model.
- **Empirical demonstration of structural brittleness.** We show that per-turn

scheduling is brittle in a structural sense: under a per-turn prediction baseline, SLO violations and energy efficiency degrade linearly in prediction error rate, while ConServe’s conversation-level placement holds constant by construction.

- **Heterogeneity and energy as corollaries.** Because placement reads only observable states, tier-aware allocation and conversation-level energy optimization follow without additional machinery, and both are largely unaddressed by prior multi-turn systems.

4.2 Background

4.2.1 LLM Serving Techniques

Transformer-based language models [Vaswani et al., 2017] generate text autoregressively in two stages [Radford et al., 2018, 2019]. Prefill processes the input prompt in a single forward pass and produces a key-value (KV) cache for every input token. Decode then generates each subsequent token by reading the full KV cache and appending one new entry. Prefill is consequently compute-bound; decode is memory-bound, gated by KV cache bandwidth at every step [Pope et al., 2023, Patel et al., 2024]. Efficient management of this growing KV cache is critical for serving throughput [Gao et al., 2024].

A few techniques have been proposed and adopted in practice to address these characteristics and improve serving quality for user-facing LLM inference requests. Prefix caching [Zheng et al., 2024] reuses the KV cache across requests sharing a common input prefix, eliminating redundant prefill computation. Dynamic batching [Yu et al., 2022] batches forward passes from different requests at each iteration so finished requests can leave and new requests can join continuously, raising GPU utilization. Its downside is that a long prefill batched into ongoing decode iterations stalls the decoders for the prefill’s duration; chunked prefill [Agrawal et al., 2024b] mitigates this by splitting long prefills into smaller chunks

that interleave with decode, bounding the per-step stall. Prefill-decode (PD) disaggregation [Zhong et al., 2024, Patel et al., 2024] eliminates the problem entirely by assigning prefill and decode to separate replicas, exploiting the compute-bound vs memory-bound asymmetry to raise aggregate GPU utilization at the cost of transferring KV state between replicas.

4.2.2 *Multi-turn Agentic Conversations*

Agentic LLM applications [Chase, 2022, Richards, 2023] follow a multi-turn ReAct execution loop, in which the model reasons about a user request prompt, emits a structured tool call (e.g., an API invocation, code execution, or database query), suspends until the tool returns, and resumes generation with the tool result appended to the context [Yao et al., 2023, Schick et al., 2023, Shinn et al., 2023, Qin et al., 2024]. We refer to the full sequence of turns required to complete a single user request as a *conversation*. A conversation begins with a detailed, initial prompt that establishes the task and environment context, followed by a variable number of turns, each consisting of an LLM generation step and a subsequent tool invocation whose result is fed back as additional input [Liu et al., 2024a]. To improve context efficiency, modern agentic systems frequently restrict tool-call outputs to task-relevant information, avoiding the inclusion of complete files, logs, or other large artifacts in the prompt [Anthropic, 2026].

Unfortunately, neither collocated nor fully disaggregated prefill-decode architectures are well matched to this workload structure. In a collocated deployment, chunked prefill [Agrawal et al., 2024b] splits the long prefill into smaller chunks interleaved with decode steps, but a turn-1 prompt spanning thousands of tokens still occupies the GPU for many chunk iterations, delaying collocated decodes throughout the chunking window. Full prefill-decode disaggregation eliminates this interference by routing prefill and decode to dedicated GPUs [Zhong et al., 2024, Patel et al., 2024]. In a multi-turn setting, however, this design routes every turn through the prefill node, disregarding a key structural property of agentic work-

loads: turn 2+ appends are short incremental prefills that could run directly on the decoder holding the cached prefix, with minimal contention (Figure 4.5). Full disaggregation forfeits this optimization opportunity in turn 2+ requests.

Two concurrent systems route incremental prefills in multi-turn disaggregated serving, and both retain the turn as the decision unit. AMPD [He et al., 2026] chooses, for each Turn 2+ prefill, between local execution and remote execution on a prefill node using real-time queue states and an offline cost model; remote execution reads the history KV from the decoder and writes the new KV back, a bidirectional transfer that departs from the one-way producer-consumer contract of standard disaggregation. PPD [Li et al., 2026] makes the same per-turn choice through an offline lookup table indexed by context length, input-output ratio, and load. Keeping the turn as the unit has two consequences. First, each decision conditions on quantities unavailable when it is made: AMPD’s cost model omits the collocated-decode interference that dominates local execution cost, and PPD’s table is indexed by a request’s output length, unknown at arrival. Second, both optimize per-turn TTFT and TBT, and neither measures TTFET or last-turn TBT, the conversation-level metrics that determine user-perceived performance (§4.1). Both consequences trace to scheduling the turn rather than the conversation. The case for per-turn routing is also narrowest where agentic workloads sit: PPD’s dynamic routing is motivated by Turn 2+ requests with heavy prefills, but agentic appends are uniformly short (§4.3), the regime in which local execution already dominates and the per-turn decision collapses to a fixed local policy. ConServe adopts that policy at conversation granularity, conditioning only on observable state and optimizing the metrics users experience.

4.2.3 *Serving with Heterogeneous GPUs*

Several recent systems serve LLMs on heterogeneous GPU clusters. Some formulate GPU allocation as constrained optimization over cost or throughput, assigning GPU types to

model replicas or partitions [Griggs et al., 2024, Mei et al., 2025, Jiang et al., 2024]. Others route the prefill and decode phases onto distinct GPU tiers, matching the compute-bound prefill and memory-bound decode to hardware suited to each [Patel et al., 2024, Jiang et al., 2025b,a, Shi et al., 2024]. Although these approaches differ in optimization formulation and partitioning strategy, they share a common insight: LLM inference places non-uniform demands on compute, memory capacity, and memory bandwidth, making it profitable to map workload to GPU capabilities rather than deploying a single homogeneous GPU type. All of this work, however, targets single-turn inference. At conversation granularity, agentic workloads exhibit a stable two-phase structure, a compute-bound first-turn prefill followed by a memory-bound trajectory, which remains unexploited.

4.3 Characterization of Agentic Workloads

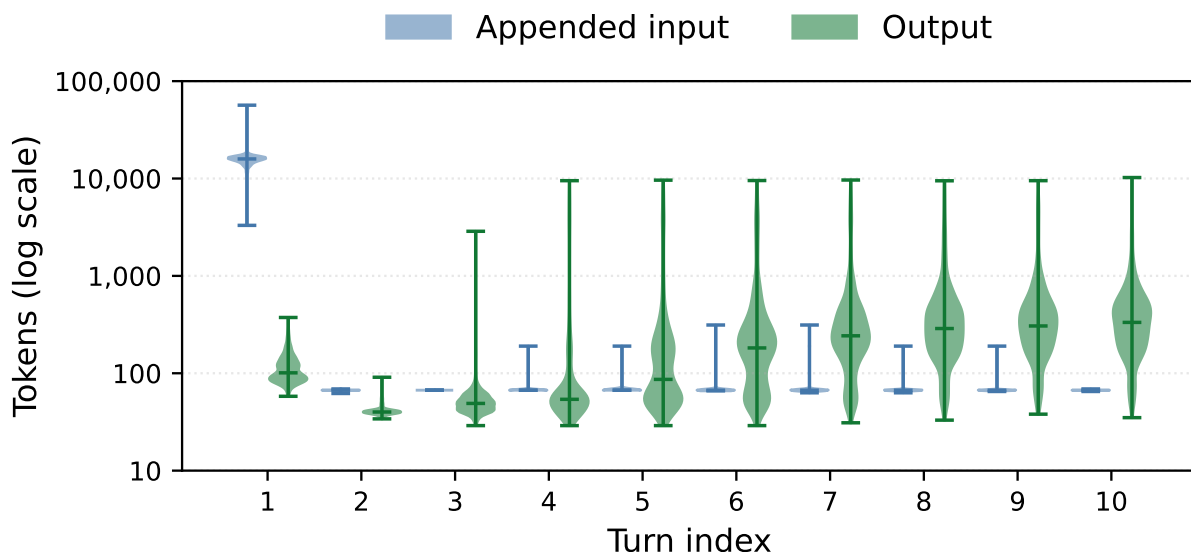


Figure 4.1: Input/Output token distribution of the first 10 turns from agentic traces. For the appended input, turn-1 is the input prompt, turn-2+ is the tool call response.

While LLM and agentic workloads share similar computational phases, agentic workloads behave differently due to their underlying characteristics, as shown empirically in Figure

4.1. The figure depicts the input and output token distributions of agentic traces generated from SWE-bench_bm25_13K with swe-agent, using Qwen3-Coder-30B-A3B-Instruct as the trace-generation model. Agentic conversations begin with an initial input of tens of thousands of tokens, whereas prompts appended in later turns are far smaller, on the order of hundreds. Output token counts, in contrast, are unpredictable and exhibit large variance. To build the quantitative foundation to understand the impact of serving techniques on agentic conversations, we profile agentic workloads across computational phases, inputs tokens, context lengths, and GPU power budgets and analyze the impact of each variable on agentic serving.

Profiling Setup. Experiments are conducted on NVIDIA A40 GPUs running Qwen3-0.6B in bf16 precision. This model size is chosen to leave sufficient GPU memory headroom to accommodate the large and growing KV caches of multi-turn agentic conversations, making the experimental setup representative of realistic agentic serving conditions. For heterogeneous GPU experiments, we power cap GPUs at 2/3 of their Thermal Dynamic Power (TDP) to represent lower-power GPUs that has less compute capacity but similar memory hardware. While results are model- and hardware-specific, the general qualitative observations hold broadly across model sizes and GPU generations as the underlying computational properties are architectural rather than configuration-specific.

4.3.1 Prefill Phase

We analyze the compute-bound prefill phase under multi-turn agentic conversations. We characterize each turn individually and use Time-to-first-token (TTFT) – latency from request arrival to the first generated token – for comparison. Our results show a clear picture: (1) first turn prefill latency is predictably expensive, but (2) caching and reusing the KV cache could substantially improve prefill latency at later turns, and (3) KV cache transfer between model replicas is relatively inexpensive in agentic scenarios. These insights motivate

the design of ConServe in §4.4.

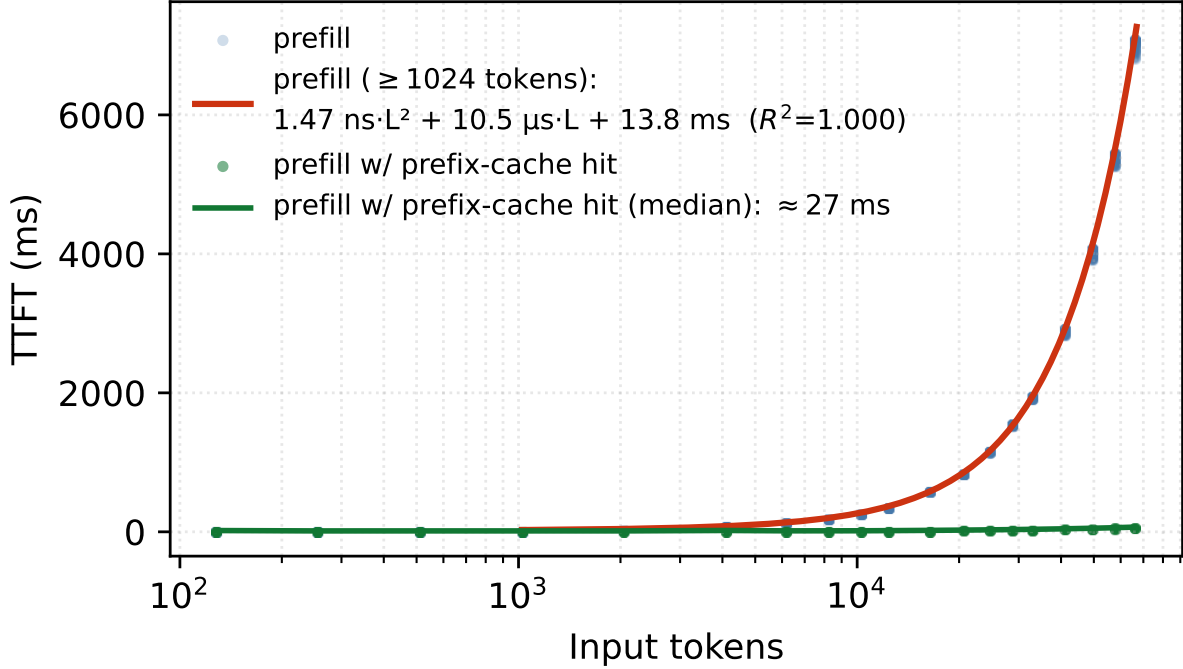


Figure 4.2: Characterization of TTFT under large inputs with and without caching. Uncached agentic prompts ($\geq 10^4$) incur quadratic latency overheads but remain predictable as shown in the label ($R^2 = 1.0$).

Initial prefills are expensive but predictable.. We profile prefill latency across a range of input lengths; results are shown in Figure 4.2. TTFT is nearly constant for short prompts, but grows quadratically once attention becomes the dominant bottleneck. Prior work often models prefill latency as linear [Databricks, 2023], which is accurate when linear projection layers dominate the quadratic attention term. First turn agentic prompts, however, can reach tens of thousands of tokens, where attention is no longer negligible and a quadratic fit is substantially more accurate. In both regimes, TTFT is determined primarily by input length, allowing prefill latency to be estimated accurately from the input-token count alone. ConServe uses this observation to route incoming conversations away from saturated prefill workers before they incur queueing delay.

Prefix caching reduces TTFT to near-constant in multi-turn agentic workloads.

Figure 4.2 includes the latency of prefill stage with prefix-caching across a range of input tokens. As shown, prefix caching reduces TTFT to near-constant, reducing latency by two orders of magnitude under long inputs. Since relatively small prompts are appended to the whole history context after the first turn, prefix caching prior prompts and only calculating new KV cache entries for the appended tokens would substantially speed up the prefill stage at later turns.

KV cache transfer between GPUs is marginal in agentic workloads. Prefill-decode disaggregation – *i.e.*, when prefill and decode phases occur on separate model replicas – incurs additional KV cache transfer between the prefill and decode GPU. Figure 4.3 depicts the latency overhead of KV-transfer and its relative impact on TTFT across a range of input tokens. While KV-transfer contributes significantly to TTFT at low input tokens ($> 20\%$), it scales linearly with respect to input tokens and its impact is thus overshadowed by quadratic prefill latency at long inputs, precisely the operating domain of agentic workloads. Moreover, if no disaggregation happens for the rest of the turns given huge speedup by prefix caching, the KV-transfer overhead becomes negligible. ConServe relies on this insight when scheduling the first turn prefill stage for agentic requests.

4.3.2 Decode Phase

Output tokens are generated autoregressively across many iterations throughout the decode stage. We conclude that (1) agentic workloads are memory bound, (2) prefix-caching reduces collocation overheads, and (3) decoding latency is unpredictable. Our conclusions inform the the design of ConServe in §4.4.

Agentic workloads are memory-bound. We profile the average TBT across a range of batch sizes and KV lengths. As shown in Figure 4.4, mean TBT is relatively stable in low memory configurations but increases substantially under high batch sizes or long contexts

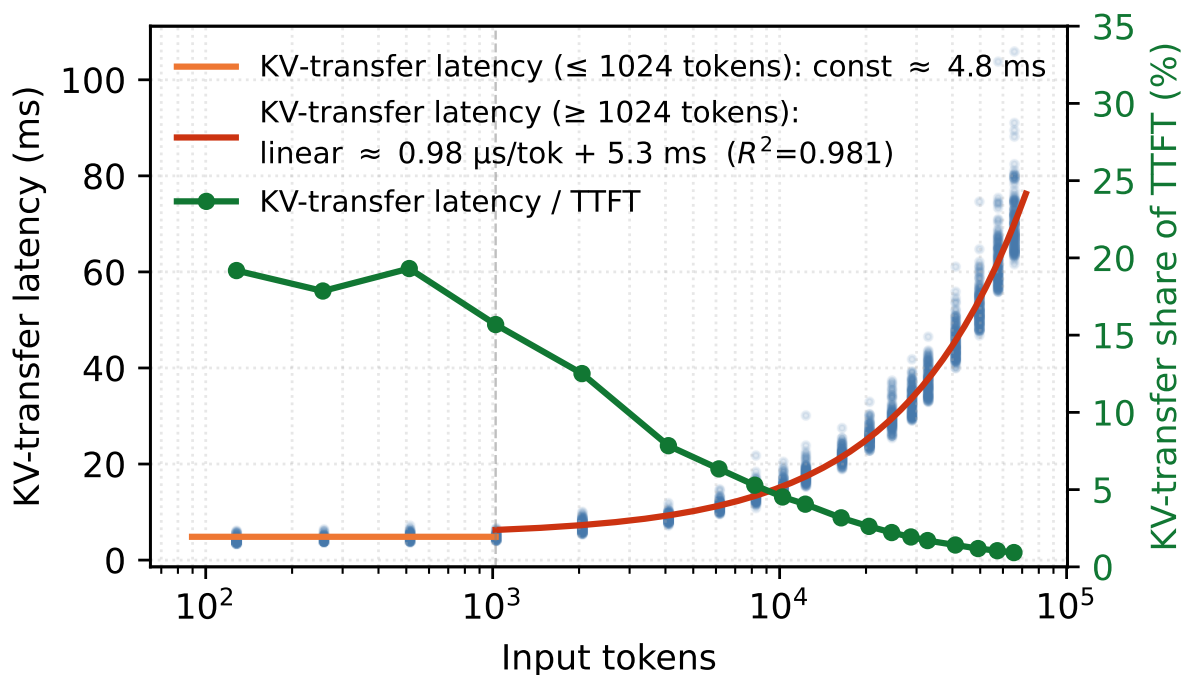


Figure 4.3: KV-transfer overhead is constant at short input lengths (< 1024) and scales linearly at long inputs (≥ 1024). KV-transfer overhead between GPUs is significant when input is short but is dominated by Prefill latency in agentic workloads where inputs are tens of thousands of tokens long.

lengths where memory bandwidth is saturated. Given the long context length of agentic tasks, these workloads are memory-bound during the decode stage.

Prefix-caching changes what impacts collocation overheads. We conduct a series of experiments to understand the impact of collocating prefill and decode stages on a per iteration basis. In the first experiment, we begin a number of decode tasks and then introduce a prefill request after 58 – 92 decode iterations and measure the collocated iterations latency. Figure 4.5.a depicts these results across a range of initial decode tasks (batch size) and prefill size requests. In our second experiment, we maintain a consistent batch size and introduce the prefill request after a range of iterations. Figure 4.5.b depicts these results across a range of prefill size requests. Both figures depict results with and without prefix-cache hits. As shown, prefix-caching significantly improves results by approximately one order of magnitude

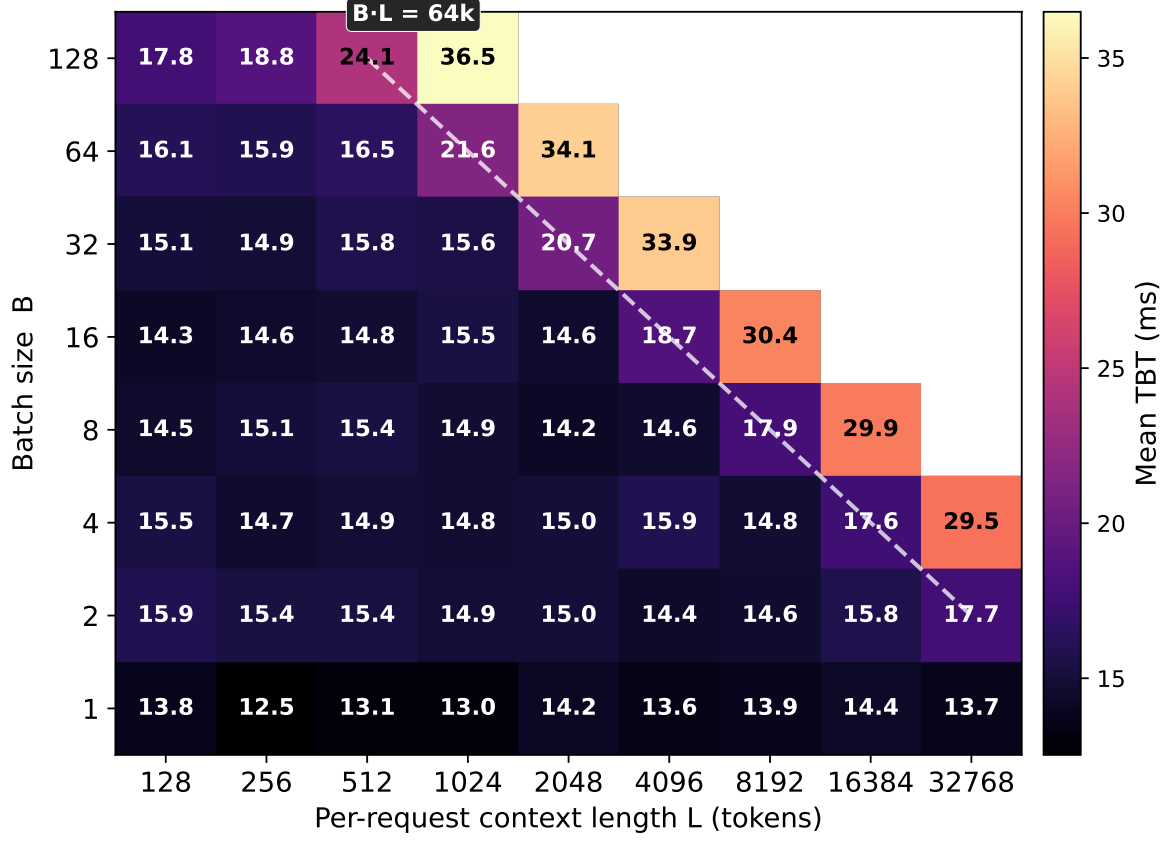


Figure 4.4: Heat map of mean time between tokens (TBT) across a range of batch sizes and context lengths. The dashed white line highlights the boundary between unsaturated (lower left) and saturated (upper right) memory bandwidth regions. Given the long context length of agentic tasks, agentic workloads lay in the saturated memory bandwidth domain.

in both experiments. However, when the decoding context gets larger to where memory cost exceeds to prefill compute cost, the iteration latency is largely affected by the decoding context length. At 262,144 active kv-cache tokens, the iteration latency is determined by the decoding instead of the collocated prefill. Hence, any prediction of such collocated iteration is invalid if the active kv-cache usage is not taken into account.

Decoding latency is highly variable. Our experimental results indicate that decoding latency is highly variable, and unlike the prefill stage, scheduling decisions should not rely on estimating decoding latency. We record the TBT for 64 different, batched prompts across 32 runs and report the iteration-level results in Figure 4.6. While the mean TBT generally

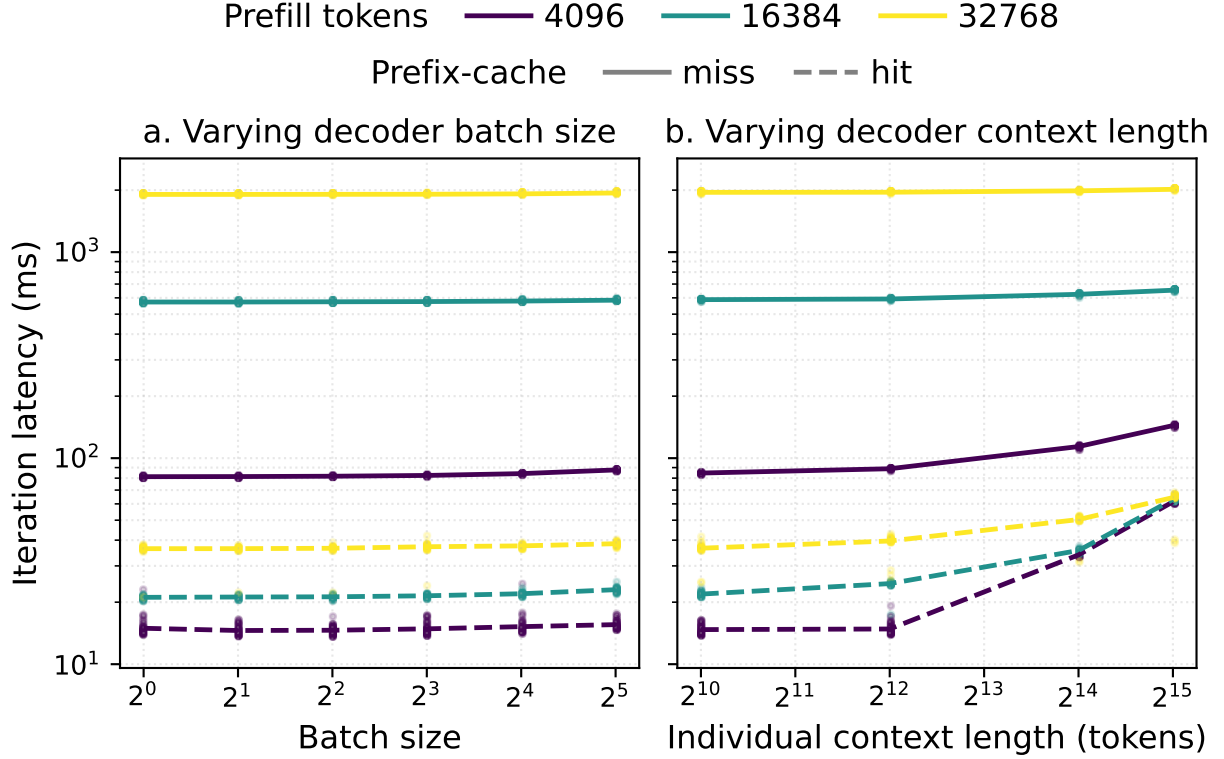


Figure 4.5: Iteration latency of collocated prefill and decode stages. (a) Latency of a prefill request arriving after 58 – 92 decode iterations across a range decoder batch sizes. (b) Latency of a prefill request arriving after a range of decoding iterations with a fixed batch size (8). Prefix-caching significantly improves collocation overhead.

increases as the context length and number of iterations increase, individual TBT remains unpredictable. Moreover, as shown in Figure 4.5.b, iteration latency under collocation and prefix-caching depends on the prefill input size and the context length of running decode tasks. Finally, the total number of output tokens is input-dependent and cannot be known apriori (Figure 4.1). Overall, while decoding latency is correlated with memory bandwidth utilization, we find that accurately estimating the per iteration or end-to-end decoding latency is a challenging task that depends on many complex or unknown factors. Hence, ConServe’s design opts to not rely on such estimates for scheduling purposes.

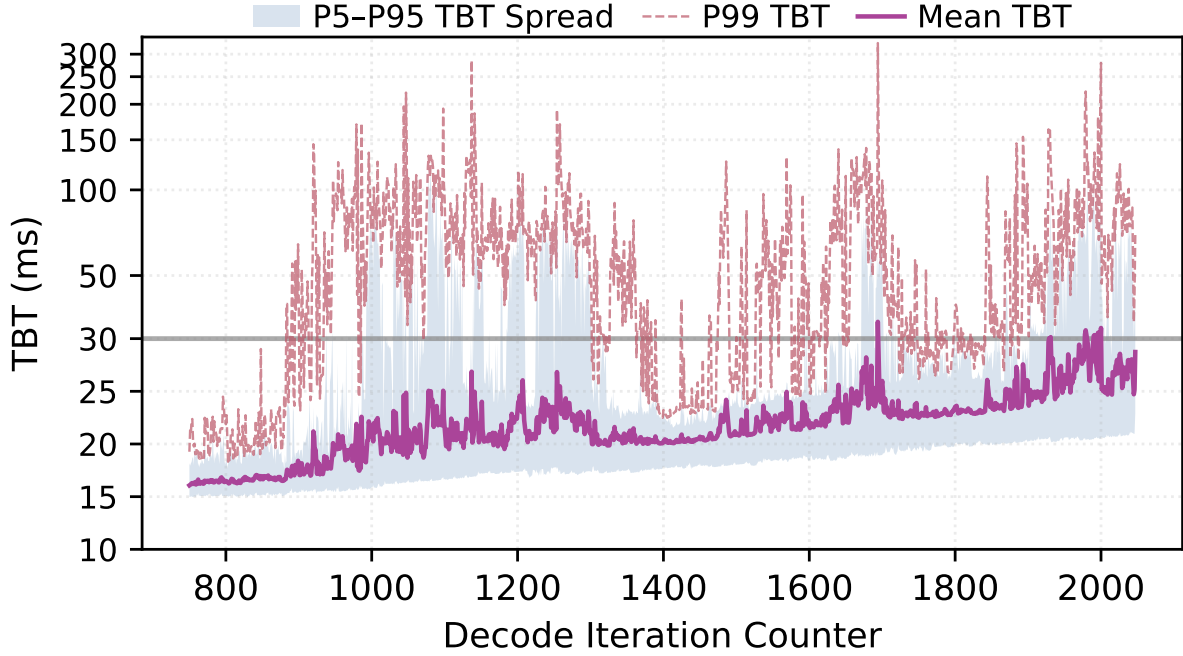


Figure 4.6: High variance in TBT throughout a long decode

4.3.3 Heterogeneous Serving

We vary the GPU power caps to simulate the impact of heterogeneous agentic serving on the prefill and decode phase. We find that prefill and decode are impacted significantly and marginally, respectively. These findings further motivate ConServe’s design choices in §4.4.

Power capping impacts the prefill stage heavily but has a marginal effect on prefix caching. We repeat the experiments in Figure 4.2 on power-capped GPUs and show the difference in performance in Figure 4.7. Since prefill is compute-bound at high input token ranges, power-capping significantly increases TTFT. On the other hand, prefill with prefix-caching remains relatively stable irrespective of input size. This insight harmonizes well with ConServe’s heterogeneous mapping: turn 1 prefill stages are scheduled on the best available GPUs, whereas later prefill stages with prefix-cache hits can occur on less powerful decode GPUs with marginal performance penalties.

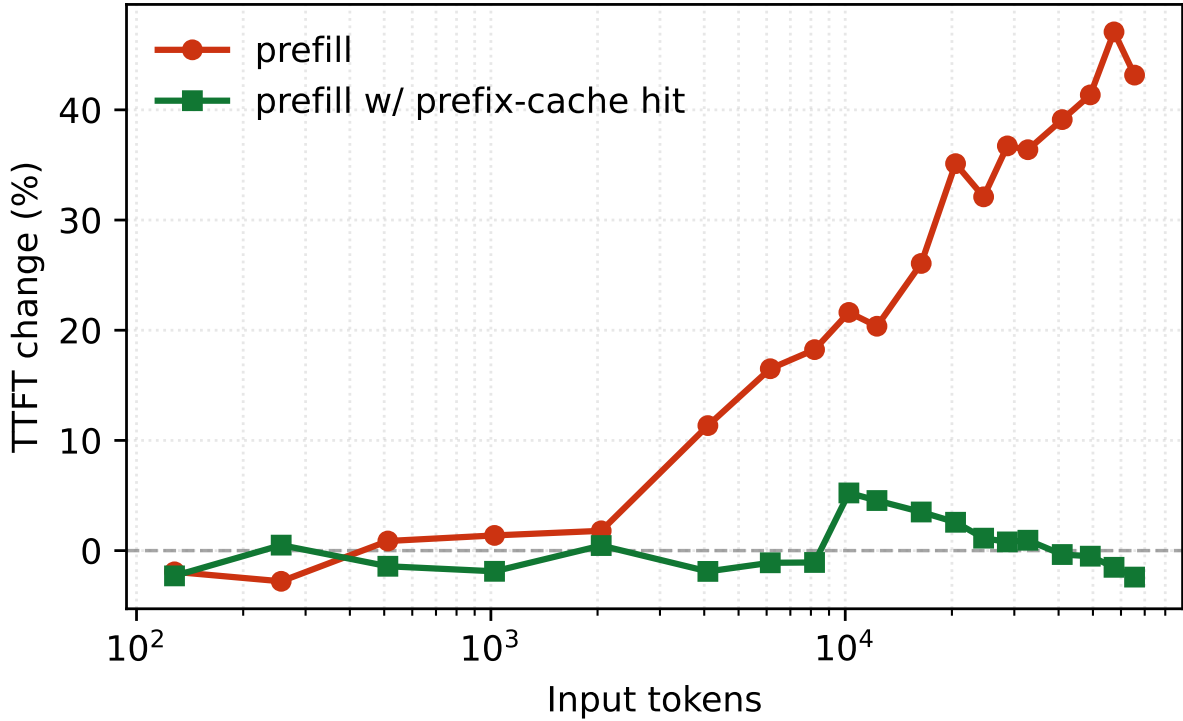


Figure 4.7: TTFT difference between uncapped GPUs and GPUs capped at 2/3 TDP. Power capping impacts the prefill stage heavily but has a marginal effect on prefix caching.

Power capping has a marginal effect on the decode stage of agentic workloads.

We repeat the experiments in Figure 4.4 on power-capped GPUs and show the results in Figure 4.8. As depicted, decoding is marginally impacted when memory is saturated, *i.e.*, the operating regime where agentic operations occur. This insight allows ConServe to delegate decoding to less powerful GPUs when possible.

4.4 ConServe

ConServe instantiates conversation-level scheduling with a minimal placement policy. As illustrated in Figure 4.9, the system runs on a prefill-decode disaggregated architecture: one model replica is dedicated to processing input prompts (the *prefiller node*), and the remaining replicas are dedicated to generating output tokens (the *decoder replicas*). A conversation is

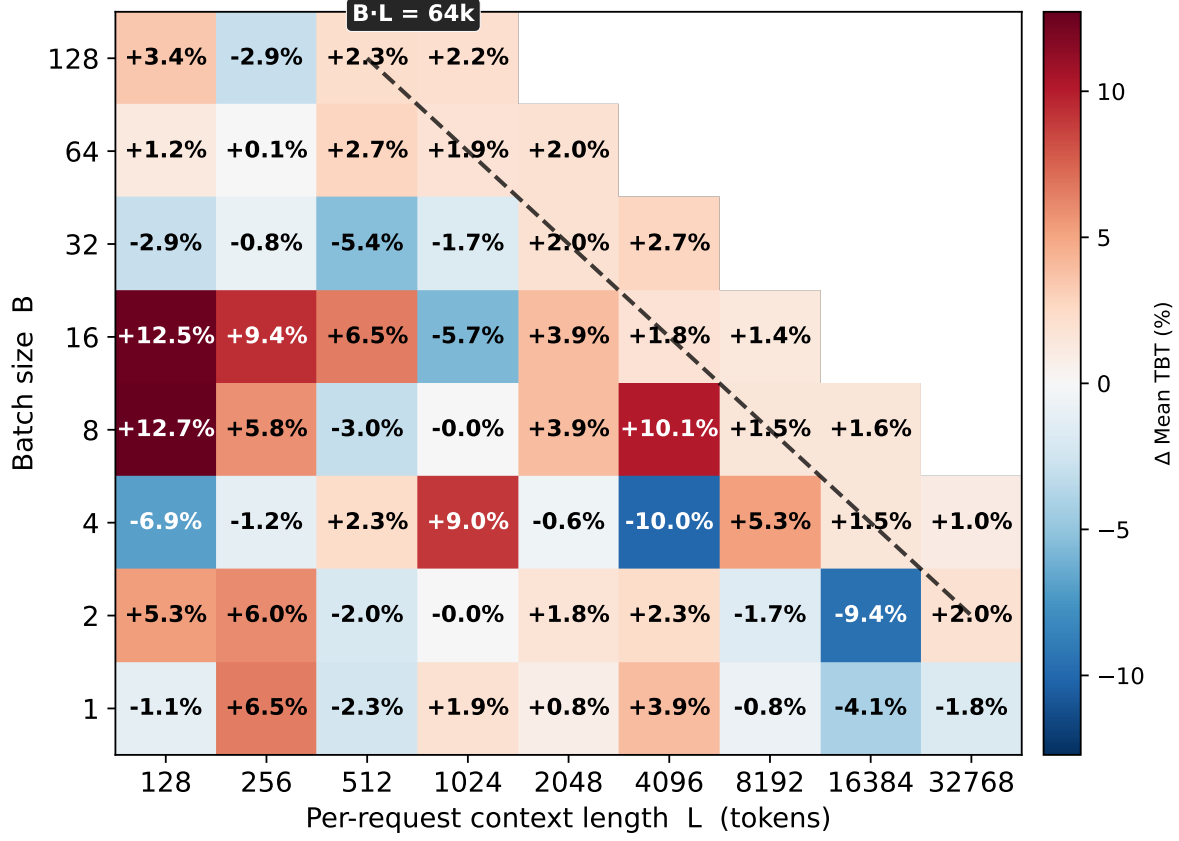


Figure 4.8: Heat map of mean TBT difference between uncapped GPUs and GPUs capped at 2/3 TDP across a range of batch sizes and context lengths. The dashed black line highlights the boundary between unsaturated (lower left) and saturated (upper right) memory bandwidth regions. Power capping has a marginal effect in the saturated memory bandwidth domain.

bound to one decoder replica at arrival, and the binding persists for the conversation’s lifetime. The first-turn prefill is routed to the prefiller where the long initial prompt is processed; the resulting KV cache is transferred once to the bound decoder. The first-turn decode and every subsequent turn’s incremental prefill and decoding then execute on that decoder with full KV cache reuse across turns. The system makes one routing decision per conversation and predicts no decode-side quantity at any point.

A ConServe deployment consists of one prefill node and N decoder replicas, with N chosen such that the prefill node saturates before the decoder pool under the workload’s

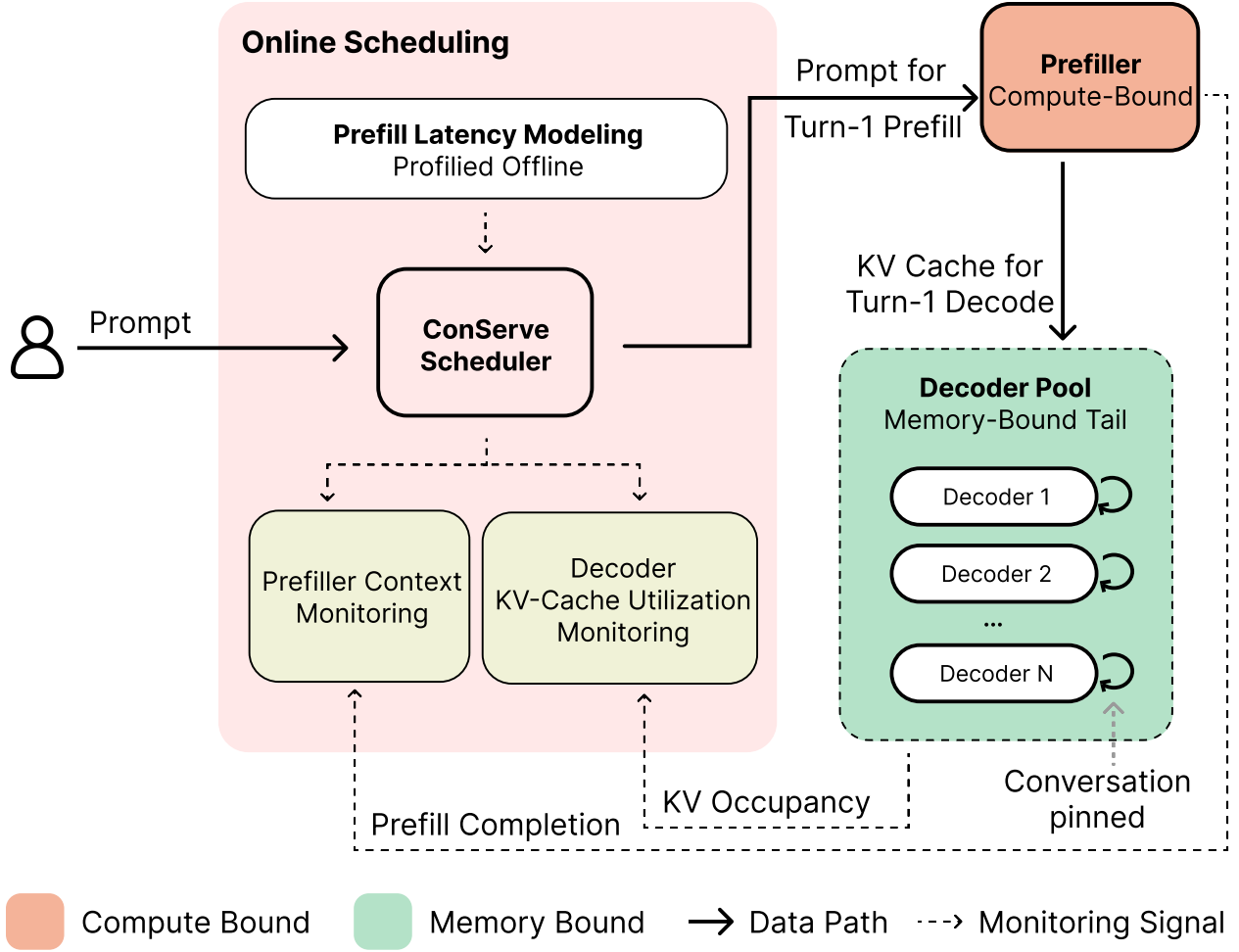


Figure 4.9: ConServe System Architecture.

input-to-output token ratio. This deliberate over-provisioning of decoders places the system’s throughput ceiling on the prefill side, where the relevant signal (input token rate) is observable at admission time and the cost-per-token relationship is deterministic (§4.3.1). The scheduler can therefore decide whether to admit a new conversation, where to bind it, and when to scale out by reading current system state rather than forecasting decode behavior.

4.4.1 Instance Configuration

Provisioning N is bounded by two constraints on the decoder side. The first is throughput: aggregate decoder throughput must process the tokens consumed and generated across each conversation’s lifetime. The second is memory: each decoder pins its bound conversations and holds their KV cache in GPU memory, limiting concurrent conversations per replica to B slots, where B is determined by GPU memory capacity and per-conversation peak KV footprint.

Let R denote the arrival rate, T_d the per-decoder token throughput, L_d the mean per-conversation token volume handled by decoders (turn-1 decode plus all turn 2+ prefill and decode), and W the mean wall-clock lifetime including time spent on external tool calls. N decoders must satisfy both constraints simultaneously:

$$N \cdot T_d \geq R \cdot L_d \quad (\text{throughput}) \tag{4.1}$$

$$N \cdot B \geq R \cdot W \quad (\text{memory}) \tag{4.2}$$

The prefill node saturates at arrival rate $R^* = T_p/L_{in}$, where T_p is the prefill input throughput and L_{in} the mean first-turn input length. ConServe provisions N as an integer more than satisfying both inequalities at $R = R^*$. This places the throughput bottleneck on the prefill node, where input token rate maps deterministically to expected utilization via the offline-profiled latency curve. The specific value of N used in our evaluation is given in §4.5.1.

4.4.2 Reactive Scheduling

The scheduler operates on two signals, both directly observable. The first, the prefill latency curve, is profiled offline as a deterministic function of input token count (§4.3.1); given an

incoming conversation’s first-turn prompt length, the scheduler reads off the prefill node’s expected utilization in constant time. The second is per-decoder *active* KV cache occupancy. Because every turn reuses the conversation’s accumulated history, every byte of KV cache held for a live conversation is part of its active working set; the scheduler decrements occupancy on conversation termination so the signal always reflects only currently active state rather than allocated-but-idle memory. Both signals are properties of state the system already maintains, not forecasts of what the workload will do next.

A new conversation enters the system at its first turn. The first-turn prefill is routed unconditionally to the prefill node. Once the prefill completes and the KV cache is transferred, the scheduler binds the conversation to the decoder replica with the lowest current KV cache occupancy. The binding is persistent: every subsequent turn for this conversation executes on the same decoder, with no re-evaluation of where it should go.

Because the two signals are direct measurements of system state, the same observability extends to capacity management. Imminent prefill saturation is detectable from the input token rate against the profiled curve, and decoder saturation surfaces as aggregate KV cache pressure across the pool. Provisioning additional capacity in either tier follows from the same observations the scheduler already makes.

4.4.3 *Heterogeneous GPU Mapping*

ConServe segments a conversation into a compute-bound first-turn prefill and a memory-bound tail, mirroring the phase structure that prefill-decode disaggregation exploits. This segmentation maps cleanly onto heterogeneous GPU tiers. Memory-bound workloads scale less aggressively with newer hardware than compute-bound workloads do, so decoder replicas, which run the memory-bound tail, can execute on older lower-power GPUs without proportional throughput loss, while the prefill node takes the newest high-power GPU available.

This mapping requires no changes to the scheduling logic. The same two observable signals drive placement on either uniform or heterogeneous hardware, and the conversation-level commitment ensures KV state is transferred between tiers at most once per conversation. Energy efficiency emerges as a consequence: the system spends high-power compute only on the turn that requires it, while the memory-bound tail runs on hardware that consumes substantially less power.

4.5 Evaluation

Our evaluation addresses the following questions:

- **Q1:** Does ConServe match or beat all baselines on latency?
- **Q2:** Does ConServe maintain SLO at saturation?
- **Q3:** Does ConServe avoid the routing error sensitivities of per-turn approaches?
- **Q4:** Does the heterogeneous variant translate to energy efficiency wins?

4.5.1 Experiment Setup

Workload and hardware We evaluate on agentic traces generated from SWE-bench _bm25_13K with swe-agent, using Qwen3-Coder-30B-A3B-Instruct as the trace-generation model. Traces are replayed on a 4-GPU NVIDIA A40 machine with Qwen3-0.6B as the served model. The smaller served model leaves sufficient memory headroom for the long KV caches characteristic of multi-turn agentic conversations. To simulate the serving performance on heterogeneous GPUs, we power cap 3 GPUs to 2/3 of their TDP (300W) to represent prior-generation GPUs with less compute capacity but similar memory hardware.

System configuration. Each served model replica occupies one GPU. The prefiller sustains roughly 25k input tokens per second, while each decoder produces 1k output tokens per

second and accommodates approximately 300k tokens of KV cache. We allocate one prefiller and three decoders across the four-GPU machine. As a sanity check following the analysis in §4.4.1, an average of 15k input tokens and 1k output tokens per conversation would require at least 1.67 decoders per prefiller. The 3 decoder configuration is a guarantee that the prefiller saturates before the decoders.

Implementation and Baselines.

We implement ConServe on with vLLM [Kwon et al., 2023] as the LLM serving engine and LMCache [Cheng et al., 2025] as the Prefill-Decode disaggregation manager. We configure the prefiller model replica with a vLLM standard 8192 chunked token size per batch to prevent aggressive queueing due to long context. We compare against three baselines.

- **Collocated** uses all four GPUs as mixed-batch replicas, and runs each conversation on a single replica with prefill and decode batched together. Each model replica is configured with a 2944 chunked token size. This size is derived from our offline profiling, such that the iteration latency is within 5x baseline SLO, as practiced in prior work [Stojkovic et al., 2025].
- **Full Disaggregation** routes every turn through a dedicated prefill node.
- We also implement **AMPD** [He et al., 2026] at our best effort, which performs per-turn prediction-based disaggregation for multi-turn workloads. As bi-directional KV cache transfer (between two model replicas) and management is infeasible to implement in a reasonable amount of time, we simulate the latency of KV cache transfer and stall requests for that period. We further post-process the experiment trace with our offline profiling results at best effort. Because in all our experiments, executing the turn 2+ prefill locally on the decoders is more efficient than migrating it to the prefiller, all reported AMPD results except Figure 4.12 induce a 10% wrong prediction rate, i.e., the scheduler migrates the turn 2+ prefill to the prefiller for 10% of all turns.

Metrics. We report three latency metrics: TTFET, last-turn TBT, and end-to-end (E2E) latency, motivated in §4.2.2. Together they capture the three user-facing performance dimensions that matter for multi-turn agentic conversations. For comparison with prior work, we additionally report conventional per-turn TTFT and TBT distributions, though these metrics conflate intermediate tool-call turns with the final user-facing reply and therefore do not reflect end-user experience.

4.5.2 Q1: Agentic Conversation Performance

Agentic performance. Figure 4.10 reports normalized geometric mean and p95 latencies for TTFET, last-turn TBT, and E2E across the five systems at request rates from 0.5 to 1.5 conversations per second under Poisson arrivals. We additionally report results at 1.634 conv/s, which is not a Poisson rate but a synthesized arrival pattern that holds the prefiller exactly at its saturation throughput, isolating behavior at the system’s capacity limit. For AMPD we fixed a wrong prediction rate of 10%, with further discussion in section 4.5.4.

ConServe outperforms AMPD by reducing up to 19.17% and +51.08% on geometric mean and p95 TTFET. At low to moderate loads (0.5 to 1.25 conv/s), ConServe, AMPD, and Collocated remain within $2\times$ of the baseline. At saturation (1.5 to 1.634 conv/s), ConServe holds steady while Collocated and AMPD degrade sharply at p95, reflecting their respective structural failure modes: Collocated suffers from prefill-decode contention growing with load, and AMPD accumulates routing-decision errors that compound into prefiler queue pressure (§4.5.4). Full Disaggregation is uncompetitive at every operating point, with gmean TTFET and E2E exceeding $10\times$ the baseline. Routing every turn through the prefill node forfeits the cross-turn KV cache reuse that dominates efficiency in multi-turn workloads, and additionally pays a per-turn KV transfer cost.

Last-turn TBT inverts the ordering. Full Disaggregation achieves the lowest TBT across all loads, since its decoder replicas process only decode tokens with no prefill interference.

At saturation, ConServe’s gmean last-turn TBT ($2.49\times$ baseline) is 91.85% higher than Full Disaggregation’s ($1.35\times$). This penalty is the price of preserving cross-turn KV cache reuse, and is dwarfed by the prefill latency that Full Disaggregation incurs on every turn: its gmean TTFET reaches $18.78\times$ baseline against ConServe’s $2.53\times$, translating to $4.7\times$ worse E2E. AMPD produces essentially identical TBT to ConServe despite disaggregating roughly 10% of turn 2+ prefills, showing that partial disaggregation at this rate gains no measurable TBT benefit while still paying the TTFET cost of those routing decisions (§4.5.4). Collocated has the worst p95 TBT at saturation ($7.52\times$ baseline, 72% worse than ConServe), reflecting the prefill-decode contention from batching long initial prefills with ongoing decoder workloads.

A deeper look at LLM metrics. Figure 4.11 shows the per-turn TTFT and TBT CDFs across all turns and conversations. Full Disaggregation pays both the full prefill cost and the KV transfer on every turn, producing the rightmost TTFT distribution; its prefill-free decoders produce the leftmost TBT distribution.

The TTFT CDF resolves into three regimes for the remaining systems. Below the 0.7 percentile, ConServe, AMPD, and Collocated overlap: these are turn 2+ prefills, which complete quickly regardless of where they execute. Between 0.7 and 0.8, ConServe pulls ahead significantly. This regime contains turn-1 prefills, which in Collocated batch with ongoing turn 2+ prefills on the same replicas, and in AMPD wait behind the queue formed by wrongly disaggregated turn 2+ prefills on the prefill node. Above 0.8, the ordering inverts: Collocated becomes best because its four-GPU pool prevents compute saturation, while AMPD becomes worst as its accumulated prefill queue stretches into the tail.

The TBT CDF completes the picture. ConServe and AMPD produce nearly identical distributions across the full range; AMPD’s 10% per-turn disaggregation does not meaningfully change which prefills reach the decoders. Below the 0.7 percentile, Collocated outperforms both because it dedicates four GPUs to decoding while ConServe and AMPD dedicate only three, giving Collocated one-third more aggregate memory bandwidth. Above the 0.7

percentile, this advantage inverts as long turn-1 prefills batch with ongoing decode on Collocated’s replicas and inflate per-step latency, while ConServe and AMPD insulate the decoders from this contention.

4.5.3 Q2: Serving Throughput and Quality

Figure 4.10 (third row) reports SLO violation rates. We set the SLO threshold at $5\times$ the baseline latency for each metric, where the baseline is measured by executing a single request without batching or interference. This $5\times$ threshold follows standard practice in prior LLM serving work. ConServe sustains zero SLO violations across all three metrics at every request rate tested, including the 1.634 conv/s saturation point. The baselines fail at saturation in metric-specific patterns that reveal their underlying failure modes. Collocated violates both TTFET and last-turn TBT SLOs at 1.634 conv/s, since prefill-decode contention degrades both phases simultaneously. AMPD violates only the TTFET SLO, because its per-turn routing errors accumulate on the prefiller but leave the decoder workload unchanged (§4.5.4). Full Disaggregation violates the TTFET and E2E SLOs at nearly 100% across all loads, with last-turn TBT remaining under the SLO due to its interference-free decoder design.

The structural reason for ConServe’s SLO robustness is that the prefiller’s load is bounded by design. Only turn-1 prefills route to the prefill node, and the rate at which they arrive is gated by incoming conversation arrival. Since turn-1 prefill latency is a deterministic function of input token count (§4.3.1), the system operates exactly at the prefiller’s saturation capacity without exceeding it. Neither baseline has this property: Collocated absorbs all load on a single batched replica pool, and AMPD’s per-turn routing injects unpredictable additional prefiller load that breaks the operating bound.

4.5.4 Q3: Pitfalls of Per-Turn Routing

AMPD’s per-turn predictions are susceptible to three sources of error: the cost model does not account for decoder KV cache utilization, which directly affects decode throughput (§4.3.2); when incremental prefills batch with heavy ongoing decode, the resulting latency has high variance that offline profiling cannot capture; and most consequentially, the cost model does not account for prefiler queueing pressure, so each erroneously disaggregated turn 2+ prefill adds to the prefiler’s queue and delays all subsequent prefill workload. ConServe sidesteps all three by scheduling on incoming input prompt size alone, which is fully observable and makes the prefiler’s saturation point deterministic, enabling predictable autoscaling.

Figure 4.12 shows that AMPD is highly sensitive to even small cost-estimation errors. At a 0% wrong-prediction rate, AMPD routes every turn 2+ prefill locally and reduces to ConServe, so the curves begin at ConServe’s operating point. At just 5% wrong predictions, SLO violations reach roughly 7.8% for TTFET and 6.3% for E2E. As the wrong-prediction rate climbs to 50%, both gmean and p95 latencies grow linearly and SLO violations exceed 50%. The linearity follows directly from the queueing mechanism: each wrong decision adds one unit of unanticipated load to a system operating at saturation, so the resulting queue delay scales with the cumulative excess load. Our evaluation runs a finite trace subset, after which the prefiler queue drains. In a sustained production workload, the queueing pressure continues to accumulate, and the SLO violations observed here form a lower bound on AMPD’s deployment-scale impact. Last-turn TBT remains flat throughout, since routing errors affect intermediate turns but never reach the user-visible final turn. ConServe has no equivalent degradation curve: it makes no per-turn routing decision and therefore generates no prediction error.

The same routing errors degrade energy efficiency (Figure 4.12.d). A wrongly migrated turn 2+ prefill carries substantial reused KV cache; mixing this memory-heavy work with the

compute-bound turn-1 prefills on the prefiller lowers the prefiller’s utilization. The misrouted conversations also incur longer latency, and these delays accumulate into a longer wall-clock time to serve the full workload. At the 1.634 conv/s saturation point, with 10% wrong predictions, AMPD takes 15% longer than ConServe to drain the workload while delivering 7.5% lower tokens-per-joule. Because the workload processes a fixed token volume, the lower efficiency reflects higher total energy; that the 15% time penalty exceeds this energy increase indicates the GPUs spend the extra time under-utilized, idling on queue rather than computing. The effect compounds with error rate: AMPD’s tokens-per-joule declines monotonically toward 58 tokens/J at a 50% wrong-prediction rate, while ConServe holds constant by construction.

4.5.5 Q4: *Serving with Heterogeneous GPUs*

ConServe segments each conversation into a compute-bound first-turn prefill and a memory-bound tail, mirroring the prefill-decode abstraction at conversation granularity. If this segmentation is faithful, the memory-bound tail should tolerate a reduced power budget on the decoder GPUs: its throughput is gated by memory bandwidth, not by the compute capacity a power cap removes. Figure 4.13 tests this, comparing a homogeneous configuration (300W prefill, 300W decoders) against a heterogeneous one (300W prefill, 200W decoders).

ConServe realizes two energy wins. Its scheduling alone is energy-efficient in the homogeneous configuration, since confining compute-bound work to one prefill node and running the memory-bound tail without per-turn disaggregation avoids the wasted transfers and idle cycles that lower the baselines’ tokens-per-joule. Capping the decoders to 200W then adds a further gain: tokens-per-joule rises by 22.75% while p95 TTFET, last-turn TBT, and E2E remain essentially unchanged. The memory-bound tail absorbs the power cap, so the heterogeneous mapping does not trade away ConServe’s scheduling advantage but compounds it.

The baselines do not share this property. Collocated batches prefill and decode on the same GPUs, so reducing decoder power to 200W starves the compute-bound prefill: its p95 TTFT worsens by 11.15% under the heterogeneous configuration. Full Disaggregation cannot meet the SLO in either configuration, leaving its energy numbers moot. AMPD gains the same power-cap benefit as ConServe, since it also runs decode on the capped GPUs, but it carries the per-turn routing penalties characterized in §4.5.5.

4.6 Related Work

LLM Request Scheduling. LLM request scheduling has been studied along several axes. Iteration-level scheduling batches and preempts at the granularity of individual decode steps to raise utilization [Yu et al., 2022, Sun et al., 2024], and recent schedulers route or reorder requests using predicted output lengths or learned rankings [Fu et al., 2024, Jain et al., 2025]. A parallel line schedules across the prefill-decode split: disaggregation systems decide where each phase executes and how the two scale independently [Zhong et al., 2024, Patel et al., 2024], while intra-GPU and unified designs revisit where the split is drawn [Shi et al., 2025, Gao et al., 2025, Wang et al., 2025]. Elastic systems extend this to autoscaling prefill and decode capacity in response to load [Zhang et al., 2025, Lai et al., 2025]. Production frameworks fold all of these together: NVIDIA Dynamo unifies KV-cache-aware routing, conditional prefill-decode disaggregation, SLO-driven autoscaling, and tiered KV cache management in one serving system [NVIDIA, 2025]. Whether these mechanisms are studied in isolation or integrated this way, the unit of scheduling remains the request, or its constituent phases and iterations. ConServe raises the unit to the conversation: it commits one placement per conversation and never revisits it, which is what lets scheduling condition on observable state rather than predicted per-request cost.

KV Cache Management. A separate line of work targets how key-value state is stored and reused rather than where the computation that produces it runs. Hierarchical and

distributed cache layers move KV state across memory tiers and across the cluster to extend reuse beyond a single instance [Gao et al., 2024, Qin et al., 2025, Hu et al., 2024, Cheng et al., 2025], and compression and offloading shrink or relocate the cache to fit larger working sets [Li et al., 2024, Liu et al., 2024b]. These mechanisms operate at a different layer than ConServe, which schedules computation and assumes only standard local prefix caching beneath it. The two compose: richer cache management widens what can be reused, while ConServe’s conversation-level placement keeps the reusable prefix on the instance that will read it, so after the first turn it never crosses the network again.

Agentic and Application-Aware Serving. A growing body of work optimizes serving for the structure of agentic applications. Parrot recovers prompt sharing and request dependencies through application-level dataflow [Lin et al., 2024]. A second line targets the tool-call period that separates an agent’s turns: InferCept retains, discards, or swaps the paused KV cache while an external call runs [Abhyankar et al., 2024], Continuum retains KV state across pauses under a time-to-live policy [Li et al., 2025], and Sutradhara co-designs orchestrator and engine to overlap tool execution with subsequent prefill and decode [Biswas et al., 2026]. KVFlow evicts and prefetches cache entries according to an agent execution graph [Pan et al., 2025], and Autellix raises the scheduling unit to the agentic program to order calls against head-of-line blocking [Luo et al., 2025]. Several of these optimize conversation-level rather than per-turn objectives: Continuum targets job completion time, and Sutradhara optimizes final-answer latency, the same quantity ConServe captures as TTFET. All of them operate within a conversation whose physical placement is fixed, managing its cache, overlapping its tool calls, and ordering its requests on an instance already chosen. None makes the placement decision itself: where the first-turn prefill runs, when its KV state transfers, and which decoder holds the conversation for its lifetime. ConServe commits this decision once from observable state and is orthogonal to this line of work, which caches, overlaps, and schedules the calls within a conversation ConServe has already placed.

4.7 Conclusion

We introduce ConServe, a conversation-level disaggregation scheduler for agentic serving. ConServe raises the scheduling unit from the turn to the conversation, which collapses turn-level irregularity into a stable two-phase structure: a compute-bound first-turn prefill followed by a memory-bound tail. ConServe routes the first-turn prefill to a high-power prefiller provisioned as the system bottleneck, transfers the KV cache exactly once, and pins each conversation to a decoder replica for its lifetime, conditioning placement only on observable turn-1 input length and per-decoder KV occupancy rather than predicted per-turn cost. Experiments on agentic workloads show that ConServe reduces p95 TTFET by 51.08% over per-turn routing while sustaining zero SLO violations at saturation, and its phase split maps onto heterogeneous GPU tiers for a further 22.75% gain in energy efficiency.

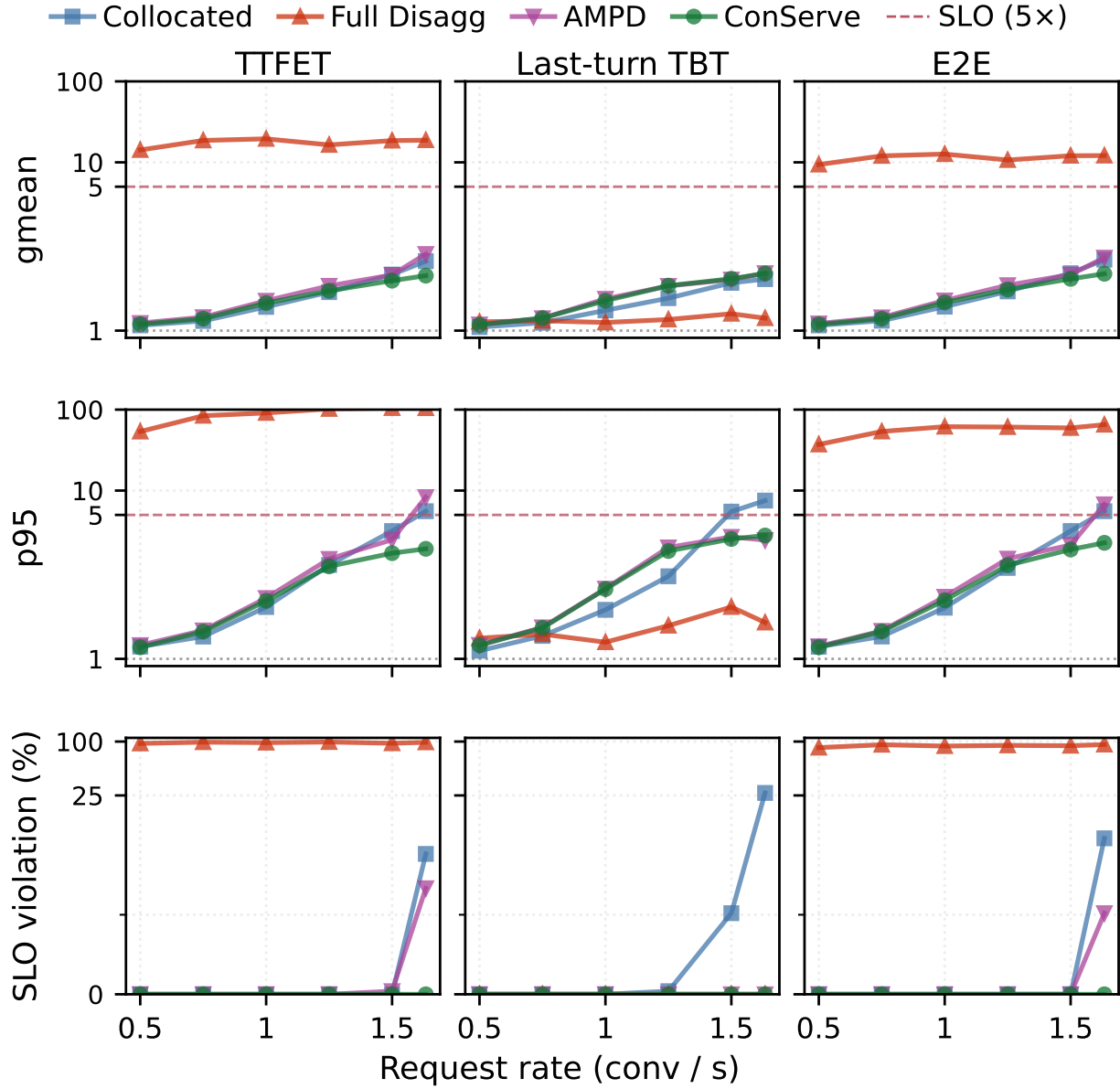


Figure 4.10: Normalized agentic performance results over multiple request arrival rates (lower is better). AMPD has 10% wrong prediction rate.

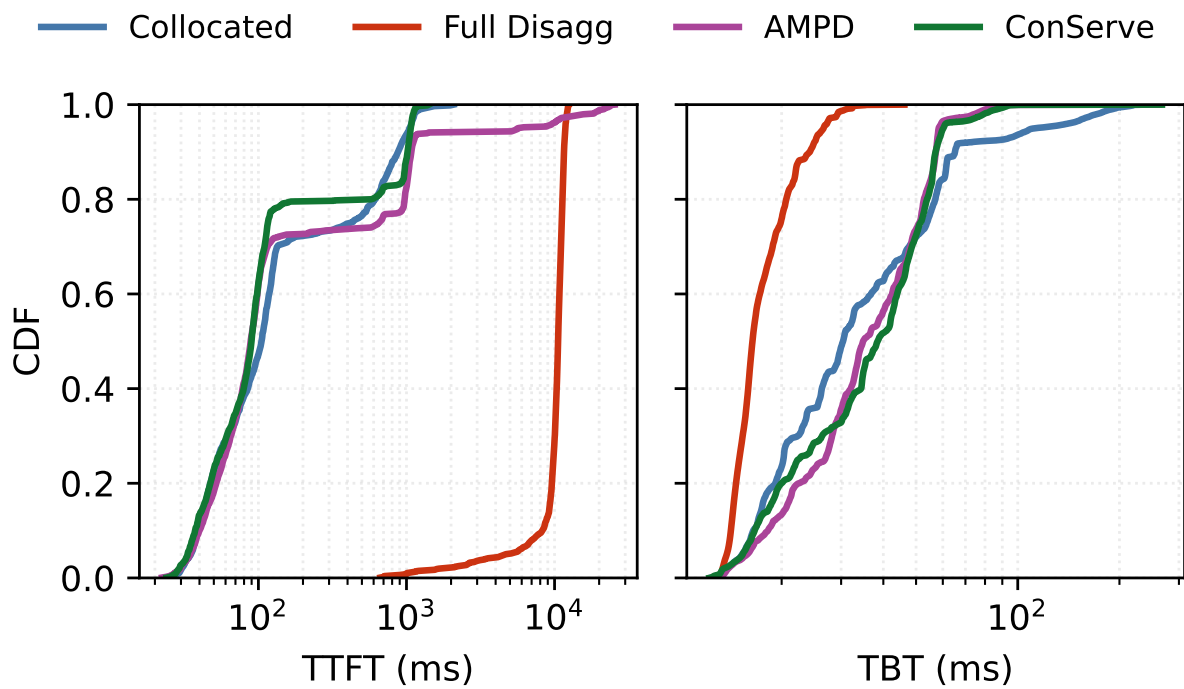
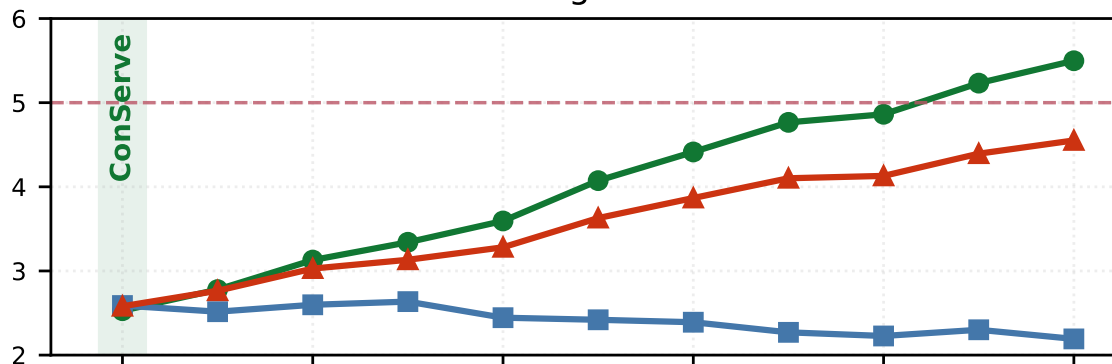


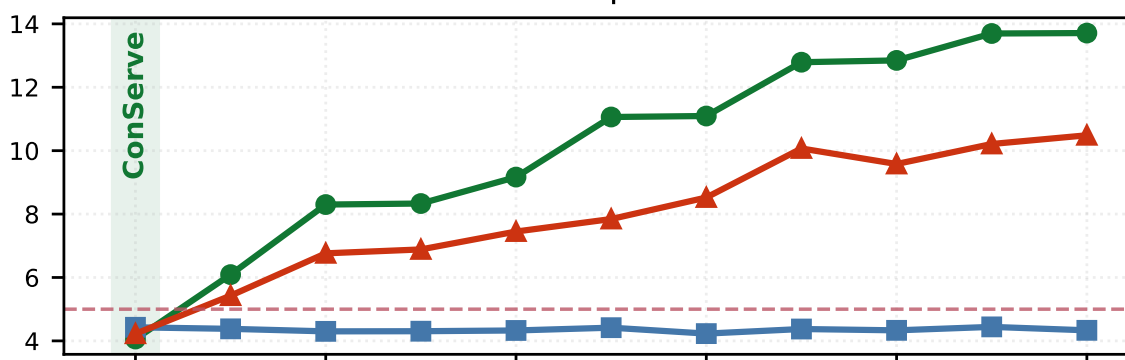
Figure 4.11: Overall TTFT and TBT distribution at RPS 1.634.

● TTFET ■ Last-turn TBT ▲ E2E - - - SLO (5× baseline)

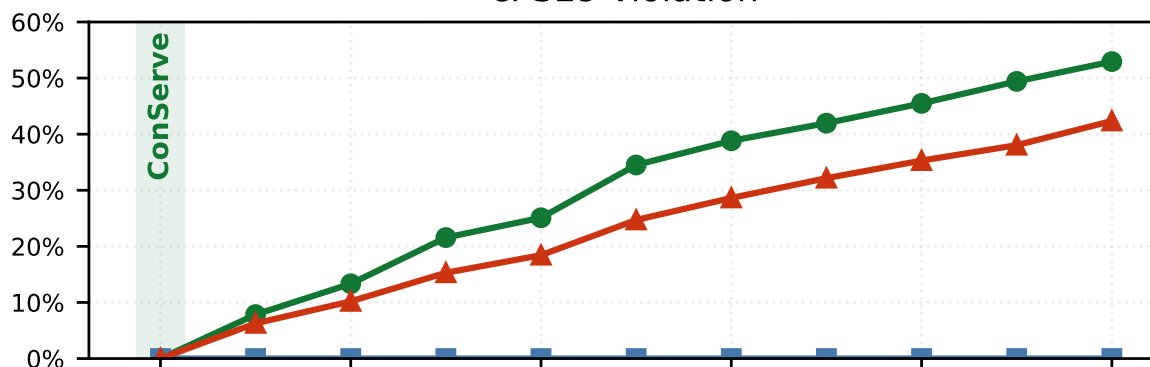
a. Normalized gmean latencies



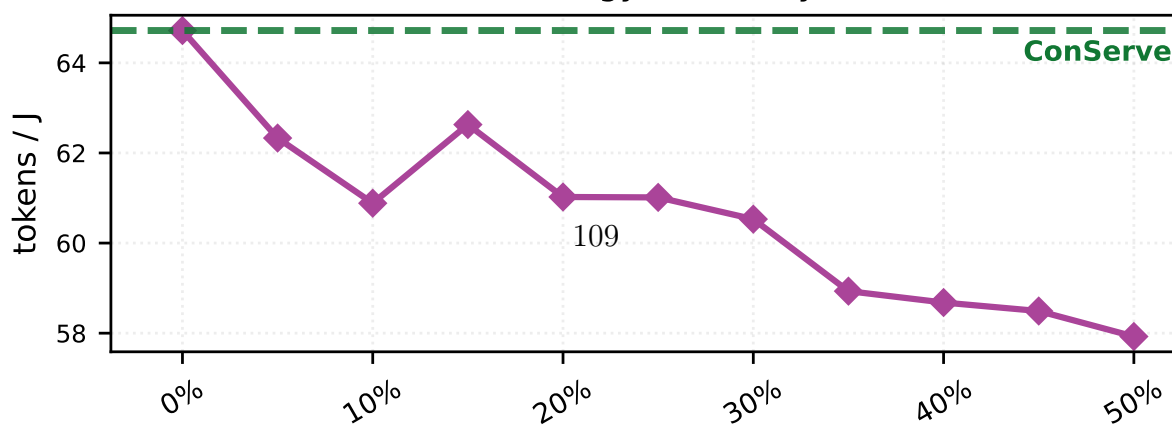
b. Normalized p95 latencies



c. SLO violation



d. Energy efficiency



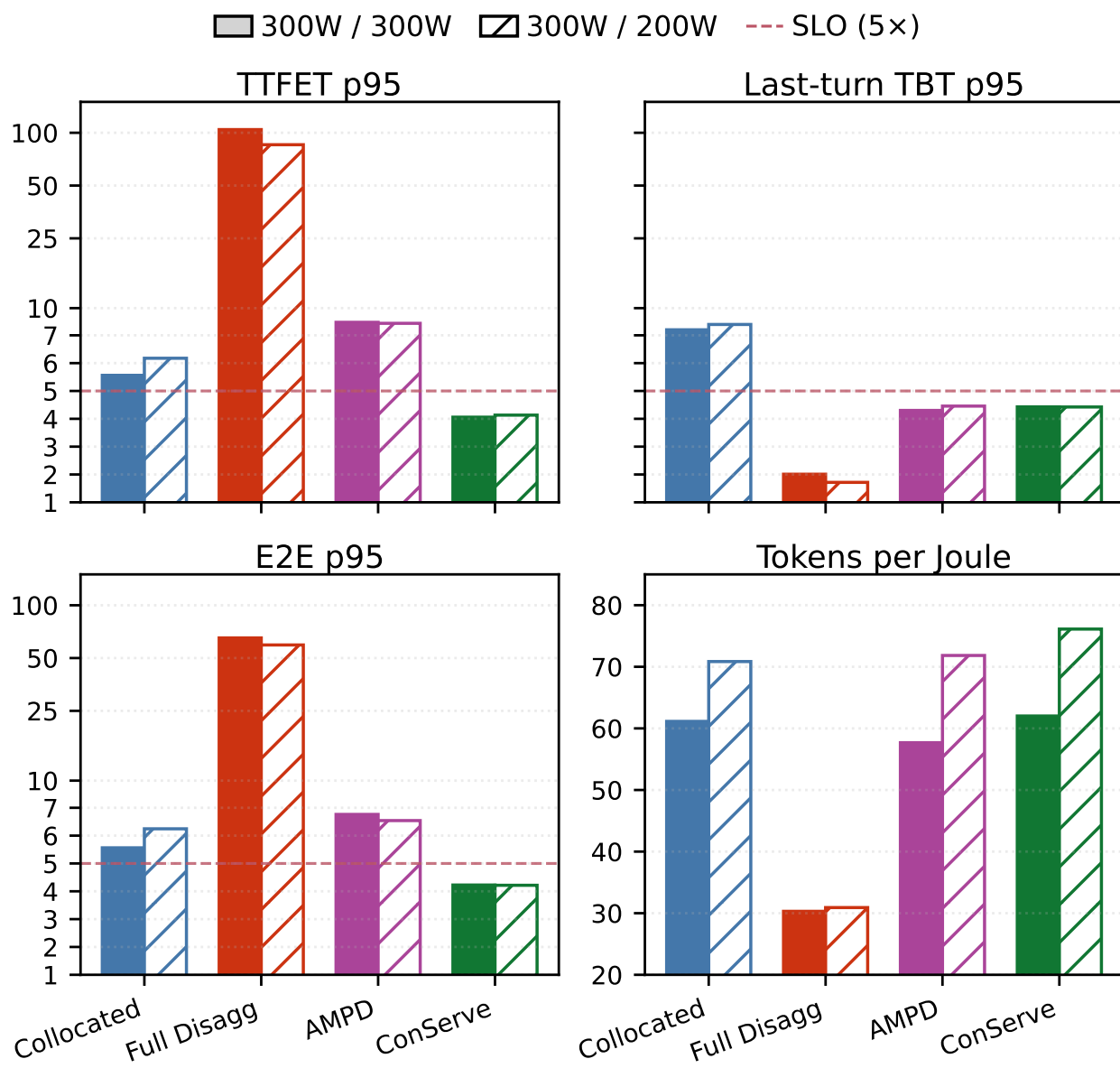


Figure 4.13: P95 agentic performance and energy efficiency on heterogeneous GPUs.

CHAPTER 5

FUTURE WORK AND CONCLUSIONS

5.1 Future Work

This dissertation establishes by construction that useful observable signals exist, but not when they do. Each system was the product of studying a particular workload until the right signal revealed itself, and a predictive account of when such a signal can be found, and how much performance conditioning on it recovers, remains open. The question can be posed sharply against the theory recalled in Section 1.1. Clairvoyant scheduling assumes complete information and non-clairvoyant scheduling assumes none; a reactive scheduler conditions on a partial but error-free observation that lies between them. Where learning-augmented analysis asks how performance degrades as a prediction’s error grows, the dual question here asks how it degrades as an observation’s completeness shrinks. A characterization of signal sufficiency, identifying when a low-cost observable quantity is a sufficient statistic for a scheduling objective and bounding the gap to the clairvoyant optimum when it is not, would turn the design method this dissertation applies by hand into one that can be reasoned about in advance.

ConServe can also be extended outward, from a serving system to the infrastructure that runs it. The adoption of large language models, and the agents and applications built on them, is expanding datacenters faster than was anticipated, and the workloads driving that expansion are new enough that historical models describe them poorly. The pressure reappears at every layer ConServe sits above: power must be allocated across machines whose draw is set by workloads as volatile as the ones ConServe schedules, cooling must remove heat those workloads generate unpredictably, and monitoring and placement must keep pace with a workload mix that shifts as fast as the applications evolve. Each is a scheduling problem in which the quantity a controller would most want to forecast is the

one the workload withholds, and each, by the argument of this dissertation, should admit an observable signal of its own. DPS takes a first step at the power layer; extending the same discipline to cooling, monitoring, and cluster-wide placement, and finding the signal each of those decisions can be conditioned on, is open work. ConServe itself covers only one agentic pattern, a single tool-using conversation with a long memory-bound tail, and the broader space of LLM applications, with parallel tool calls, branching sub-agents, and mixed interactive and batch demand, will need its own observable structure before the conversation-level abstraction reaches it.

A further direction is to learn the observation itself. Both DPS and ConServe already contain fitted functions over observable quantities: DPS’s Kalman filter recovers a node’s current power from noisy readings, and ConServe’s offline curve maps a measured input length to its prefill cost. In each, the fitted function summarizes state that is already determined rather than forecasting state that is not. Generalizing this, a scheduler could learn a richer sufficient statistic of present state from raw telemetry, in place of a hand-chosen one such as DPS’s power derivative. Much existing work already learns to predict system behavior, performance, power, contention, or runtime, from such metrics. But learning here plays as the main role: the prediction is the decision, and the schedule is only as good as the model. Used as an observation, learning plays a smaller role. It refines the signal a reactive scheduler conditions on, while the worst-case guarantee, such as DPS’s floor at constant allocation, rests on the reactive policy and not on the learned signal at all. Learning can then only raise performance in the common case and never put the guarantee at risk.

5.2 Conclusion

Each of the three systems in this dissertation began from a problem that appeared to demand prediction and ended without one. Robin-Hood schedules tasks whose arrivals cannot be enumerated, DPS allocates power whose demand shifts faster than a model can track, and

ConServe serves conversations whose cost is settled only as they run. In each case the workload withholds exactly the quantity a predictive scheduler would need, and the resolution took the same form: rather than estimate that quantity, find a signal already determined when the decision is made, and schedule on it.

What made this possible was never the signal alone but the representation that exposed it. Robin-Hood’s unified task abstraction lets a single queue hold both in-memory and streaming work, so one observable quantity, queue occupancy, governs both. DPS’s power dynamics turn a node’s recent history into a quantity that anticipates its near future without modeling its workload. ConServe’s shift from the turn to the conversation collapses an irregular, prediction-laden problem into a two-phase structure whose costs are read from input length and cache occupancy. The recurring design act is the choice of representation under which a reactive signal becomes sufficient; where a dependence on prediction appeared, it was a property of the unit being scheduled, not of the workload itself.

This dissertation has described reactive, observation-based scheduling systems and argued for the principle they share. A reactive scheduler conditions its decisions only on quantities already determined at the moment it acts, never on a forecast of demand it cannot yet see. A signal of this kind carries no estimation error, costs nothing to keep accurate, and cannot go stale as a workload drifts. The central claim is that such a signal is not merely a fallback for when prediction fails: wherever it carries enough structure to schedule well, conditioning on it suffices at none of prediction’s cost, and where prediction is impossible, it stands in prediction’s place. The work of this dissertation has been to show that such signals exist, and to find them, in settings where they are far from obvious.

REFERENCES

- Top 500 supercomputing site, Jun 2022. URL <https://top500.org/>.
- Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiyang Zhang. Inference: Efficient inference support for augmented large language model inference, 2024. URL <https://arxiv.org/abs/2402.01869>.
- Amin Aghdashi and Seyed Latif Mirtaheeri. Novel dynamic load balancing algorithm for cloud-based big data analytics. *The Journal of Supercomputing*, 78:4131–4156, 2022. doi:10.1007/s11227-021-04024-8. URL <https://doi.org/10.1007/s11227-021-04024-8>.
- Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. VIDUR: A large-scale simulation framework for LLM inference. In Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa, editors, *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024a.
- Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 117–134. USENIX Association, 2024b.
- Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *Proc. VLDB Endow.*, volume 8, pages 1792–1803, 2015.
- George Almasi. Pgas (partitioned global address space) languages., 2011.
- Henrique C. M. Andrade, Buğra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014. ISBN 9781107015548.
- Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13*, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450317580. doi:10.1145/2488222.2488267. URL <https://doi.org/10.1145/2488222.2488267>.
- Anthropic. Models, usage, and limits in claude code. <https://support.claude.com/en/articles/14552983-models-usage-limits-in-claude-code>, April 2026. Accessed: 2026-05-30.

- Apache Software Foundation. Apache storm, 2011. URL <https://github.com/apache/storm>. Accessed: 2024-11-13.
- Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold S. Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 601–613, 2018.
- Sidra Aslam and Munam Ali Shah. Load balancing algorithms in cloud computing: A survey of modern techniques. In *2015 National Software Engineering Conference (NSEC)*, pages 30–35, 2015. doi:10.1109/NSEC.2015.7396341.
- David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.
- Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007. doi:10.1109/MC.2007.443.
- R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–76, 2004. doi:10.1109/MC.2004.217.
- Anish Biswas, Kanishk Goel, Srivarshinee S, Jayashree Mohan, Alind Khare, Anjaly Parayil, Ramachandran Ramjee, and Chetan Bansal. Sutradhara: An intelligent orchestrator-engine co-design for tool-based agentic inference, 2026. URL <https://arxiv.org/abs/2601.12967>.
- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’95, page 207–216, New York, NY, USA, 1995a. Association for Computing Machinery. ISBN 0897917006. doi:10.1145/209936.209958. URL <https://doi.org/10.1145/209936.209958>.
- Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995b. ISSN 0362-1340. doi:10.1145/209937.209958. URL <https://doi.org/10.1145/209937.209958>.
- OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2023. URL <https://www.openmp.org>.
- Stephanie Brink, Matthew Larsen, Hank Childs, and Barry Rountree. Evaluating adaptive and predictive power management strategies for optimizing visualization performance on supercomputers. *Parallel Computing*, 104-105:102782, 2021. ISSN 0167-8191. doi:<https://doi.org/10.1016/j.parco.2021.102782>. URL <https://www.sciencedirect.com/science/article/pii/S0167819121000399>.

- Thilina Buddhika, Ryan Stern, Kira Lindburg, Kathleen Ericson, and Shrideep Pallickara. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569, 2017. doi:10.1109/TPDS.2017.2723403.
- Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. In *Proceedings of the VLDB Endowment*, volume 8, pages 1441–1451, 2015. doi:10.14778/2824032.2824063.
- Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS ’15, page 344–347, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332866. doi:10.1145/2675743.2776766. URL <https://doi.org/10.1145/2675743.2776766>.
- Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, DEBS ’16, page 69–80, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340212. doi:10.1145/2933267.2933312. URL <https://doi.org/10.1145/2933267.2933312>.
- Badrish Chandramouli, Jonathan Goldstein, Ryan Groves, Donald Kossmann, Justin Levandoski, James Hunter, Michael Barnett, Roman Soule, John W. T. Lillibridge, John C. Platt, James F. Terwilliger, and Andrew Tomkins. Trill: A high-performance incremental query processor for diverse analytics. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 401–413, 2016.
- Steve J. Chapin, Dimitrios Katramatos, John Karpovich, and Andrew S. Grimshaw. The legion resource management system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 162–178, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-47954-3.
- Harrison Chase. Langchain, 2022. URL <https://github.com/langchain-ai/langchain>.
- Shilpa Chaturvedi, Sahil Tyagi, and Yogesh Simmhan. Collaborative reuse of streaming dataflows in iot applications. In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 403–412, 2017. doi:10.1109/eScience.2017.54.
- Andreas Chatzistergiou and Stratis D. Viglas. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM ’14, page 1579–1588, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325981. doi:10.1145/2661829.2661882. URL <https://doi.org/10.1145/2661829.2661882>.

- Huankai Chen, Frank Wang, Na Helian, and Gbola Akanmu. User-priority guided min-min scheduling algorithm for load balancing in cloud computing. In *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pages 1–8, 2013. doi:10.1109/ParCompTech.2013.6621389.
- Yihua Cheng, Yuhan Liu, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference. *arXiv preprint arXiv:2510.09665*, 2025.
- Andrew A. Chien, Charles Colley, Jianru Ding, Alexander Fell, David F. Gleich, Henry Hoffmann, Moubarak Jeje, Rajat Khandelwal, Yanjing Li, Jose M. Monsalve-Diaz, Marziyeh Nourian, Andronicus Rajasukumar, Jiya Su, Tianshuo Su, Yuqing Wang, Wenyi Wang, Ruiqi Xu, and Tianchi Zhang. Updown: A supercomputer co-designed for scalable graph processing. *IEEE Transactions on Parallel and Distributed Systems*, 37(7):1553–1570, 2026. doi:10.1109/TPDS.2026.3681212.
- Genshen Chu, Yuanjie He, Lingyu Dong, Zhezhao Ding, Dandan Chen, He Bai, Xuesong Wang, and Changjun Hu. Efficient algorithm design of optimizing spmv on gpu. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '23*, page 115–128, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701559. doi:10.1145/3588195.3593002. URL <https://doi.org/10.1145/3588195.3593002>.
- J. Corbalan, A. Duran, and J. Labarta. Dynamic load balancing of mpi+openmp applications. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 195–202 vol.1, 2004. doi:10.1109/ICPP.2004.1327921.
- Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, August 2004a. ISSN 0146-4833. doi:10.1145/1030194.1015471. URL <https://doi.org/10.1145/1030194.1015471>.
- Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, page 15–26, New York, NY, USA, 2004b. Association for Computing Machinery. ISBN 1581138628. doi:10.1145/1015467.1015471. URL <https://doi.org/10.1145/1015467.1015471>.
- Databricks. Llm inference performance engineering: Best practices. <https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices>, 2023.
- Howard David, Eugene Gorbatoov, Ulf R. Hanenbutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, 2010. doi:10.1145/1840845.1840883.

- Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. ISSN 0098-3500. doi:10.1145/2049662.2049663. URL <https://doi.org/10.1145/2049662.2049663>.
- Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.
- Sagar Dhakal, Majeed M. Hayat, Jorge E. Pezoa, Cundong Yang, and David A. Bader. Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):485–497, 2007. doi:10.1109/TPDS.2007.1009.
- Tu Mai Anh Do, Loïc Pottier, Orcun Yildiz, Karan Vahi, Patrycja Krawczuk, Tom Peterka, and Ewa Deelman. Accelerating scientific workflows on hpc platforms with in situ processing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 1–10, 2022. doi:10.1109/CCGrid54584.2022.00009.
- Daniel Ellsworth, Tapasya Patki, Swann Perarnau, Sangmin Seo, Abdelhalim Amer, Judicael Zounmevo, Rinku Gupta, Kazutomo Yoshii, Henry Hoffman, Allen Malony, Martin Schulz, and Pete Beckman. Systemwide power management with argo. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1118–1121, 2016. doi:10.1109/IPDPSW.2016.81.
- Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. Pow: System-wide dynamic reallocation of limited power in hpc. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC ’15, page 145–148, New York, NY, USA, 2015a. Association for Computing Machinery. ISBN 9781450335508. doi:10.1145/2749246.2749277. URL <https://doi.org/10.1145/2749246.2749277>.
- Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. Dynamic power sharing for higher job throughput. In *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2015b. doi:10.1145/2807591.2807643.
- Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. Highly scalable large-scale asynchronous graph processing using actors. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, pages 242–248, 2023. doi:10.1109/CCGridW59191.2023.00049.
- Nadim Elsakaan and Kamal Amroun. A novel multi-level hybrid load balancing and tasks scheduling algorithm for cloud computing environment. *J. Supercomput.*, 80(9): 13434–13474, March 2024a. ISSN 0920-8542. doi:10.1007/s11227-024-05990-5. URL <https://doi.org/10.1007/s11227-024-05990-5>.
- Nadim Elsakaan and Kamal Amroun. A novel multi-level hybrid load balancing and tasks scheduling algorithm for cloud computing environment. *J. Supercomput.*, 80(9):

13434–13474, March 2024b. ISSN 0920-8542. doi:10.1007/s11227-024-05990-5. URL <https://doi.org/10.1007/s11227-024-05990-5>.

Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

Kobra Etminani, Mahmoud Naghibzadeh, and Noorali Raeeji Yanehsari. A hybrid min-min max-min algorithm with improved performance. 2009. URL <https://api.semanticscholar.org/CorpusID:18667448>.

Alexander Fell, Yuqing Wang, Tianshuo Su, Marziyeh Nourian, Wenyi Wang, Jose M. Monsalve-Diaz, Andronicus Rajasukumar, Jiya Su, Ruiqi Xu, Rajat Khandelwal, David F. Gleich, Tianchi Zhang, Yanjing Li, Hank Hoffmann, and Andrew A. Chien. Kvmsr+udweave: Extreme-scaling with fine-grained parallelism on the updown graph supercomputer. In *Parallel Applications Workshop, Alternatives To MPI+X (PAWS-ATMSC)*, 2025.

Xixhou Feng, Rong Ge, and K.W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10 pp.–, 2005. doi:10.1109/IPDPS.2005.346.

Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08*, page 323–338, USA, 2008. USENIX Association.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 2021. URL <https://www.mpi-forum.org>.

Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, 33(2):507–541, November 2023. ISSN 1066-8888. doi:10.1007/s00778-023-00819-8. URL <https://doi.org/10.1007/s00778-023-00819-8>.

Vinicius Freitas, Laércio L. Pilla, Alexandre de L. Santana, Márcio Castro, and Johanne Cohen. Packsteallb: A scalable distributed load balancer based on work stealing and workload discretization. *Journal of Parallel and Distributed Computing*, 150:34–45, 2021. ISSN 0743-7315. doi:<https://doi.org/10.1016/j.jpdc.2020.12.005>. URL <https://www.sciencedirect.com/science/article/pii/S0743731520304214>.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223. ACM, 1998. doi:10.1145/277650.277725.

- Kaihua Fu, Quan Chen, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo. Blad: Adaptive load balanced scheduling and operator overlap pipeline for accelerating the dynamic gnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi:10.1145/3581784.3607040. URL <https://doi.org/10.1145/3581784.3607040>.
- Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank, 2024. URL <https://arxiv.org/abs/2408.15792>.
- Alexander Fuerst and Prateek Sharma. Locality-aware load-balancing for serverless clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, page 227–239, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391993. doi:10.1145/3502181.3531459. URL <https://doi.org/10.1145/3502181.3531459>.
- Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-efficient large language model serving for multi-turn conversations with cachedattention, 2024. URL <https://arxiv.org/abs/2403.19708>.
- Lei Gao, Chaoyi Jiang, Hossein Entezari Zarch, Daniel Wong, and Murali Annavaram. Duet-serve: Harmonizing prefill and decode for LLM serving via adaptive GPU multiplexing. *arXiv preprint arXiv:2511.04791*, 2025.
- R. Ge, Xizhou Feng, and K.W. Cameron. Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 34–34, 2005. doi:10.1109/SC.2005.57.
- Yiannis Georgiou, Thomas Cadeau, David Glessner, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy accounting and control with slurm resource and job management system. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, pages 96–118, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-45249-9.
- Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association. URL <https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types>.
- Neha Gholkar, Frank Mueller, and Barry Rountree. Power tuning hpc jobs on power-constrained systems. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 179–190, 2016. doi:10.1145/2967938.2967961.
- Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. Pshifter: Feedback-based dynamic power shifting within hpc jobs for performance. In *Proceedings of the*

27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '18, page 106–117, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357852. doi:10.1145/3208040.3208047. URL <https://doi.org/10.1145/3208040.3208047>.

In Gim, Zhiyao Ma, SeungSeob Lee, and Lin Zhong. Pie: A programmable serving system for emerging LLM applications. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 415–430. ACM, 2025.

Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. M\’elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

J. Octavio Gutierrez-Garcia, Joel Antonio Trejo-Sánchez, and Daniel Fajardo-Delgado. Agent coalitions for load balancing in cloud data centers. *Journal of Parallel and Distributed Computing*, 172:1–17, 2023. ISSN 0743-7315. doi:<https://doi.org/10.1016/j.jpdc.2022.10.006>. URL <https://www.sciencedirect.com/science/article/pii/S0743731522002143>.

Daniel F. Harlacher, Harald Klimach, Sabine Roller, Christian Siebert, and Felix Wolf. Dynamic load balancing for unstructured meshes on space-filling curves. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW '12*, page 1661–1669, USA, 2012. IEEE Computer Society. ISBN 9780769546766. doi:10.1109/IPDPSW.2012.207. URL <https://doi.org/10.1109/IPDPSW.2012.207>.

Wenhao He, Youhe Jiang, Penghao Zhao, Quanqing Xu, Eiko Yoneki, Bin Cui, and Fangcheng Fu. Efficient multi-round llm inference over disaggregated serving. *arXiv preprint arXiv:2602.14516*, 2026.

Zeyu He, Qiuli Huang, Zhifang Li, and Chuliang Weng. Handling data skew for aggregation in spark sql using task stealing. *Int. J. Parallel Program.*, 48(6):941–956, December 2020. ISSN 0885-7458. doi:10.1007/s10766-020-00657-z. URL <https://doi.org/10.1007/s10766-020-00657-z>.

Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, page 198–214, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi:10.1145/2815400.2815403. URL <https://doi.org/10.1145/2815400.2815403>.

Henry Hoffmann and Martina Maggio. PCP: A generalized approach to optimizing performance under power constraints through resource management. In Xiaoyun Zhu, Giuliano Casale, and Xiaohui Gu, editors, *11th International Conference on Autonomic Computing*,

- ICAC '14, Philadelphia, PA, USA, June 18-20, 2014*, pages 241–247. USENIX Association, 2014. URL <https://www.usenix.org/conference/icac14/technical-sessions/presentation/hoffman>.
- Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024. URL <https://arxiv.org/abs/2401.08671>, 2024.
- Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Yang Hu, and Minyi Guo. Ant-man: Towards agile power management in the microservice era. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020. doi:10.1109/SC41405.2020.00082.
- Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Memserve: Context caching for disaggregated llm serving with elastic memory pool, 2024. URL <https://arxiv.org/abs/2406.17565>.
- S. Huang and W. Feng. Energy-efficient cluster computing via accurate workload characterization. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 68–75, 2009. doi:10.1109/CCGRID.2009.88.
- Shengsheng Huang, Jie Huang, Yan Liu, and Jinqian Dai. Hibenach : A representative and comprehensive hadoop benchmark suite. 2012.
- Sungjin Im, Ravi Kumar, Mahshid Montazer Qaem, and Manish Purohit. Non-clairvoyant scheduling with predictions. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 285–294, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380706. doi:10.1145/3409964.3461790. URL <https://doi.org/10.1145/3409964.3461790>.
- Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. Copper: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing, ICAC 2019, Umeå, Sweden, June 16-20, 2019*, pages 31–41. IEEE, 2019. doi:10.1109/ICAC.2019.00015. URL <https://doi.org/10.1109/ICAC.2019.00015>.
- Igor N. Ivanisenko and Tamara A. Radivilova. Survey of major load balancing algorithms in distributed system. In *2015 Information Technologies in Innovation Business Conference (ITIB)*, pages 89–92, 2015. doi:10.1109/ITIB.2015.7355061.
- Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 88:50–71, 2017. ISSN 1084-8045. doi:<https://doi.org/10.1016/j.jnca.2017.04.007>. URL <https://www.sciencedirect.com/science/article/pii/S1084804517301480>.

- Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, Anoop Kulkarni, Steve Kofsky, and Saravan Rajmohan. Intelligent router for llm workloads: Improving performance through workload-aware load balancing, 2025. URL <https://arxiv.org/abs/2408.13510>.
- Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R Nutter, and Jeremy D Schaub. Tencent sort. *Technical Report*, 2016. URL <http://sortbenchmark.org/TencentSort2016.pdf>.
- Yichuan Jiang. A survey of task allocation and load balancing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):585–599, Feb 2016a. ISSN 1558-2183. doi:10.1109/TPDS.2015.2407900.
- Yichuan Jiang. A survey of task allocation and load balancing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):585–599, 2016b. doi:10.1109/TPDS.2015.2407900.
- Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of large language model over heterogeneous environment. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 21946–21961. PMLR / OpenReview.net, 2024.
- Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. Thunderserve: High-performance and cost-efficient LLM serving in cloud environments. In Matei Zaharia, Gauri Joshi, and Yingyan (Celine) Lin, editors, *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. OpenReview.net/mlsys.org, 2025a.
- Youhe Jiang, Ran Yan, and Binhang Yuan. Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025b.
- Yuxuan Jiang, Zhe Huang, and Danny H.K. Tsang. Towards max-min fair resource allocation for stream big data analytics in shared clouds. *IEEE Transactions on Big Data*, 4(1):130–137, 2018. doi:10.1109/TBDATA.2016.2638860.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL <https://arxiv.org/abs/2310.06770>.
- Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad,

- Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. Hpx - the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352, 2020. doi:10.21105/joss.02352. URL <https://doi.org/10.21105/joss.02352>.
- Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 91–108, New York, NY, USA, 1993a. Association for Computing Machinery. ISBN 0897915879. doi:10.1145/165854.165874. URL <https://doi.org/10.1145/165854.165874>.
- Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, October 1993b. ISSN 0362-1340. doi:10.1145/167962.165874. URL <https://doi.org/10.1145/167962.165874>.
- Dong-Ki Kang, Yun-Gi Ha, Limei Peng, and Chan-Hyun Youn. Cooperative distributed gpu power capping for deep learning clusters. *IEEE Transactions on Industrial Electronics*, 69(7):7244–7254, 2022. doi:10.1109/TIE.2021.3095790.
- Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Collieran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 219–233. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL <https://www.usenix.org/conference/atc20/presentation/keahey>.
- Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018. ISSN 2376-3639. doi:10.1145/3177754. URL <https://doi.org/10.1145/3177754>.
- Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 169–182, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319942. doi:10.1145/2465351.2465369. URL <https://doi.org/10.1145/2465351.2465369>.
- Yuetsu Kodama, Tetsuya Odajima, Eishi Arima, and Mitsuhsa Sato. Evaluation of power management control on the supercomputer fugaku. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 484–493, 2020. doi:10.1109/CLUSTER49012.2020.00069.
- Boonhatai Kruekaew and Warangkhan Kimpan. Multi-objective task scheduling optimization for load balancing in cloud computing environment using hybrid artificial bee colony algorithm with reinforcement learning. *IEEE Access*, 10:17803–17818, 2022. doi:10.1109/ACCESS.2022.3149955.

- Neetesh Kumar, Deo Prakash Vidyarthi, and Antonio Fernandez Anta. A hybrid heuristic for load-balanced scheduling of heterogeneous workload on heterogeneous systems. *The Computer Journal*, 62(2):276–291, 2019. doi:10.1093/comjnl/bxy085.
- Ravi Kumar, Manish Purohit, and Zoya Svitkina. Improving online algorithms via ml predictions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS’18, page 9684–9693, Red Hook, NY, USA, 2018. Curran Associates Inc.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.
- YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC ’10, page 75–86, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300360. doi:10.1145/1807128.1807140. URL <https://doi.org/10.1145/1807128.1807140>.
- S Labasan, M Larsen, B Rountree, and H Childs. Paviz: A power-adaptive framework for optimal power and performance of scientific visualization algorithms. 3 2017. URL <https://www.osti.gov/biblio/1366964>.
- Ruiqi Lai, Hongrui Liu, Chengzhi Lu, Zonghao Liu, Siyu Cao, Siyang Shao, Yixin Zhang, Luo Mai, and Dmitrii Ustiugov. Tokenscale: Timely and accurate autoscaling for disaggregated LLM serving with token velocity. *arXiv preprint arXiv:2512.03416*, 2025.
- Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoeffler, and Fabrizio Petrini. Polarstar: Expanding the horizon of diameter-3 networks. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’24, page 345–357, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704161. doi:10.1145/3626183.3659975. URL <https://doi.org/10.1145/3626183.3659975>.
- Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Online scheduling via learned weights. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’20, page 1859–1877, USA, 2020. Society for Industrial and Applied Mathematics.
- Savoie Lee, David K. Lowenthal, Bronis R. De Supinski, Tanzima Islam, Kathryn Mohror, Barry Rountree, and Martin Schulz. I/o aware power shifting. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 740–749, 2016. doi:10.1109/IPDPS.2016.15.

- Youngsoo Lee and Sunghee Choi. A greedy load balancing algorithm for faas platforms. In *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing*, ICCBDC '21, page 63–69, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390408. doi:10.1145/3481646.3481657. URL <https://doi.org/10.1145/3481646.3481657>.
- Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- Scott Levy, Kurt B. Ferreira, Patrick Widener, Patrick G. Bridges, and Oscar H. Mondragon. How i learned to stop worrying and love in situ analytics: Leveraging latent synchronization in mpi collective algorithms. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI '16, page 140–153, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342346. doi:10.1145/2966884.2966920. URL <https://doi.org/10.1145/2966884.2966920>.
- Hanchen Li, Runyuan He, Qiuyang Mang, Qizheng Zhang, Huanzhi Mao, Xiaokun Chen, Hangrui Zhou, Alvin Cheung, Joseph Gonzalez, and Ion Stoica. Continuum: Efficient and robust multi-turn llm agent scheduling with kv cache time-to-live. *arXiv preprint arXiv:2511.02230*, 2025.
- Xiao Li, Ming-chun Zheng, Xinxin Ren, Xuan Liu, Panpan Zhang, and Chao Lou. An improved task scheduling algorithm based on potential games in cloud computing. pages 346–355, 01 2014. ISBN 978-3-319-09264-5. doi:10.1007/978-3-319-09265-2_35.
- Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation, 2024. URL <https://arxiv.org/abs/2404.14469>.
- Zongze Li, Jingyu Liu, Zhen Xu, Yineng Zhang, Tahseen Rabbani, and Ce Zhang. Not all prefills are equal: Ppd disaggregation for multi-turn llm serving, 2026. URL <https://arxiv.org/abs/2603.13358>.
- Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, page 137–148, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450308052. doi:10.1145/2287076.2287103. URL <https://doi.org/10.1145/2287076.2287103>.
- Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 929–945. USENIX Association, 2024.

- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024a.
- Xunyun Liu and Rajkumar Buyya. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Comput. Surv.*, 53(3), May 2020a. ISSN 0360-0300. doi:10.1145/3355399. URL <https://doi.org/10.1145/3355399>.
- Xunyun Liu and Rajkumar Buyya. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. *ACM Comput. Surv.*, 53(3), May 2020b. ISSN 0360-0300. doi:10.1145/3355399. URL <https://doi.org/10.1145/3355399>.
- Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*, pages 38–56. ACM, 2024b.
- Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for llm agents as general programs, 2025. URL <https://arxiv.org/abs/2502.13965>.
- Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun. Understanding parallelism in graph traversal on multi-core clusters. *Comput. Sci.*, 28(2–3):193–201, May 2013. ISSN 1865-2034. doi:10.1007/s00450-012-0207-3. URL <https://doi.org/10.1007/s00450-012-0207-3>.
- Thodoris Lykouris and Sergei Vassilvtiskii. Competitive caching with machine learned advice. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3296–3305. PMLR, 10–15 Jul 2018. URL <https://proceedings.mlr.press/v80/lykouris18a.html>.
- Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: Efficient isolated execution of hybrid oltp + olap workloads. In *SIGMOD*, 2017.
- Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. A run-time system for power-constrained hpc applications. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, pages 394–408, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20119-1.

- Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Ghaleb Abdulla, and Barry Rountree. An empirical survey of performance and energy efficiency variation on intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351324. doi:10.1145/3149412.3149421. URL <https://doi.org/10.1145/3149412.3149421>.
- Alessandro Margara, Gianpaolo Cugola, Nicolò Felicioni, and Stefano Cilloni. A model and survey of distributed data-intensive systems. *ACM Comput. Surv.*, 56(1), August 2023. ISSN 0360-0300. doi:10.1145/3604801. URL <https://doi.org/10.1145/3604801>.
- Ivana Marincic, Venkatram Vishwanath, and Henry Hoffmann. Seesaw: Optimizing performance of in-situ analytics applications under power constraints. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 789–798. IEEE, 2020. doi:10.1109/IPDPS47924.2020.00086. URL <https://doi.org/10.1109/IPDPS47924.2020.00086>.
- Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37, June 2021. ISSN 0163-5980. doi:10.1145/3469379.3469383. URL <https://doi-org.proxy.uchicago.edu/10.1145/3469379.3469383>.
- Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 586–602. ACM, 2025.
- D. Meisner, C.M. Sadler, Luiz Barroso, W. Weber, and Thomas Wenisch. Power management of online data-intensive services. pages 319–330, 01 2011. ISBN 978-1-4503-0472-6. doi:10.1145/2000064.2000103.
- Paul Messina. The usdoe exascale computing project—goals and challenges, 2017.
- Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. *SIGPLAN Not.*, 50(4):267–281, mar 2015. ISSN 0362-1340. doi:10.1145/2775054.2694373. URL <https://doi.org/10.1145/2775054.2694373>.
- Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions, 2020. URL <https://arxiv.org/abs/2006.09123>.
- David Mizell and Kristyn Maschhoff. Early experiences with large-scale cray xmt systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–9, 2009. doi:10.1109/IPDPS.2009.5161108.

- Abdul Sajid Mohammed, Venkata Ramana Saddi, Santhosh Kumar Gopal, Nasmin Jiwani, and J. Logeshwaran. Dynamic scheduling algorithms for serverless computing solutions in the cloud. In *2024 International Conference on E-mobility, Power Control and Smart Systems (ICEMPS)*, pages 1–7, 2024. doi:10.1109/ICEMPS60684.2024.10559356.
- Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, page 422–431, USA, 1993. Society for Industrial and Applied Mathematics. ISBN 0898713137.
- Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi:10.1145/2517349.2522738. URL <https://doi.org/10.1145/2517349.2522738>.
- Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, 2017.
- P. Nehra and Nishtha Kesswani. Efficient resource allocation and management by using load balanced multi-dimensional bin packing heuristic in cloud data centers. *J. Supercomput.*, 79(2):1398–1425, July 2022. ISSN 0920-8542. doi:10.1007/s11227-022-04707-w. URL <https://doi.org/10.1007/s11227-022-04707-w>.
- NVIDIA. NVIDIA Dynamo: A low-latency distributed inference framework for scaling reasoning AI models. <https://developer.nvidia.com/dynamo>, 2025.
- NVIDIA Corporation. *CUDA C Programming Guide*. Accessed: Apr. 18, 2023.
- NVIDIA Corporation. Nvidia h100 tensor core gpu datasheet, 2022. URL <https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet>. Accessed: 2024-11-01.
- NVIDIA Corporation. Nvidia blackwell architecture overview, 2024. URL <https://resources.nvidia.com/en-us-blackwell-architecture>. Accessed: 2024-11-01.
- Girish Palshikar et al. Simple algorithms for peak detection in time-series. In *Proc. 1st Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*, volume 122, 2009.
- Zaifeng Pan, Ajikumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows, 2025. URL <https://arxiv.org/abs/2507.07400>.
- Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, pages 118–132. IEEE, 2024.

- Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 173–182, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321303. doi:10.1145/2464996.2465009. URL <https://doi.org/10.1145/2464996.2465009>.
- Tapasya Patki, David K. Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis R. de Supinski. Practical resource management in power-constrained, high performance computing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 121–132, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335508. doi:10.1145/2749246.2749262. URL <https://doi.org/10.1145/2749246.2749262>.
- Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 149–161, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336185. doi:10.1145/2814576.2814808. URL <https://doi.org/10.1145/2814576.2814808>.
- You Peng, Youhe Jiang, Wenqi Jiang, Chen Wang, and Binhang Yuan. Hexgen-flow: Optimizing llm inference request scheduling for agentic text-to-sql. *arXiv preprint arXiv:2505.05286*, 2025.
- Swann Perarnau, Rajeev Thakur, Kamil Iskra, Ken Raffanetti, Franck Cappello, Rinku Gupta, Pete Beckman, Marc Snir, Henry Hoffmann, Martin Schulz, and Barry Rountree. Distributed monitoring and management of exascale systems in the argo project. In Alysson Bessani and Sara Bouchenak, editors, *Distributed Applications and Interoperable Systems*, pages 173–178, Cham, 2015. Springer International Publishing. ISBN 978-3-319-19129-4.
- P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, 2006. doi:10.1109/ICDE.2006.105.
- Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Cham, 6th edition, 2022. ISBN 978-3-031-05920-9. doi:10.1007/978-3-031-05921-6.
- Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In Dawn Song, Michael Carbin, and Tianqi Chen, editors, *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*. mlsys.org, 2023.
- Borja Pérez, E. Stafford, J.L. Bosque, and R. Beivide. Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. *Journal of Parallel and Distributed Computing*,

- 157:30–42, 2021. ISSN 0743-7315. doi:<https://doi.org/10.1016/j.jpdc.2021.06.003>. URL <https://www.sciencedirect.com/science/article/pii/S0743731521001325>.
- Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation - A kvcache-centric architecture for serving LLM chatbot. In Haryadi S. Gunawi and Vasily Tarasov, editors, *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, pages 155–170. USENIX Association, 2025.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.
- Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.
- Andronicus Rajasukumar, Jiya Su, Yuqing, Wang, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Mubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. Updown: Programmable fine-grained events for scalable performance on irregular applications, 2024a. URL <https://arxiv.org/abs/2407.20773>.
- Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A. Chien. Updown: A novel architecture for unlimited memory parallelism. In *Proceedings of the International Symposium on Memory Systems, MEMSYS '24*, page 61–77, New York, NY, USA, 2024b. Association for Computing Machinery. ISBN 9798400710919. doi:10.1145/3695794.3695801. URL <https://doi.org/10.1145/3695794.3695801>.
- Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12*, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450317610. doi:10.1145/2391229.2391245. URL <https://doi.org/10.1145/2391229.2391245>.
- Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: sharing-aware component composition for distributed stream processing systems. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, page 322–341, Berlin, Heidelberg, 2006a. Springer-Verlag.

- Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In Maarten van Steen and Michi Henning, editors, *Middleware 2006*, pages 322–341, Berlin, Heidelberg, 2006b. Springer Berlin Heidelberg. ISBN 978-3-540-68256-1.
- Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Ai and ml accelerator survey and trends. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, 2022. doi:10.1109/HPEC55821.2022.9926331.
- Toran Bruce Richards. Autogpt, 2023. URL <https://github.com/Significant-Gravitas/AutoGPT>.
- Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Providing qos guarantees in large-scale operator networks. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 337–345, 2010. doi:10.1109/HPCC.2010.53.
- Stamatia Rizou, Frank Diirr, and Kurt Rothermel. Fulfilling end-to-end latency constraints in large-scale streaming environments. In *30th IEEE International Performance Computing and Communications Conference*, pages 1–8, 2011. doi:10.1109/PCCC.2011.6108086.
- Henriette Röger and Ruben Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2), April 2019. ISSN 0360-0300. doi:10.1145/3303849. URL <https://doi.org/10.1145/3303849>.
- Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS ’09, page 460–469, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584980. doi:10.1145/1542275.1542340. URL <https://doi.org/10.1145/1542275.1542340>.
- Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys ’11, page 139–152, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306348. doi:10.1145/1966445.1966459. URL <https://doi.org/10.1145/1966445.1966459>.
- Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’20, page 497–511, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi:10.1145/3373376.3378533. URL <https://doi.org/10.1145/3373376.3378533>.

- Ryuichi Sakamoto, Thang Cao, Masaaki Kondo, Koji Inoue, Masatsugu Ueda, Tapasya Patki, Daniel Ellsworth, Barry Rountree, and Martin Schulz. Production hardware overprovisioning: Real-world performance optimization using an extensible power-aware resource management framework. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 957–966, 2017. doi:10.1109/IPDPS.2017.107.
- R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 111–, 2004. doi:10.1109/IPDPS.2004.1303065.
- Osman Sarood, Akhil Langer, Laxmikant Kalé, Barry Rountree, and Bronis de Supinski. Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013. doi:10.1109/CLUSTER.2013.6702684.
- Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818, 2014. doi:10.1109/SC.2014.71.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.
- Donatien Schmitz, Luc Berrewaerts, Guillaume Rosinosky, Sabri Skhiri, and Etienne Rivière. A tale of many streams: Characterizing a hybrid batch-stream production workload in digazu. In *DEBS (Distributed Event-based Systems)*. ACM, 2025. doi:10.1145/3701717.3734462.
- Scott Schneider and Kun-Lung Wu. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. *SIGPLAN Not.*, 52(6):648–661, June 2017. ISSN 0362-1340. doi:10.1145/3140587.3062366. URL <https://doi.org/10.1145/3140587.3062366>.
- Jianshu She, Zonghang Li, Hongchao Du, Shangyu Wu, Wenhao Zheng, Eric Xing, Zhengzhong Liu, Huaxiu Yao, Jason Xue, and Qirong Ho. LAPS: A length-aware-prefill LLM serving system. *arXiv preprint arXiv:2601.11589*, 2026.
- Kai Shen, Arrvinth Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. *SIGPLAN Not.*, 48(4):65–76, mar 2013a. ISSN 0362-1340. doi:10.1145/2499368.2451124. URL <https://doi.org/10.1145/2499368.2451124>.
- Kai Shen, Arrvinth Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore

- servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 65–76, New York, NY, USA, 2013b. Association for Computing Machinery. ISBN 9781450318709. doi:10.1145/2451116.2451124. URL <https://doi.org/10.1145/2451116.2451124>.
- Tianyao Shi, Yanran Wu, Sihang Liu, and Yi Ding. Greenllm: Disaggregating large language model serving on heterogeneous gpus for lower carbon emissions, 2024. URL <https://arxiv.org/abs/2412.20322>.
- Xiaoxiang Shi, Colin Cai, Junjia Du, and Zhihao Jia. Nexus: Proactive intra-GPU disaggregation of prefill and decode in LLM serving. *arXiv preprint arXiv:2507.06608*, 2025.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.
- Martin Simonovsky and Nikos Komodakis. Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs . In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 29–38, Los Alamitos, CA, USA, July 2017. IEEE Computer Society. doi:10.1109/CVPR.2017.11. URL <https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.11>.
- Tinku Singh, Shivam Gupta, Satakshi, and Manish Kumar. Adaptive load balancing in cluster computing environment. *The Journal of Supercomputing*, 79(6):1–20, 2023. doi:10.1007/s11227-023-05434-6. URL <https://link.springer.com/article/10.1007/s11227-023-05434-6>.
- David Snowdon, Etienne Sueur, Stefan Petters, and Gernot Heiser. Koala a platform for os-level power management. pages 289–302, 01 2009. doi:10.1145/1519065.1519097.
- Tapan Srivastava, Huazhe Zhang, and Henry Hoffmann. Penelope: Peer-to-peer power management. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP '22*, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450397339. doi:10.1145/3545008.3545047. URL <https://doi.org/10.1145/3545008.3545047>.
- Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, page 1348–1362. IEEE, March 2025. doi:10.1109/hpca61900.2025.00102. URL <http://dx.doi.org/10.1109/HPCA61900.2025.00102>.
- Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 24), pages 173–191. USENIX Association, 2024.

Dandan Sun, Min Wu, Zhiqiang Yang, et al. A two-tier coordinated load balancing strategy over skewed data streams. *The Journal of Supercomputing*, 79:21028–21056, 2023. doi:10.1007/s11227-023-05473-z. URL <https://doi.org/10.1007/s11227-023-05473-z>.

Dawei Sun, Guangyan Zhang, Songlin Yang, Weimin Zheng, Samee U. Khan, and Keqin Li. Re-stream. *Inf. Sci.*, 319(C):92–112, October 2015. ISSN 0020-0255. doi:10.1016/j.ins.2015.03.027. URL <https://doi.org/10.1016/j.ins.2015.03.027>.

Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Education, 2nd edition, 2007. ISBN 9780132392273.

Thomas N. Theis and H.-S. Philip Wong. The end of moore’s law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017. doi:10.1109/MCSE.2017.29.

Ibrahim Thiyeb and Sharaf Alhomdy. Hamm: A hybrid algorithm of min-min and max-min task scheduling algorithms in cloud computing. volume 9, 11 2020. doi:10.35940/ijrte.D4874.119420.

Zhao Tong, Xiaomei Deng, Hongjian Chen, and Jing Mei. Ddmts: A novel dynamic load balancing scheduling scheme under sla constraints in cloud computing. *Journal of Parallel and Distributed Computing*, 149:138–148, 2021. ISSN 0743-7315. doi:<https://doi.org/10.1016/j.jpdc.2020.11.007>. URL <https://www.sciencedirect.com/science/article/pii/S0743731520304147>.

Vibhore Vardhan, Wanghong Yuan, Albert III, Sarita Adve, Robin Kravets, Klara Nahrstedt, Daniel Sachs, and Douglas Jones. Grace-2: Integrating fine-grained application adaptation with global adaptation for saving energy. *IJES*, 4:152–169, 01 2009. doi:10.1504/IJES.2009.027939.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. *SIGARCH Comput. Archit. News*, 38(1): 205–218, mar 2010a. ISSN 0163-5964. doi:10.1145/1735970.1736044. URL <https://doi.org/10.1145/1735970.1736044>.

- Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 205–218, New York, NY, USA, 2010b. Association for Computing Machinery. ISBN 9781605588391. doi:10.1145/1736020.1736044. URL <https://doi.org/10.1145/1736020.1736044>.
- Espen Volnes, Thomas Plagemann, and Vera Goebel. To migrate or not to migrate: An analysis of operator migration in distributed stream processing. *IEEE Communications Surveys & Tutorials*, 26(1):670–705, 2024. doi:10.1109/COMST.2023.3330953.
- Chao Wang, Pengfei Zuo, Zhangyu Chen, Yunkai Liang, Zhou Yu, and Ming-Chang Yang. Prefill-decode aggregation or disaggregation? unifying both for goodput-optimized llm serving, 2025. URL <https://arxiv.org/abs/2508.01989>.
- Xiaorui Wang and Ming Chen. Cluster-level feedback power control for performance optimization. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 101–110, 2008. doi:10.1109/HPCA.2008.4658631.
- Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. Sol: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’22, page 622–634, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450392051. doi:10.1145/3503222.3507704. URL <https://doi.org/10.1145/3503222.3507704>.
- Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. Sol: safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’22, page 622–634, New York, NY, USA, 2022b. Association for Computing Machinery. ISBN 9781450392051. doi:10.1145/3503222.3507704. URL <https://doi.org/10.1145/3503222.3507704>.
- Yuqing Wang, Andronicus Rajasukumar, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Ahsan Pervaiz, Jerry Ding, Charles Colley, Wenyi Wang, Yanjing Li, et al. Efficiently exploiting irregular parallelism using keys at scale. In *2023 Workshop on Languages and Compilers for Parallel Computing*, 2023.
- Zhikui Wang, Cliff McCarthy, Xiaoyun Zhu, Partha Ranganathan, and Vanish Talwar. Feedback control algorithms for power management of servers. 01 2008.
- Andreas Weissel, Björn Beutel, and Frank Bellosa. Cooperative I/O: A novel I/O semantics for Energy-Aware applications. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, Boston, MA, December 2002. USENIX Association. URL <https://www.usenix.org/conference/osdi-02/cooperative-io-novel-io-semantics-energy-aware-applications>.

- Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. 1995.
- Will Whiteside, Shelby Funk, Aniruddha Marathe, and Barry Rountree. Pann: Power allocation via neural networks dynamic bounded-power allocation in high performance computing. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351324. doi:10.1145/3149412.3149420. URL <https://doi.org/10.1145/3149412.3149420>.
- Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544, 2014. doi:10.1109/ICDCS.2014.61.
- Jingyi Yang, Songsong Mo, Jiachen Shi, Zihao Yu, Kunhao Shi, Xuchen Ding, and Gao Cong. Arcade: A real-time data system for hybrid and continuous query processing across diverse data modalities. *arXiv preprint arXiv:2509.19757*, 2025.
- Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 94–109. ACM, 2025.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.
- Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39727-4.
- Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 521–538. USENIX Association, 2022.
- Jiadong Yu, Haopeng Chen, and Fei Hu. Sasm: Improving spark performance with adaptive skew mitigation. In *2015 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 102–107, 2015. doi:10.1109/PIC.2015.7489818.
- Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, page 149–163, New York, NY, USA, 2003. Asso-

ciation for Computing Machinery. ISBN 1581137575. doi:10.1145/945445.945460. URL <https://doi.org/10.1145/945445.945460>.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, volume 10, 2010.

Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Blitzscale: Fast and live large model autoscaling with $o(1)$ host caching. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 275–293. USENIX Association, 2025.

Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 545–559, New York, NY, USA, 2016a. Association for Computing Machinery. ISBN 9781450340915. doi:10.1145/2872362.2872375. URL <https://doi.org/10.1145/2872362.2872375>.

Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *SIGARCH Comput. Archit. News*, 44(2):545–559, mar 2016b. ISSN 0163-5964. doi:10.1145/2980024.2872375. URL <https://doi.org/10.1145/2980024.2872375>.

Huazhe Zhang and Henry Hoffmann. Performance & energy tradeoffs for dependent distributed applications under system-wide power caps. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365109. doi:10.1145/3225058.3225098. URL <https://doi.org/10.1145/3225058.3225098>.

Huazhe Zhang and Henry Hoffmann. Podd: Power-capping dependent distributed applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi:10.1145/3295500.3356174. URL <https://doi.org/10.1145/3295500.3356174>.

Jing Zhang, Chunlin Li, Liye Zhu, and Yanpei Liu. The real-time scheduling strategy based on traffic and load balancing in storm. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 372–379, 2016. doi:10.1109/HPCC-SmartCity-DSS.2016.0060.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark W. Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs. In

Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 193–210. USENIX Association, 2024.

Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *Proceedings of the 2006 Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I*, ODBASE’06/OTM’06, page 54–71, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540482873. doi:10.1007/11914853_5. URL https://doi.org/10.1007/11914853_5.