Programmable and Adaptive Scheduling for Distributed Systems

Yuyao Wang Xiangfeng Zhu Ratul Mahajan Stephanie Wang

University of Washington

Abstract

Existing frameworks for managing distributed systems hard-code scheduling policies and their implementations (e.g., centralized vs. decentralized), limiting customization and hurting performance across diverse applications and workloads. We argue for an adaptive scheduling approach, where developers express policies in a high-level, framework-agnostic DSL, and a compiler generates optimized implementations based on policy semantics, workload characteristics, and execution environments. We demonstrate that our compiler-guided approach can significantly improve both scheduling quality and performance.

CCS Concepts

• **Networks** → **Programming interfaces**; **Network resources allocation**.

Keywords

Resource Scheduling, Distributed Systems

ACM Reference Format:

Yuyao Wang, Xiangfeng Zhu, Ratul Mahajan, Stephanie Wang. 2025. Programmable and Adaptive Scheduling for Distributed Systems. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772356.3772391

1 Introduction

Distributed computing has become the dominant paradigm for large-scale applications from LLM training and inference to data processing and video analysis [4, 9, 11, 36]. Many such applications decompose work into fine-grained (sub-second or sub-100-millisecond [28]) tasks, driven by paradigms such as serverless computing, microservices, and real-time streaming processing [19, 35]. Consequently, efficient and high-quality scheduling, i.e., dispatching tasks to nodes, is crucial for application performance [29].

At the same time, applications are becoming more diverse and have disparate scheduling concerns. A video analysis application may want to de-prioritize the processing of low-priority frames [22], which can be implemented via task queue ordering heuristics, whereas LLM serving may want locality between tasks and nodes to reduce KV cache recomputation [37]. To efficiently serve such different needs, developers must be able to customize scheduling [1].

However, today's frameworks for running distributed applications support limited customization of scheduling. They offer a set of predefined scheduling policies with low-level knobs and they hard-code an implementation approach (e.g., centralized or decentralized). For example, Ray [25] allows developers to provide hints that help select one of the available scheduling policies, and it uses a decentralized implementation with one scheduler instance per node for scalability [5]; Kubernetes has a fixed policy decision workflow which developers can influence at certain extension points using Go plugins, and it uses a centralized implementation to ensure strong consistency of scheduling information [2, 3]. This state of affairs is problematic for two reasons.

Limited policy expressiveness. Many policies of interest cannot be expressed in current frameworks. For instance, most schedulers follow a sequential procedure of selecting a task first and then choosing the node for that task. If task performance depends on the node, this procedure cannot implement a policy that aims to maximize throughput by minimizing the time until the next task completion because that requires reasoning about combinations of tasks and nodes. When developers cannot express desired policies, they have to implement their own schedulers from scratch (or heavily modify the framework), leading to substantial engineering burden [6, 7, 23].

Poor application performance. Application performance depends on the speed and quality of scheduling decisions, which in turn depend on how the scheduling is implemented. Sub-optimal implementations can hurt scheduling speed when a scheduler instance cannot make decisions fast enough, and it can hurt quality when the decisions are based on

HotNets '25, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/25/11

https://doi.org/10.1145/3772356.3772391

stale information. Unfortunately, the optimal implementation depends on several factors, including policy semantics (e.g., whether the policy uses global information), deployment variables such as number of nodes and inter-node latency (which determines the cost of gathering information about nodes), and incoming task rate (which determines how many decisions per second need to be made). Implementation choices go beyond centralized or distributed and include how sub-modules are implemented (e.g., information on nodes' current load can be gathered via periodic heartbeats or via probing). Hard-coded implementations of current frameworks do not adapt to such dynamic, application-specific factors, which can degrade application performance.

Both problems above stem from the fact that today's schedulers are implemented in a low-level, framework-specific way. We argue that scheduling policy specifications should be decoupled from implementations, and argue for a different approach based on two principles:

(1) **High-level programmability.** Developers should be able to express application-specific scheduling policies in a concise, framework-agnostic manner.
(2) **Adaptive execution.** The system should automatically adjust the implementation strategy based on current conditions to optimize performance.

We propose a language- and compiler-based approach in which developers specify scheduling policies in a high-level DSL (domain-specific language) and the compiler generates an optimized implementation that can be dynamically plugged into application frameworks. The implementation adapts to policy semantics, workload characteristics, and deployment environments.

We outline ideas on how to realize this approach. Our DSL allows developers to specify task and node scoring policies as well as scheduling state updates throughout the task lifecycle (arrival, dispatch, completion, etc.), which offers greater flexibility over prior declarative abstractions [31] and enables a richer space for optimization. To deploy the policy, the compiler searches for an implementation plan that (1) satisfies throughput requirements, (2) minimizes scheduling overhead, and (3) bounds the quality gap to an ideal, omniscient scheduler within a developer-acceptable range. Preliminary experiments validate core aspects of our design.

2 Motivating Example: DRF-LB

To illustrate why today's rigid scheduling frameworks fall short, consider a simple example policy: *DRF-LB*, which combines Dominant Resource Fairness [12] (DRF) with load balancing (LB). It aims for fair resource allocation across tenants while also distributing tasks evenly across the cluster. As shown in Algorithm 1, each task t contains its submitter tenant $t.tenant$ and its required execution resource vector $t.rvec$.

Algorithm 1 Dominant Resource Fairness + Load Balancing Scheduling Policy (DRF-LB)

```
1:  function DominantResource(task)
2:      rvec ← ResourceVector[task.tenant]
3:      return max_r{rvec[r]/capacity[r]}
4:  end function
5:  function Schedule
6:      Sleep until any of the following cases happen:
7:      case ∃ pending tasks:
8:          Pick the task t with the smallest DominantResource(t) (break ties according to arrival time)
9:          Pick the least-loaded node n and assign t to n
10:         ResourceVector[t.tenant] += t.rvec
11:     case task t completed:
12:         ResourceVector[t.tenant] -= t.rvec
13: end function
```

The scheduling policy consists of two key pieces of logic: *(1) DRF:* DominantResource computes a tenant's dominant resource share as the maximum ratio of its resource usage to the corresponding cluster capacity across all resource types, which is called in Line 8 to pick the task whose submitter has the smallest share. *(2) LB:* Line 9 probes current load across nodes and places the task on the least-loaded one. To ensure real-time fairness, the per-tenant resource usage needs to be updated upon task dispatching (Line 10) and completion (Line 12).

Although the policy only involves simple computations and state updates, developers are unable to easily implement *DRF-LB*. Distributed systems schedulers that offer a menu of pre-defined policies, such as Ray, do not include it as an option, and implementing it in schedulers that offer low-level knobs, such as Kubernetes, would also be difficult because they do not offer an extension point for customized logic on task completion.

Even if developers are willing to invest engineering effort in writing their policy from scratch, they would be limited to the scheduling architecture of their target framework. The two most common architectures today are centralized [2, 13, 17] and decentralized [5, 29, 30] architectures. In centralized systems (Figure 1a), all tasks go through a single scheduler in the control plane, while in decentralized systems (Figure 1b), incoming tasks are load-balanced across multiple scheduling instances which make independent scheduling decisions and optionally rely on a centralized store for synchronizing state [8, 30].

Importantly, the right choice of the architecture depends on the application and its workload. For *DRF-LB*, centralized scheduling offers strong fairness guarantees—ensuring tenants receive their fair share of resources—but becomes

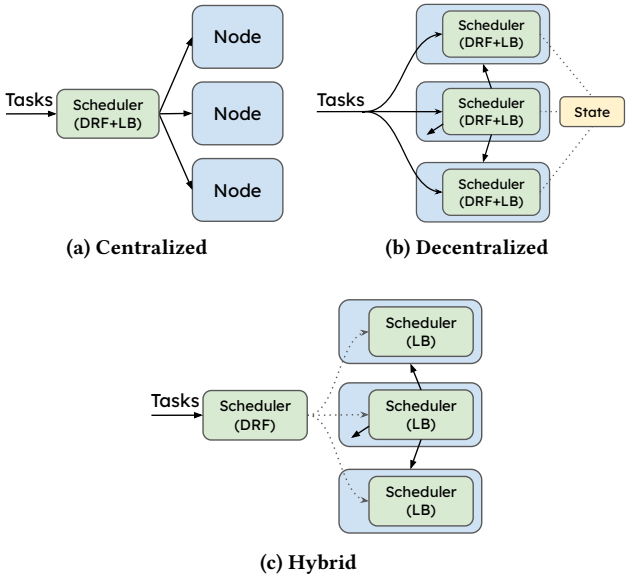


Figure 1: Scheduling Architectures. Solid arrows stand for task submission and dispatching, and dashed arrows/lines stand for internal communications during scheduling.

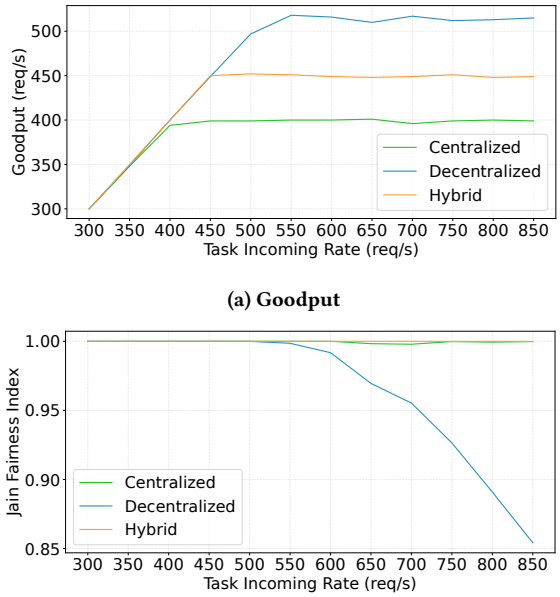


Figure 2: Implementation Comparisons

a bottleneck under high task arrival rates. Decentralized scheduling scales better, but can compromise fairness (and even cause starvation in extreme cases) because (1) each scheduler instance is unaware of the potentially existing task candidates with higher priority on other schedulers, and (2) local dominant shares are not up-to-date and thus cannot accurately reflect the real-time resource allocation ranking among tenants. Existing systems do not adaptively choose the appropriate architecture; they implement one, which all applications must use at all times.

In fact, there exists another architecture that is promising for some applications and policies but is not implemented by existing frameworks. Observe that *DRF-LB* has two pieces of logic with varying characteristics: (1) the DRF logic (DRF) requires global task information and is thus highly sensitive to architecture choice–executing it in a centralized manner is desirable; (2) the load balancing logic (LB) is slow as it probes the current load across nodes, especially when the cluster is large and the network latency is high, but its effectiveness is independent of the architecture. Thus, we can split the logic and do scheduling in a hybrid manner, as shown in Figure 1c: the centralized scheduler takes in all the submitted tasks, executes the DRF logic to pick the next task to be scheduled, and forwards the task to one of the decentralized schedulers, which executes the load balancing logic. In this way, part of the logic on the critical path is still distributed to increase scalability, and the quality of fairness enforcement is ensured.

We demonstrate empirically that the scheduling architecture matters. We simulate different implementations of *DRF-LB* under various task arrival rates and measure the resulting goodput and fairness. Decentralized scheduling uses a Redis store for periodic synchronization. We construct a multi-tenant workload with 3 active tenants of equal priority, among which the "greedy" tenant's task submission rate is 3 times higher than the others. The cluster has 12 nodes, each with a queue holding incoming tasks and a single-threaded executor running tasks in FIFO order. We also implement a simple overload control mechanism that drops timed-out tasks.

Figure 2a compares the goodput of different architectures. As expected, decentralized architecture has the best scalability, whose ~510 req/s goodput is bottlenecked by task execution time. Hybrid architecture has higher goodput than the centralized version as part of the scheduling logic is distributed, but neither can fully utilize the computation resources (i.e., nodes) under high load.

Figure 2b compares the fairness using the Jain Fairness Index [18] between tenants on their service ratio [27], i.e., the ratio between their actual and expected goodput according to max-min fair share. The decentralized architecture, although producing a larger total goodput, fails to enforce fairness

Rate	Goodput	Fairness	Best Architecture
<400	C = D = H	C = D = H	Any (C preferred)
400-450	D = H > C	C = D = H	D, H
450-510	D > H > C	C = D = H	D
>510	D > H > C	C = H > D	H

Table 1: Best architecture. Each column shows the comparison between implementations from a specific perspective. C, D, and H stand for Centralized, Decentralized, and Hybrid implementations, respectively.

between tenants. When the task arrival rate is high, non-greedy tenants suffer. In contrast, the centralized and hybrid architectures are fair across the workload regime.

Table 1 summarizes the conclusions from our experiment. When the load is low, all architectures behave similarly, and the centralized may be preferred because of its simplicity. When the workload increases and scheduling becomes the system bottleneck, decentralized implementation outperforms in scalability. However, when there exists significant resource contention, the hybrid implementation should be selected to ensure fairness.

While our experiment focuses on the trade-off between scheduling architectures, the space of implementation optimization goes further. Other opportunities include trading operation accuracy for speed (e.g., rely on cached information instead of remote probing) or parameter tuning, and whether they are effective is policy- and workload-dependent. These observations highlight the need for a new scheduling paradigm adaptive to dynamic environments.

3 Rethinking Scheduling for Distributed Systems

To overcome the limitations of today's hard-coded, one-size-fits-all schedulers, we argue for the decoupling of scheduling policies and implementations, and propose a new scheduling approach that supports customized policies and adaptive implementations. The framework provides a high-level, framework-agnostic DSL that enables developers to express decision-making and state update logic. A compiler then determines how to realize the scheduling policy in the deployment environment. The exact implementation strategy depends on three key factors: (1) the semantics of the scheduling policy, such as whether it requires global state, is compute-intensive, or benefits from parallelism; (2) workload characteristics, including task arrival rate, burstiness, and task duration; and (3) network conditions, such as latency and bandwidth across nodes. As these factors change, the framework is responsible for dynamically adapting the implementation—by switching architectures, tuning parameters such as heartbeat period, auto-scaling schedulers, etc.—to maintain optimal performance.

4 Key Research Questions

Realizing the approach requires answering the following research questions.

Q1: *What abstractions should our DSL provide to express scheduling policies?* The abstraction should be high-level, yet expressive enough to capture a wide range of scheduling policies. It should allow developers to specify not only decision constraints and optimization objectives—which previous works mainly focus on—but also task ordering strategies (e.g., fairness rules) and state-maintenance logic that governs how scheduling information evolves over time. Moreover, it should facilitate reasoning about internal states, as this is crucial for informed implementation strategy choices.

Q2: *How to quantify the scheduling quality drop introduced by concrete implementations?* Developers typically design scheduling policies assuming an ideal, omniscient scheduler with perfect global knowledge and zero latency. In practice, however, information is distributed and delayed, and every real implementation is an approximation of this ideal. For example, a centralized architecture, besides limited throughput under high load, might also suffer from information staleness if remote information was periodically collected via heartbeats. In addition, if the implementation includes decentralized decision-making, schedulers not aware of others' actions may independently choose the same node as the target, leading to a herding effect.

The gap between practical implementations and ideal ones can manifest as higher task completion times, throughput bottlenecks, fairness violations, or increased peak resource usage. The challenge lies in defining a unified model to capture these deviations, quantifying them for each policy–implementation pair, and using these metrics to inform deployment decisions.

Q3: *How to automatically find and switch to optimized implementation?* Given a scheduling policy specification, the system needs to (1) identify optimization opportunities , (2) select the most suitable implementation based on the current conditions and then configure the communication pattern between schedulers to support it, and (3) enable seamless transitions between implementations when conditions change, without disrupting active workloads.

5 A Potential Approach

We outline a potential approach to realize our vision. Developers describe expected scheduling policies in our DSL. The framework consists of a monitor that continuously tracks workload information and produces performance profilings, and a compiler that computes and generates the best implementations of the policies based on the monitor statistics.

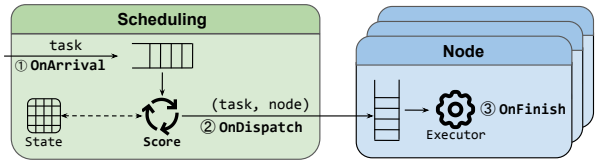


Figure 3: Abstraction of Task Lifecycle.

The underlying cluster contains programmable scheduler instances that accept scheduling executables to coordinatively realize various implementation plans.

5.1 Programming Abstractions

Inspired by existing scheduling DSLs such as DCM [31], our abstraction allows developers to define the task format as a struct and use the information to specify scheduling constraints and objectives. However, we significantly differ from previous works in the following aspects.

Scheduling target. Figure 3 shows the abstraction of a task's lifecycle, including scheduling and execution. Our abstraction allows developers to specify a (*task*, *node*) pair in each iteration as the scheduling decision, meaning that *task* is expected to be dispatched to *node*. Specifically, we require developers to complete a function `Score(task,node)`, which returns a score for any given (*task*, *node*) pair. The scheduler will use the function to evaluate all possible task and node combinations and select the pair with the best score. We choose the scoring abstraction because of its simplicity and ease of decision quality analysis in the compiler (explained later). This design opens up space for describing task selection policies such as fairness or priorities, and even more sophisticated policies requiring joint consideration of tasks and nodes.

State management. We enable developers to define and initialize internal states in the `State` section, and expose 3 hook points in the task lifecycle (annotated in Figure 3) for developers to express state update logic: (1) `OnArrival`, the moment when a task comes into the scheduler. (2) `OnDispatch`: the moment when a scheduling decision is finalized, and (3) `OnFinish`, the moment when a task is completed. The reason behind this design choice is that lots of common scheduling policies contain logic beyond the expressibility of purely declarative, side-effect-free abstraction. For example, even a simple round-robin policy for load balancing requires a counter that increments every time a new assignment is made. Therefore, supporting explicit state management across these lifecycle stages is essential for capturing a wide range of realistic scheduling behaviors.

```
1  Task:
2    tenant: string
3    resource_vec: vec<float>
```

(a) Task Format

```
1  State:
2    alloc: map<string, vec<float>>
3    capacity: vec<float> = [1.5, 2.0, 1.0]
4    // Capacity of 3 resource types, numbers are
       random.
5
6  Score(task, node): [limit=1.2]
7    tscore = max(alloc[task.tenant] / capacity)
8    nscore = get_load(node)
9    return tscore * nscore
10
11 OnDispatch(task, node):
12   alloc[task.tenant] += task.resource_vec
13
14 OnFinish(task, node):
15   alloc[task.tenant] -= task.resource_vec
```

(b) Scheduling Policy

Figure 4: DRF-LB Scheduling Policy Specification

Figure 4 describes the DRF-LB policy in § 2. The `Task` struct includes 2 scheduling-related fields: the tenant sending the task, and the resource vector it requires for execution. The scheduler declares a map `alloc` maintaining per-tenant resource allocation vectors, which are updated on task dispatching and completion (specified in `OnDispatch` and `OnFinish` hook points). The `Score` function computes the score by combining the task submitter's dominant resource share and the load of the target node. The `limit` annotation is used for quality analysis and will be explained later.

5.2 Compiler

Taking the scheduling policy specification and the statistics collected by the monitor (e.g., task incoming rate, network latency, historical performance profilings) as input, the compiler is responsible for finding and generating optimized implementations. Internally, after converting the specification into IR, the compiler divides the program into smaller execution units. (For example, in Figure 4b, Line 7 and Line 8 can be considered as separate execution units.) Then the compiler explores optimization opportunities from the following aspects to generate implementation plans:

- *Logic placement*: For each execution unit, the compiler considers which scheduler the unit should be placed in. Fully centralized and fully decentralized architectures can be regarded as two special cases of placement

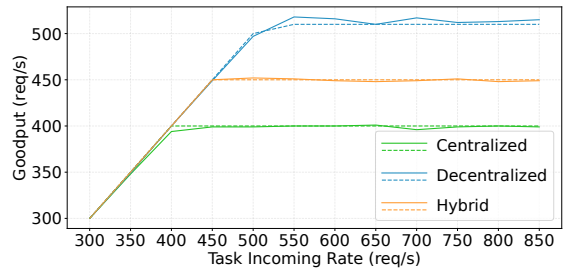
decisions. Smartly distributing part of the logic (i.e., hybrid architecture) can increase throughput without hurting decision accuracy.

- *Operation approximation*: Certain operations, under specific conditions, can be approximated with acceptable accuracy loss and decreased latency. For example, if a state is not changing rapidly, remote query of that state could be approximated as querying a local cache and updating the cache in the background periodically. The famous power-of-2 choices [24] can also be considered as an approximation for the global minimum by reducing the number of candidates. The compiler follows a rule-based approach, searching for matching patterns in the program and trying to apply approximations to optimize performance.

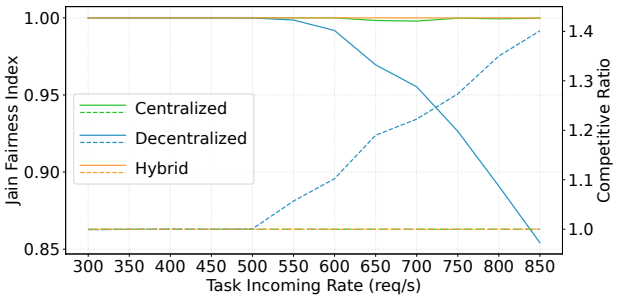
The compiler periodically evaluates potential implementation plans on 3 performance metrics: latency, throughput, and quality. End-to-end scheduling latency is estimated by summing up the estimated latencies of each execution unit based on profiling statistics provided by the monitor and approximation strategies applied to that unit. With latency results, throughput can be estimated mathematically in the following way: suppose the policy contains a set of execution units $U = U_c \cup U_d$, where U_c and U_d are units planned to be placed on centralized and decentralized schedulers, respectively. Then,

$$\text{Throughput}_{estimated} = \min\left\{\frac{1}{\sum_{u\in U_c} Latency(u)}, \frac{\#\text{Decentralized Scheduler}}{\sum_{u\in U_d} Latency(u)}\right\}$$

We quantify scheduling quality as the *competitive ratio* between scores of (hypothetical) ideal decisions and implementation decisions. Specifically, we emulate several rounds of scheduling decisions based on the traced historical workload. For ideal decisions, scores are computed with omniscient state values at that moment, while for the implementation decisions, computations are based on restricted values according to the optimizations. For example, if the implementation places all logic into decentralized schedulers, then schedulers cannot see non-local tasks waiting for scheduling. Values of some discrete states (e.g., counter) can be fully rebuilt by replaying the traced events, while some real-time values cannot be recovered for arbitrary moments. For the latter case, the compiler computes an approximate value for a given moment by adding an estimated Δ to the most recent traced value. After getting the ideal and the implementation decisions, the competitive ratio is computed as the ideal score ratio of the two decisions.



(a) Goodput Estimation



(b) Competitive Ratio Estimation. Solid lines are JFI and dashed lines are competitive ratios.

Figure 5: Evaluations of the compiler's ability to estimate implementation performance. Solid lines are execution results (same as § 2) and dashed lines are compiler estimations.

Formally, for any moment, if (t_{ideal}, n_{ideal}) and (t_{impl}, n_{impl}) are ideal and implementation decisions, respectively, i.e.,

$$(t_{ideal}, n_{ideal}) = \arg\min_{(task,node)} Score_{ideal}(task, node)$$
$$(t_{impl}, n_{impl}) = \arg\min_{(task,node)} Score_{impl}(task, node)$$

Then

$$Competitive\ Ratio = \frac{Score_{ideal}(t_{impl}, n_{impl})}{Score_{ideal}(t_{ideal}, n_{ideal})}$$

The closer the ratio is to 1, the closer scheduling decisions are to ideal ones, indicating better decision quality. We allow developers to annotate a ratio limit for the `Score` function, which represents the maximum decision quality drop they can tolerate. The compiler will filter out implementation plans with a competitive ratio larger than the limit, and select one from the rest with minimum latency and throughput capable of handling the current workload rate.

6 Preliminary Evaluation

We built a prototype framework consisting of a compiler that takes in scheduling policies written in DSL and evaluates implementation plans, plus a monitor that traces statistics

and scheduling-related events. In our evaluation, we run the same workload as the one in § 2 under various task incoming rates, and focus on whether the goodput and competitive ratio estimations computed by the compiler can reflect all the properties we observed in § 2.

Figure 5 presents the results. Figure 5a shows that goodput estimations based on profiled latency statistics closely match the actual values. In Figure 5b, we observe that for centralized and hybrid implementations that rigorously enforce fairness, the estimated competitive ratio remains consistently close to 1. For decentralized implementation, the competitive ratio starts to increase precisely at the point when fairness degrades, and they share the same trend. The evaluation demonstrates that our estimations serve as reliable guidance for the compiler to analyze the properties of implementations.

7 Related Works

Scheduling Abstractions. Designing expressive and concise abstractions for scheduling has been extensively studied in distributed systems [7, 31, 32], operating systems [15, 20], and high-performance computing [16, 33], from which we draw lots of inspiration for our design. However, these abstractions remain limited in expressiveness, do not utilize high-level semantics for implementation optimization, or have significantly different concerns because of domain gaps.

Scheduling Architecture. Previous works have proposed various kinds of scheduling architectures. Mesos [14] divides a cluster into partitions, allowing each partition to run separate scheduling policies. Omega [30] adopts a decentralized architecture with shared states for synchronization and lock-free concurrency control. Mercury [21] and Hawk [10] combine centralized and decentralized schedulers by assigning high-priority tasks to the former and best-effort tasks to the latter. These works, while excelling at specific scenarios, are not flexible enough to deal with various workloads, and do not expose abstractions for expressing or composing new policies.

Scheduling Optimization. There's a rich line of work focusing on scheduling overhead reduction. For example, Sparrow [29] uses power-of-two choices and batch probing to reduce latency in decentralized schedulers. Firmament [13] applies algorithmic optimizations to min-cost-max-flow problems to increase scheduling scalability. POP [26] and DeDe [34] further accelerate the resource allocation computation by applying domain-specific heuristics. These efforts can be integrated into our compiler as potential optimizations, but our focus on decoupling policy from implementation is orthogonal to this line of work.

8 Conclusion

We propose a new approach to scheduling in distributed systems that decouples what scheduling logic is implemented from how and where it is executed. By enabling developers to specify high-level policies in a domain-specific language and using a compiler to generate optimized, adaptive implementations, our framework addresses two fundamental limitations of current systems: limited expressiveness in policy and suboptimal performance due to rigid implementations. Our early results demonstrate that deployment architecture has a significant impact on both scheduling quality and system throughput, and that compiler-driven adaptation can navigate these trade-offs.

Acknowledgments

We thank the HotNets reviewers and our shepherd, Scott Shenker, for their valuable feedback. This work was supported by UW FOCI and its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware), and by NSF Grant 2402695.

References

[1] 2021. On the In-Depth Cluster Scheduling and Management. https://www.alibabacloud.com/blog/on-the-in-depth-cluster-scheduling-and-management_598012.

[2] 2025. Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[3] 2025. Kubernetes Scheduling Framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[4] 2025. llm-d: a Kubernetes-native high-performance distributed LLM inference framework. https://llm-d.ai/.

[5] 2025. Ray Scheduling. https://docs.ray.io/en/latest/ray-core/scheduling/index.html.

[6] 2025. Uber's Journey to Ray on Kubernetes: Resource Management. https://www.uber.com/blog/ubers-journey-to-ray-on-kubernetes-resource-management/.

[7] Romil Bhardwaj, Alexey Tumanov, Stephanie Wang, Richard Liaw, Philipp Moritz, Robert Nishihara, and Ion Stoica. 2022. ESCHER: expressive scheduling with ephemeral resources. In *Proceedings of the 2022 ACM Symposium on Cloud Computing*. 47–62.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 285–300.

[9] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, et al. 2024. MAST: Global scheduling of ML training across geo-distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 563–580.

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 499–510.

[11] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and Ulterior Software Engineering* (2017), 195–216.

[12] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[13] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

[14] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[15] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM 28th Symposium on Operating Systems Principles*. 588–604.

[16] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 703–718.

[17] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 261–276.

[18] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* 21, 1 (1984), 2022–2023.

[19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[20] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM 28th Symposium on Operating Systems Principles*. 605–620.

[21] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 485–497.

[22] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. 2020. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the 2020 ACM SIGCOMM Conference*. 359–376.

[23] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the 2019 ACM SIGCOMM Conference*. 270–288.

[24] Michael Mitzenmacher. 2002. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2002), 1094–1104.

[25] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[26] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. 2021. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM 28th Symposium on Operating Systems Principles*. 521–537.

[27] Jason Nieh, Christopher Vaill, and Hua Zhong. 2001. Virtual-time round-robin: an O (1) proportional share scheduler.. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. 245–259.

[28] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2013. The case for tiny tasks in compute clusters. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.

[29] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the ACM 24th Symposium on Operating Systems Principles*. 69–84.

[30] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 351–364.

[31] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building scalable and flexible cluster managers using declarative programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 827–844.

[32] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11st European Conference on Computer Systems*. 1–16.

[33] Anjiang Wei, Rohan Yadav, Hang Song, Wonchan Lee, Ke Wang, and Alex Aiken. 2025. Mapple: A Domain-Specific Language for Mapping Distributed Heterogeneous Parallel Programs. *arXiv preprint arXiv:2507.17087* (2025).

[34] Zhiying Xu, Minlan Yu, and Francis Y Yan. 2025. Decouple and decompose: Scaling resource allocation with DeDe. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[35] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the ACM 24th Symposium on Operating Systems Principles*. 423–438.

[36] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. 2017. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 377–392.

[37] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. SGlang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems* 37 (2024), 62557–62583.