Efficiently Improving the Performance of Serverless Applications with Microtask-based Scheduling

Sacheendra Talluri, Sven Lankester, Bogdan Ene, Jesse Donkervliet, Animesh Trivedi, Alexandru Iosup
Vrije Universiteit Amsterdam, The Netherlands
s.talluri@vu.nl, s.lankester@student.vu.nl, b.g.ene@student.vu.nl, j.j.r.donkervliet@vu.nl, a.trivedi@vu.nl, a.iosup@vu.nl

***Abstract*—Serverless computing promises to make cloud computing cheaper and easier to use. However, serverless platforms use coarse-grained scheduling which decreases efficiency and application performance. We propose a fine-grained application model for serverless applications, and use it to design a scheduler to improve application performance and efficiency. We model serverless applications as being composed of *microtasks*, each with its own unique resource requirements. Microtasks are easily identified via distinct application phases like initialize, read, and process. We provide evidence for the existence of microtasks by experimentally evaluating a serverless online game. We design a scheduler that separates microtasks with different CPU requirements into different queues so that the appropriate amount of CPU cores could be allocated to each queue based on the CPU requirements of the microtasks in that queue. We implement and evaluate the design in an application-level proof-of-concept microtask-based scheduler and compare it to task-based scheduling commonly used by serverless platforms. For a distributed sort application, the microtask-based scheduler decreases application makespan by 37% and the duration of I/O based application stages by 81%, compared to task-based scheduling. Our work suggests that there is potential in extracting and using microtask information from serverless applications.**

***Index Terms*—cloud, serverless, performance, scheduling, game**

I. Introduction

Cloud computing is widely used by society, with end-user spending on the cloud predicted to reach over $500 billion [1]. Serverless computing promises to make the cloud cheaper and easier to use [2]. But, serverless platforms still inefficiently allocate resources in a coarse-grained manner for the whole duration of the application task [3]. Existing work tackles this problem, but does not take into account the fine-grained structure of serverless tasks. We investigate whether indeed task resource requirements and allocated resource granularity match in serverless computing and, after showcasing evidence they do not, propose a finer-granularity scheduler.

Serverless computing promises to fulfill the pay-per-use promise of cloud computing, whereby users pay only for resources they use at a fine granularity [2]. The community has made much progress in making this possible as resources can be leased for as little as 100 milliseconds. But, all popular open-source serverless platforms come with the limitation that the quantity of resources (CPU, RAM, and network bandwidth) allocated to a serverless task is fixed for its whole duration [4]–[6]. At best, they use simple oversubscription. The fixed resource quantity violates the pay-per-use promise,

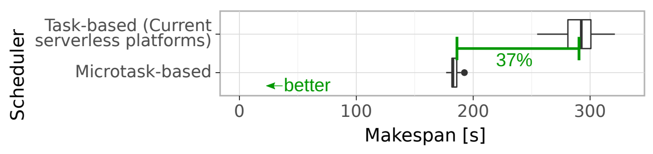

Fig. 1. Exemplary result comparing the makespan of serverless sort implementation with microtask-based scheduling to one with task-based scheduling. The microtask-scheduling based version finishes 37% earlier, by median values.

as applications do not completely use the leased resources all the time [3]. This inefficiency also hurts the performance of applications as resources are reserved for applications which do not use them.

Do real-world serverless applications leave room for better scheduling? Many current serverless applications are based on relatively short tasks (100s of milliseconds to a few seconds) [7]–[9]. At this fine granularity, it is unclear if there is room to improve resource utilization and reduce makespan with better scheduling. Through an experimental evaluation, we demonstrate that the fine-grained parts of the serverless tasks which correspond to different resource demands can be clearly seen in the structure of the application. We call these clearly identifiable portions *microtasks*. We show the presence of microtasks in two applications: sort, a popular benchmark representative of the shuffle phase found in data analytics applications [10], and a serverless online game [11].

Even after identifying the microtask-based structure of applications, using it for better scheduling is challenging as applications are not setup to expose their microtasks, and the schedulers not setup to use that information. We propose that programmers can communicate an application's microtask structure to the serverless runtime using annotations, for better scheduling decisions. We present a scheduler design for one resource, where microtasks with different CPU demands are scheduled using different queues. CPU cores are allocated to service each queue based on the intensity of the microtasks that queue services.

Although appealing, the concept of microtasks has not been evaluated in practice. In this work, we conduct early experiments with microtask-based scheduling (§IV). We evaluate if exposing an application's microtask-based structure improves the performance of the application. We compare the makespan, time elapsed from the start of the application

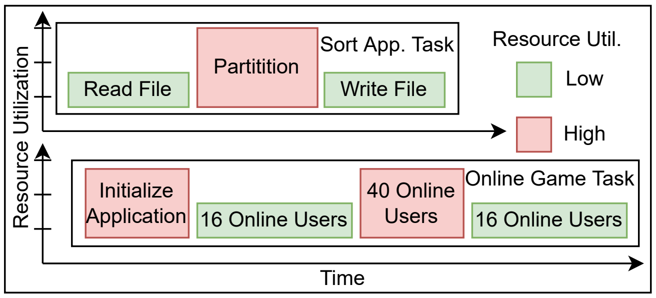

Fig. 2. Tasks from two applications, sort and online game, represented with a microtask-based model. Notice that different microtasks in each application task have different resource utilization. Read, write, and 16 online users require less resources and do not use all resources allocated to the application. The other microtasks can make use of all the allocated resources. Also notice that the microtasks correspond to easily identifiable parts of the applications.

to its completion, of the sort application implemented using microtask-based scheduling to the sort application using task-based scheduling. Figure 1 depicts an exemplary result that captures the contribution of this work. Using our method (§III) and experimental evaluation (§IV), the median duration of a serverless application can be reduced from 292 seconds to 183 seconds when using microtasks (a 37% improvement), and the rest of the statistical distribution of makespan shows similar improvements.

Addressing the problem of efficiently improving the performance of serverless applications, we make a three-fold contribution:

1) We model serverless applications as being composed of microtasks in §II. We support our model with the experimental evaluation of an online game.
2) We design a microtask-based scheduler that separates microtasks with different CPU requirements into different queues in §III. It then partitions the available CPU cores across these queues based on the the CPU requirements on the microtasks the queue serves.
3) We evaluate the microtask-based scheduler design by implementing and comparing it to the task-based scheduler design used in popular open-source serverless platforms in §IV.

II. Real-World Serverless Task-Structure Leaves Room for Better Scheduling

In this section, we present an application model based on microtasks, and experimental evidence to support that model.

A. Microtask-based Model of Serverless Applications

We depict the microtask-based execution model of two applications, sort [10] and serverless online game [12], and their constituent microtasks in Figure 2. Distributed serverless applications like sort and the online game are composed of many tasks. Each task performs a specific function like processing a piece of data, or simulating a potion of the online game. These tasks are further divided into *microtasks*, each of which has specific resource requirements and can be easily identified in the structure of the application.

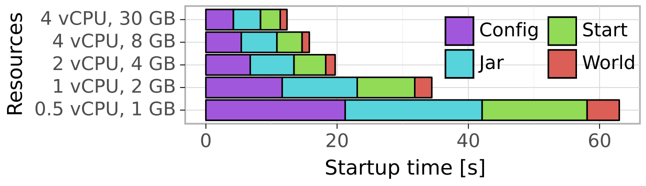

Fig. 3. Breakdown of startup time by process step, for several resource configuration, for a serverless online game. Notice the decrease in startup time as the amount of resources increases.

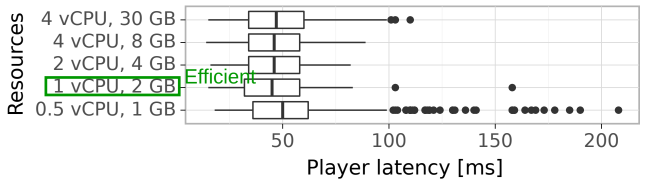

Fig. 4. The distribution of the duration of simulation ticks, when 16 players execute a standard benchmark, for several resource configurations. Notice the lack of change after 1 vCPU. At 0.5 vCPU, there are many outliers which is not desirable for a game. Therefore, going higher up to 1 vCPU is beneficial. Anymore resources than 1 vCPU are redundant.

The serverless sort application takes as input files filled with unsorted 100 byte records. The application works in two stages. First, the unsorted records are partitioned into categories according to the first byte and the outputs written to storage. In the second stage, the categories output by the first stage are sorted. Figure 2 depicts a single function execution during the first stage. The function execution is split into 3 microtasks: read, partition, and write. The read and write microtasks use little CPU compared to the partition microtask. But, the function needs to be configured with the high amount of CPU necessary for the partition microtask.

The serverless online game application is composed of a single stage. The application function is initialized, and can then simulate the environment for the connected players. The function execution is simplistically split into 3 microtasks: initialization, 16 users online, and 40 users online. The 16 users online microtask consumes little CPU compare to the initialization or the 40 users online microtasks. But, the function needs to be configured with the resources necessary for the larger microtasks.

In both the aforementioned applications, the applications are configured with a fixed amount of resources irrespective of how many resources the microtask executing at that time actually needs. In our model, the read/write stages in the sort application and the 16 online users stage in the game have low CPU utilization. But, a high amount of CPU are reserved for the whole application based on the highest resource utilization microtasks. We will validate this model for the serverless online game in §II-B.

B. Experimental Evidence for the Microtask-based Model

To support our claim about the existence of microtasks with different resource demands, we design an experiment

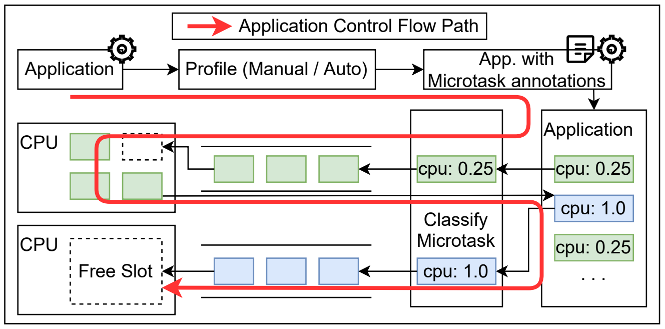

Fig. 5. Design of the microtask-based scheduler.

to evaluate the performance of microtasks as the resources allocated to them change. We perform this evaluation with one of the two modeled application, the serverless online game.

Figure 3 depicts the initialization time of a game function under different resource allocations. Notice the 80% drop in initialization time from 63 seconds to 12 seconds when we allocate 4 vCPUs instead of 0.5 vCPUs. Hence, the initialization microtask can make use of a large amount of allocated resources for better performance.

Figure 4 depicts the latency experienced by users as more resources are allocated to the function. The number of outliers decreases going from 0.5 vCPU to 1 vCPU, which is important for game. Beyond 1 vCPU, there is no drop in latency. Hence, the game simulation microtask responsible for users does not perform better as more resources are allocated. An efficient scheduler would just allocate the 1 vCPU the simulation microtask requires to perform well.

Therefore, we demonstrate that some microtasks of the same function are more resource hungry than others.

III. Better Scheduling Using Microtasks

Our design goal is to make it possible for the serverless runtime system to allocate the exact amount of resource required by a microtask. This has the potential of users paying for exactly the resources they use. We explore several alternative approaches for this, and explain how microtask-based scheduling achieves our goal.

A. Design Alternatives

Existing operational techniques for efficient resource allocation use statistical properties of applications to schedule multiple applications together. The scheduling is done such that resource left unused by one application are used by others. Three major categories of operational techniques exist classified based on the kind of performance isolation they provide.

Oversubscription: All resources on a machine are available to all applications scheduled onto that machine. The applications to co-locate are chosen such that even if the combined resource requirements are greater than the capacity of the machine, the statistically expected resource usage is less than the resources available. The statistical nature of the co-location means that there exist some periods when the resource usage could be higher than available resources. This can lead to increased tail latency.

Vertical autoscaling: Only resources allocated by the autoscaler are available to each application. The autoscaler dynamically decides the amount of resources allocated to each application based on prediction techniques such as ML [13], moving average [14], or other models. The models have to predict the time of resource requirements change, and the magnitude of the change. These predictions are subject to availability of data and can have significant error.

Batching: Multiple tasks of an individual application can be batched and processed together to amortize costs [15]. Batching also allows inter-task scheduling whereby microtasks can be combined together across tasks for better resource efficiency. But, batching introduces additional latency whereby the runtime of the fastest task in the batch is bound by the slowest task in the batch.

B. Design of a Microtask-aware Scheduler

Microtask-based scheduling takes advantage of the fact that application tasks are composed of microtasks with their own resource requirements. Application developers can easily find out the resource requirements of the microtasks by manual or automated profiling. But, current serverless platforms lack the ability to obtain this information from the developer and take advantage of it.

Our design enables application developers to inform the serverless runtime of the microtasks in an application and their resource requirement. The developers communicate this information using annotations on the functions that correspond to microtask boundaries. The runtime makes efficient scheduling and resource management decisions based on the information received from all microtasks running on a machine.

Figure 5 depicts the design of the microtask-based CPU scheduler. First, the application is profiled, either manually by the developer, or using automated tools to extract the resource requirements of the individual microtasks it is composed of. The microtask start locations and the resource requirements are then embedded in the application. When the application is executed, the resource requirements are supplied to the scheduler so that the microtask can be classified and the application enqueued into the appropriate queue based on its resource requirements. All microtasks which belong to the same resource requirement class, hence have the same resource requirements, are serviced by the same queue. A microtask is considered complete when the application is descheduled, or the start of a different microtask is encountered. The old microtask is then retired and the application is placed in a new queue based on the requirements of the new microtask.

Each queue schedules a variable number of microtasks on a CPU based on the CPU fraction specified by the resource requirements class it services. Each queue is assigned a specific number of CPU cores that are used to service the tasks in that queue. Several queues exist, each for different sized microtasks, on a single machine reminiscent of slab

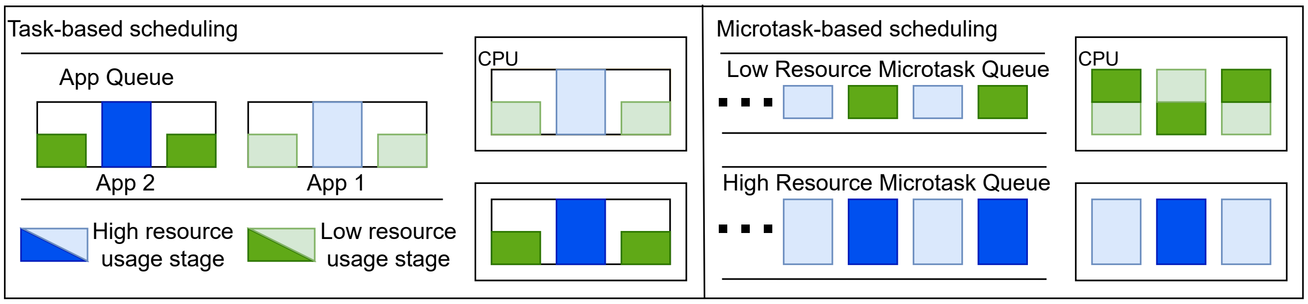
Task-based scheduling
App Queue
App 2
App 1
High resource usage stage
Low resource usage stage
CPU
Microtask-based scheduling
Low Resource Microtask Queue
High Resource Microtask Queue
CPU

Fig. 6. Illustration of scheduling tasks of two applications when using task-based and microtask-based scheduling. The washed out task stages belong to App 1, and the bright ones to App 2. Blue stages require a high amount of CPU, and the green stages a low amount. Notice that the whole CPU is not used during the low resource usage stage (green) when using task-based scheduling. When microtasks are explicitly identified, low resource usage stages can be scheduled together.

memory allocation. The exact number of CPU cores per queue is decided dynamically based on the relative lengths of queues an their resource requirements. For the evaluation in this paper, we use static allocations. But, we expect that in a fully realized system, the allocations are dynamic as the workload serviced by a machine changes over time.

It possible to implement microtask-based scheduling both at the application level and at the operating system level. Application level schedulers are common in complex workflow-based [16], pilot-based [17], or other complex applications. Operating system level schedulers are commonly used for single task, even monolithic applications. One advantage of operating system level scheduling is that it allows microtask-based scheduling across different applications written in multiple programming language. Application level schedulers are limited to the tasks of that single application. In this work, to evaluate our idea, we focus on application level schedulers due to ease of implementation.

Figure 6 contrasts microtask-based scheduling with task-based scheduling. Task-based scheduling is commonly used in open-source serverless platforms [4]–[6]. In this work, we only consider one resource, the CPU, but the resource model can be extended to include other resources such as I/O and memory. In task-based scheduling, application tasks enter the queue to be serviced. They are scheduled onto an available free CPU. In the figure, we have tasks from two applications. Each task occupies a whole CPU core for it's full duration. In microtask-based scheduling, the constituent microtasks of a task are assigned to different queues. The scheduler knows the the resource requirements of the microtasks in a queue. If a microtask uses less than the full CPU core, multiple microtasks are scheduled onto that core. In the figure, notice that multiple green microtasks are serviced at the same time when using microtask-based scheduling. When using task-based scheduling, the part of the CPU not used is left idle when the green microtask is running.

IV. Evaluation of Microtask-based Scheduling using the Sort Application

We experimentally evaluate the microtask-based scheduler introduced in §III-B by implementing it as a proof-of-concept application level scheduler in the sort application. We compare it to a task-based scheduler each task in the sort application gets its own CPU core. Task-based scheduling is used in popular open-source serverless platforms [4]–[6].

We chose an experimental evaluation as precise requirements of the microtasks are not yet characterized enough for a theoretical evaluation. Existing evidence, while hinting their existence [3], does not evaluate using them as a scheduling block. We use sort, as it is a representative workload, who performance is predictive of the performance of the shuffle stage of data analytics applications. It has also been used in prior work evaluating the suitability of serverless computing for data analytics [18].

We use the microtask-based scheduler design from §III-B configured with 3 queues: for read, process, and write microtasks each with different resource requirements. The number of CPU cores allocated for each queue in the microtask-based scheduler were 6, 10, and 8 respectively in the stage 1, and 14, 6, and 4 in stage 2. We configure the sort application to sort a 100GB dataset. The experiments were run in a cluster environment using 22 servers, each with 24 cores and 128GB RAM, for executing the application. The input data was read from a NVMe SSD accessed over a 100Gbps network interface, using MinIO, an S3-compatible object storage system. Each experiment is repeated 10 times.

A. Evaluation of Makespan and Individual Stages of the Distributed Sort

We compare the performance of the distributed sort application using the microtask-based scheduling approach to one using the task-based scheduling approach. Figure 7 depicts the ECDFs of 3 application metrics for both approaches: the makespan, which is the total time elapsed between the start of the application and the end; the duration of the sub-stages that write categorized partitions at the end of the categorize records stage; and the duration of the sub-stages that sort the category after all parts of the category have been retrieved in the sort category stage.

Sort 37% faster when using microtask-based scheduling compared to task-based scheduling: We observe that the median makespan when using microtask-based scheduling, 183

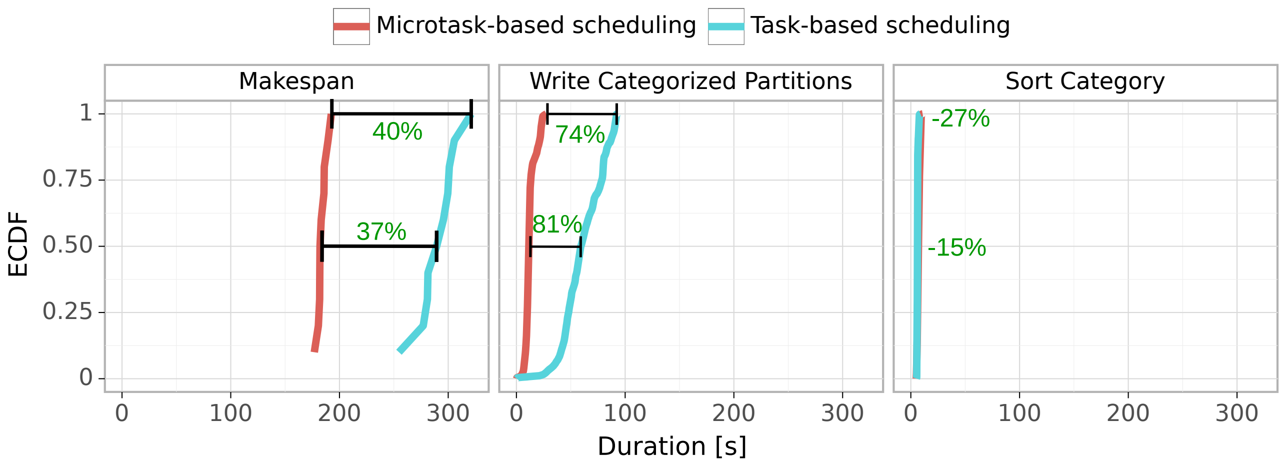

Fig. 7. The triptych compares the performance of task-based and microtask-based scheduling across the whole sort application makespan, and two sub-stages. Microtask-base scheduling achieves a median improvement of 37% for the application makespan. The write categorized partition stage see a median improvement of 81%. The sort category sub-stage, which uses the whole CPU core, sees a median performance decrease of 15% due to overheads introduced by microtask-based scheduling.

seconds, is 37% lower than the median makespan when using task-based scheduling, 292 seconds. The maximum makespan when using microtask-based scheduling, 192 seconds, is 40% shorter than that of task-based scheduling, 320 seconds. This difference in performance on the same hardware suggests that microtask-based scheduling utilizes the available CPU resources more efficiently than task-based scheduling. The I/O is not saturated in this work under any configuration and we leave further analysis of task based scheduling considering I/O as future work.

Performance of I/O intensive stages improves up to 81%, while performance CPU intensive stages decreases by up to 27%: We further analyze which sub-stage benefited most from microtask-based scheduling, and which benefited the least. We depict the ECDF of two sub-stages which gain (Write Categorized Partitions) and lose (Sort Category) performance due to microtask-based scheduling. The median duration of the Write Categorized Partitions sub-stage improved 81%, from 59 seconds to 11 seconds, when we move from task-based scheduling to microtask-based scheduling. The 99th percentile performance of Write Categorized Partitions sub-stage improved 74%, from 92 seconds to 24 seconds, when we move from task-based scheduling to microtask-based scheduling.

The median duration of the Sort Category sub-stage deteriorated 15%, from 6.1 seconds to 7 seconds, when we move from task-based scheduling to microtask-based scheduling. The 99th percentile performance of Sort Category sub-stage deteriorated 27%, from 7.5 seconds to 9.5 seconds, when we move from task-based scheduling to microtask-based scheduling.

Cause of the performance difference: The sub-stages whose performance improved, such as Write Categorized Partitions, were the ones which were performing I/O and were not fully utilizing the CPU when a whole core is allocated to them. In microtask-based scheduling, when resources could be customized for different sub-stages, more of these CPU-light sub-stages could be run concurrently. Hence, the improved performance.

The sub-stages whose performance worsened, such as Sort Category, were the ones which were fully utilizing the allocated CPU core. In microtask-based scheduling, the increased scheduling overhead did not result in any improved resource utilization for these tasks. Hence, the decreased performance.

Comparison to related results: Vertical autoscaling by Autopilot [14] reduces the number of unused resource by 50%. Adaptive batching [15] also reduces the application runtime cost by up to 50%, albeit limited to ML inference applications. While we do not directly measure the reduction in unused resources, we can reasonably assume that the performance improvement is the lower bound on the number of unused resources, as all the improved performance comes from improved scheduling. Hence, microtask-based scheduling has the potential to reduce the number of unused resources by at least 37%.

V. Discussion

We address the challenges to the validity of the work, and discuss the problems that need to be solved before before a system that utilizes the design of this work is fully realized.

Quantitative comparison to other techniques: This paper does not quantitatively compare microtasks to other techniques like vertical scaling and oversubscription. We expect to develop a framework for comparing these different techniques in the follow up work.

Dynamic CPU allocation: We allocate a fixed number of CPUs to each type of microtasks. In a production workload, the workload is continuously changing and the allocation needs to change along with that. While we expect a basic greedy allocator which allocates more CPUs to a type when there is more demand to work, it needs to be evaluated.

Extending the microtask model to other resources: We apply microtask-based scheduling only to the CPU in this work. Complex applications, such as multiplayer games, use multiple resources at the same time. The game could be simulating player actions and communicating with client as the same time. This requires that microtask-based resource management system to consider multiple resource type, network in this case.

Application vs. OS level scheduling: History has shown that both classes of schedulers are important. (1) OS level schedulers acquire key generic features of app-level schedulers, in time. (2) Application level schedulers allow application specific performance improvements to reach users faster, with the limitation of not being usable across applications. While our application level scheduler demonstrates the potential of microtask-based scheduling, an OS level implementation would bring its benefits to many more applications.

VI. Related Work

Closest to our work is monotasks [19]. Monotasks is a model for big data applications whereby tasks can only request one kind of resource. The single resource monotasks help with performance clarity. Our work demonstrates that a wider variety of apps than just big data apps can benefit from fine-grained scheduling and resource management. Our work also demonstrates the performance benefit of fine-grained resource management in the serverless context.

Autopilot [14] and tiny autoscalers [13] both use vertical scaling, and use moving average estimates to predict resource demand. We argue that for certain classes of applications, online interactive and big data, live prediction is not necessary. The applications have fixed structure, and can inform the resource manager of the current microtask they are executing.

Sizeless [20] predicts the optimal resource allocation for serverless tasks, but the predictions are coarse grained for the whole duration of the task. Nightcore [21] uses fine-grained scheduling to achieve better latency and utilization for microsecond scale tasks. But, it requires applications to use a custom I/O stack and runtime.

Batch [15] dynamically batches multiple ML inference requests together to amortize costs and fully make use of allocated resources. But, not all workloads all amenable to batching. Latency sensitive workloads like interactive applications are particularly impacted by the additional latency.

Function fusion [22], [23] and decomposition [23] have been proposed to improve serverless application performance. Applications that are I/O light but have compute intensive parts were decomposed for better scheduling. Applications bottlenecked on I/O were fused to reduce I/O overhead. Microtask-based scheduling can be used in both scenarios to improve resource utilization in a single machine.

Existing characterizations of serverless applications [3] indicate the existence of application parts with different resource demands. We demonstrate that the different resource demands can be used to improve the performance and efficiency of serverless applications.

VII. Conclusion

Serverless computing continues to make cloud computing cheaper and more accessible. Towards addressing the problem of efficiently improving the performance of serverless applications, we make a three-fold contribution. We model serverless applications as being composed of microtasks, and support our model with experimental evaluation of an online game. We design a microtask-based scheduler which uses different queues for microtasks with different CPU requirements. We evaluate the microtask-based scheduler design by comparing to task-based scheduler designs found in popular open-source serverless platforms. Finally, we discuss the shortcomings of this and propose promising directions for improvement so that a microtask-based schedulers can benefit more applications.

References

[1] Gartner, "Gartner forecasts worldwide public cloud end-user spending to reach nearly $500 billion in 2022," 2022. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022

[2] E. V. Eyk *et al.*, "Serverless is more: From paas to present cloud computing," *IEEE Internet Comput.*, 2018.

[3] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *MICRO 2019*. ACM, 2019.

[4] Rodric Rabbah and others, "Apache openwhisk," 2022. [Online]. Available: https://github.com/apache/openwhisk

[5] S. Hendrickson *et al.*, "Serverless computation with openlambda," in *HotCloud 2016*.

[6] CNCF, "Knative," 2022. [Online]. Available: https://knative.dev/docs/

[7] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, 2021.

[8] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *ATC 2020*.

[9] S. Fouladi *et al.*, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *ATC 2019*.

[10] Chris Nyberg and Mehul Shah, "Sort benchmark homepage," 2022. [Online]. Available: http://sortbenchmark.org/

[11] J. Donkervliet, A. Trivedi, and A. Iosup, "Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems," in *HotCLoud 2020*.

[12] Jesse Donkervliet and team, "Opencraft minecraft server," 2022. [Online]. Available: https://github.com/atlarge-research/opencraft

[13] Y. Zhao and A. Uta, "Tiny autoscalers for tiny workloads: Dynamic CPU allocation for serverless functions," in *CCGrid 2022*.

[14] K. Rzadca *et al.*, "Autopilot: workload autoscaling at google," in *EuroSys 2020*.

[15] A. Ali *et al.*, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *Supercomputing 2020*. IEEE/ACM, 2020.

[16] M. Zaharia *et al.*, "Apache spark: a unified engine for big data processing," *Commun. ACM*, 2016.

[17] M. Turilli, M. Santcroos, and S. Jha, "A comprehensive perspective on pilot-job systems," *ACM Comput. Surv.*, 2018.

[18] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *NSDI 2019*.

[19] K. Ousterhout *et al.*, "Monotasks: Architecting for performance clarity in data analytics frameworks," in *SOSP 2017*.

[20] S. Eismann *et al.*, in *Middleware 2021*.

[21] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *ASPLOS 2021*.

[22] T. Elgamal *et al.*, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *SEC 2018*.

[23] A. Mahgoub *et al.*, "SONIC: application-aware data passing for chained serverless applications," in *ATC 2021*, I. Calciu and G. Kuenning, Eds.