Jiffy: Elastic Far-Memory for Stateful Serverless Analytics

Anurag Khandelwal
Yale

Yupeng Tang
Yale

Rachit Agarwal
Cornell

Aditya Akella
UT Austin

Ion Stoica
UC Berkeley

Abstract

Stateful serverless analytics can be enabled using a remote memory system for inter-task communication, and for storing and exchanging intermediate data. However, existing systems allocate memory resources at job granularity—jobs specify their memory demands at the time of the submission; and, the system allocates memory equal to the job's demand for the entirety of its lifetime. This leads to resource underutilization and/or performance degradation when intermediate data sizes vary during job execution.

This paper presents Jiffy, an elastic far-memory system for stateful serverless analytics that meets the instantaneous memory demand of a job at seconds timescales. Jiffy efficiently multiplexes memory capacity across concurrently running jobs, reducing the overheads of reads and writes to slower persistent storage, resulting in $1.6-2.5\times$ improvements in job execution time over production workloads. Jiffy implementation currently runs on Amazon EC2, enables a wide variety of distributed programming models including MapReduce, Dryad, StreamScope, and Piccolo, and natively supports a large class of analytics applications on AWS Lambda.

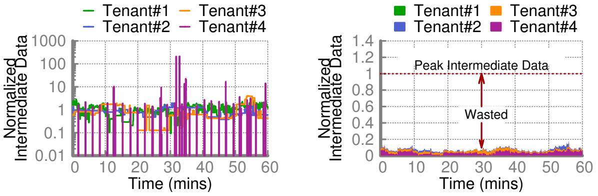
CCS Concepts: • **Computer systems organization → Cloud computing**.

Keywords: serverless computing, far-memory, data analytics

ACM Reference Format:
Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3492321.3527539

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04...$15.00
https://doi.org/10.1145/3492321.3527539



(a) Intermediate data (normalized by mean usage)

(b) Cumulative intermediate data (normalized by peak usage)

Fig. 1. Analysis of production workloads from Snowflake [20] for four tenants over a 1 hour window: (a) the ratio of peak to average storage usage for a job can vary by an order of magnitude during its execution; and (b) provisioning for peak usage results in average utilization < 10%. Across all tenants, the average utilization is 19%.

1 Introduction

Serverless architectures offer on-demand elasticity of compute and persistent storage, while charging users for resources consumed by their jobs at fine-grained timescales [1–3]. While originally deemed useful only for web microservices, IoT and ETL workloads [4, 5], recent work on serverless analytics has demonstrated the benefits of serverless architectures for resource- and cost-efficient data analytics [6–22].

The core idea in serverless analytics is to use a remote low-latency, high-throughput shared far-memory system for: (1) inter-task[1] communication; and (2) for multi-stage jobs, storing intermediate data beyond the lifetime of the task that produced the data (until it is consumed by downstream tasks). We use far-memory to refer to memory on remote servers accessed over the network [23, 24], including disaggregated memory [25–40]. Such far-memory systems thus allow decoupling storage, communication and lifetime management of intermediate data from individual compute tasks, enabling serverless analytics frameworks to exploit the on-demand compute elasticity offered by serverless architectures.

Existing far-memory systems [7, 8], however, suffer from a fundamental limitation: they allocate storage resources at the job granularity. That is, jobs specify their memory demands at the time of the submission; and, the system allocates and

[1] Existing distributed programming frameworks, while different in underlying programming models and semantics, share a common structure (Fig. 2, Fig. 3) — the job is split into multiple *tasks*, possibly organized along multiple stages or a directed acyclic graph. Each task generates *intermediate* data during its execution; upon completion, each task partitions its intermediate data and exchanges it with tasks in the next stage.

reserves memory resources equal to the job's demand (potentially by elastic scaling of total system capacity) for the entirety of its lifetime [8, Fig. 1].

The problem of performance degradation and resource underutilization for such job-level resource allocation is well-understood [20, 41]. On the one hand, if jobs specify their average demand, their performance degrades when instantaneous demand is higher than their average demand (due to read/write requests being executed on slower secondary storage, *e.g.*, S3), as shown in Fig. 1(a). On the other hand, if jobs specify their peak demand, the system suffers from resource underutilization when their instantaneous demand is lower than the peak demand, as shown in Fig. 1(b). Indeed, the problem worsens as the difference between the peak demand and the average demand increases. Unfortunately, the target use case of far-memory systems for serverless analytics—storage and exchange of intermediate data—is a bad-case scenario for the difference between peak and average demands: recent deployment studies have reported that intermediate data sizes can vary over multiple orders of magnitude during the lifetime of the job [20]. For instance, Fig. 1 presents our analysis of the publicly-released dataset of > 2000 tenants from Snowflake [20]: it shows that the ratio of peak to average demands in Snowflake production workloads can vary by two orders of magnitude over a period of minutes! As a result, job-level memory allocation in existing systems can lead to significant performance degradation and resource underutilization — over $4.1\times$ performance degradation and 60% resource underutilization in our evaluation (§6).

We present Jiffy, an elastic far-memory system for stateful serverless analytics. Jiffy allocates memory resources at the granularity of small fixed-size memory blocks—multiple memory blocks store intermediate data for individual tasks within a job. Jiffy design is motivated by virtual memory design in operating systems that also does memory allocation to individual processes at the granularity of fixed-size memory blocks (pages); indeed, Jiffy adapts this design to stateful serverless analytics. Performing resource allocation at the granularity of small memory blocks allows Jiffy to elastically scale memory resources allocated to individual jobs *without* a priori knowledge of intermediate data sizes, and to meet the instantaneous job demands at seconds timescales. As a result, Jiffy can efficiently multiplex the available faster memory capacity across concurrently running jobs, thus minimizing the overheads of reads and writes to significantly slower secondary storage (*e.g.*, S3 or disaggregated storage [20, 42, 43]).

Enabling fine-grained resource allocation requires resolving four unique challenges introduced by serverless analytics:

- First, each serverless analytics job can be organized around multiple stages (or a directed acyclic graph), with tens to thousands of individual tasks in each stage [6–20]. Thus, performing fine-grained resource allocation requires an efficient mechanism to keep an up-to-date mapping between tasks and memory blocks allocated to individual tasks.
- Second, the number of tasks reading and writing to the shared far-memory system can change rapidly in serverless analytics. Thus, task-level isolation becomes critical: arrival and departure of new tasks should not impact the performance of existing tasks.
- Third, decoupling of serverless tasks from their intermediate data means that the tasks can fail independent of the intermediate data. Thus, we need mechanisms for explicit lifetime management of intermediate data.
- Fourth, decoupling of tasks from their intermediate data also means that data partitioning upon elastic scaling of memory capacity becomes challenging, especially for certain data types used in serveless analytics (*e.g.*, key-value stores [6–8, 11, 13, 15, 19]). Indeed, naïvely delegating this to applications would require large network transfers (between compute tasks and the far memory system) and data read/write operations every time the capacity is scaled (§3). Thus, we need new mechanisms to efficiently enable data partitioning within the far memory system.

Jiffy resolves these challenges by integrating several mechanisms into an end-to-end system. First, in a sharp contrast to classical distributed shared memory systems [44–48] and recent in-memory stores [49–52] that use a global address space, Jiffy exposes a hierarchical address space that captures the structure of the analytics job (*e.g.*, directed-acyclic graphs with individual tasks) [20, 41]. Such a hierarchical addressing mechanism allows Jiffy to both efficiently manage the mapping between memory blocks and tasks, and provide task-level isolation. Second, Jiffy ties the hierarchical addresses with a lease-based mechanism for efficient lifetime management of intermediate data. Finally, similar to function shipping, Jiffy supports partition-function shipping — analytics jobs can offload data repartitioning upon resource allocation/deallocation to Jiffy, that performs seamless data repartitioning. We discuss in §3 how, for each of these techniques, the aforementioned unique challenges introduced by serverless architectures require Jiffy to make different design decisions than original realizations of these mechanisms. Jiffy integrates these mechanisms into an end-to-end far-memory system for stateful serverless analytics that provides resource elasticity at the granularity of seconds, matching the compute elasticity timescales of serverless architectures.

We have realized an end-to-end implementation of Jiffy, now open-sourced at `https://github.com/resource-disaggregation/jiffy`. Jiffy's data plane enables compute tasks to read/write intermediate data to their blocks via an intuitive, programmable, API (§4.1). We demonstrate the expressiveness of Jiffy's API by realizing serverless incarnations of several powerful distributed programming frameworks atop Jiffy (§5): MapReduce [53], Dryad [54], StreamScope [55] and Piccolo [56]. We compare Jiffy against five state-of-the-art far-memory systems over a variety of stateful serverless workloads and cluster configurations. Our evaluation suggests that, compared to state-of-the-art systems for stateful serverless analytics [8], Jiffy's fine-grained resource allocation achieves as much as 3× better resource utilization, and improves application performance by a factor of $1.6 - 2.5\times$.

2 Motivation

The state-of-the-art system for stateful serverless analytics is Pocket [8], a distributed low-latency, high-throughput system for storing intermediate data. Pocket already resolves a number of interesting challenges pertinent to stateful serverless analytics, as we describe next.

Scalable centralized management. Pocket architecture (Fig. 2) comprises decoupled control, metadata and data planes. While the data storage itself is distributed across multiple storage servers, the storage management functionalities via control and metadata planes are logically centralized. This greatly simplifies management since the controller and metadata servers have a global view of the entire system. Specifically, the controller allocates storage resources to analytics jobs and decides where to place the data for different jobs based on its global view of load across storage servers. The metadata plane, in turn, organizes job data into buckets across the storage allocated by the controller, and stores the mapping from buckets to their physical locations at the data plane for directing client requests appropriately. A single centralized metadata server can support 90K requests per second per core (sufficient to support thousands of serverless tasks).

Multi-tiered data storage. Pocket data plane simply stores the job data in a bucket across multiple storage servers and serves them via a key-value API (i.e., `get()`, `put()`). Once a job obtains the physical resource locations from the metadata server, it can read and write data directly from the storage servers. Pocket supports multi-tiered storage: jobs can store data across DRAM, Flash or HDD tiers at the data plane, based on their performance and/or cost constraints. In our work, we focus on DRAM as the storage medium for ephemeral data to realize a far-memory system; as such, we subsequently refer to storage servers as 'memory' servers.

Adding/removing memory servers. If the aggregate demand of jobs storing data on Pocket grows so much that the system memory capacity is insufficient to serve all of them, then Pocket can scale up the memory capacity by adding more memory servers at the data plane. Similarly, the controller can also scale down the capacity if it falls below a low threshold.

Analytics execution with Pocket. We now describe how a serverless analytics job interacts with Pocket using Fig. 2. When the job first starts, it registers itself with the control plane (①), either specifying the amount of memory resources it expects to use, or providing hints that Pocket can use to estimate it. The controller uses this information to allocate resources for the job, and informs the metadata plane regarding the resource placement at the data plane (②). When the job's serverless compute tasks first attempt to read or write intermediate data on the job's allocated memory, it contacts the metadata service to get the IP addresses for the memory servers with its allocated resources (ⓐ). The serverless tasks can subsequently access data directly from the memory servers (ⓑ). Once the job is finished, it deregisters itself at the control plane to release its resources (③).

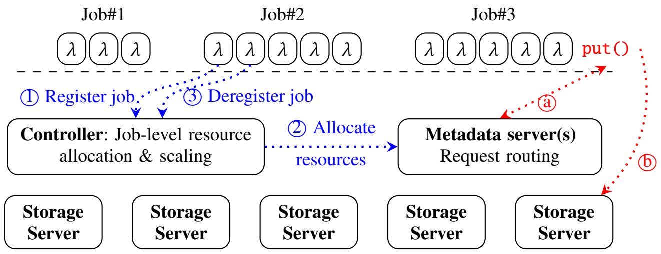


Fig. 2. Pocket Architecture. Steps ①-③ show how jobs are registered, resources are allocated and jobs are deregistered, while ⓐ, ⓑ show how operations are routed from serverless compute tasks to servers in the data plane via metadata servers (Fig. adapted from [8]).

For the remainder of the paper, we do not focus on the issues that Pocket has already addressed—Jiffy uses the same centralized management mechanism, can add or remove memory servers when all the capacity is utilized and uses the same analytics execution pipeline as Pocket. Instead, we focus on the specific problems arising out of Pocket's resource allocation mechanisms, which we describe next.

2.1 Limitations of Pocket Resource Allocation

The core challenge in Pocket's resource allocation is that it allocates memory at the granularity of jobs. Upon submission, the job specifies its memory demands; and, Pocket allocates and reserves memory resources equal to the job's demand for the entirety of its lifetime (see Fig. 1 in [8]), only releasing them when the job explicitly deregisters.

Such job-level resource allocation is problematic due to two reasons. First, accurately predicting intermediate data sizes is hard for many analytics jobs. Analysis of production workloads [20] have shown that intermediate data sizes have little or no correlation with the amount of persistent data read or the expected execution query time. Indeed, a job's intermediate data size depends on its execution plan, which, in turn, can be adapted dynamically by a query planner [41].

Second, even if one could accurately predict the intermediate data sizes, Pocket's resource allocation mechanism requires jobs to specify their demands at the time of the submission (note that hints in Pocket are only used for sharing resources across jobs, not for dynamically changing the memory capacity allocated to individual jobs). This leads to the standard tradeoff between performance degradation and resource underutilization: if jobs specify their average demand, their performance degrades when instantaneous demand is

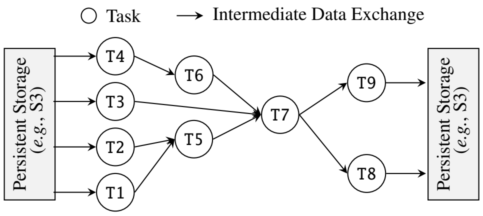

○ Task
⟶ Intermediate Data Exchange
Persistent Storage (e.g., S3)
T4
T3
T2
T1
T6
T5
T7
T9
T8
Persistent Storage (e.g., S3)

Fig. 3. Execution DAG example for a typical analytics job. Intermediate data exchange across tasks occurs via Jiffy.

higher than their average (due to read/write requests being executed on slower secondary storage) and if jobs specify their peak demand, the system suffers from resource underutilization when their instantaneous demand is lower than the peak. Since intermediate data sizes naturally increase and decrease over time (as tasks in different stages are executed), Pocket's job-level resource allocation will result in either performance degradation or resource underutilization. While we have already seen this for the Snowflake workload in Fig. 1(b), similar observations have been made in prior studies for other workloads [7, 57] as well, *e.g.*, the intermediate data size across various stages in a typical TPC-DS query [58] ranges from 0.8MB to 66GB, a difference of 5 orders of magnitude!

3 Jiffy Design

Jiffy enables fine-grained sharing of far-memory capacity across concurrently running serverless analytics jobs for storing intermediate data. Inspired by virtual memory, Jiffy partitions the memory capacity into fixed-sized blocks (akin to virtual memory pages), and performs memory allocations at the granularity of these blocks. This allows Jiffy to achieve two desirable properties. First, multiplexing the available capacity at block granularity allows Jiffy to match instantaneous job demands at seconds timescales. Second, Jiffy does not require jobs to know (even an estimate of) intermediate data sizes a priori; as tasks write/delete data, Jiffy dynamically allocates/deallocates resources at block granularity.

Remark. Multiplexing available memory capacity is different from scaling the capacity of the memory pool. Prior systems, including Pocket, focus on the latter: since resource allocation is done at job granularity, as jobs arrive or finish, these systems add and remove the memory servers to elastically scale the system capacity. However, existing capacity may be underutilized since a job may not be using memory allocated to it. Jiffy focuses on the former: efficiently sharing the capacity available at any given time across concurrently running jobs. When the memory capacity utilization is high (i.e., many jobs are actually using the capacity), Jiffy can add memory servers to scale up the capacity similar to Pocket. Interestingly, by efficiently multiplexing the available capacity across concurrently running jobs, Jiffy also reduces the frequency at which memory servers need to be added/removed from the pool.

As discussed in §1, enabling fine-grained resource allocation requires resolving four unique challenges introduced by serverless analytics. In this section, we describe how Jiffy employs hierarchical addressing (§3.1), data lifetime management (§3.2) and flexible data repartitioning (§3.3) to resolve these challenges. To assist our discussion, we will use the example in Fig. 3, which shows the execution plan for a representative analytics job. The plan is organized as a directed acyclic graph (DAG) where nodes correspond to computation tasks (implemented as serverless functions[2]), while edges denote intermediate data exchange between them via Jiffy.

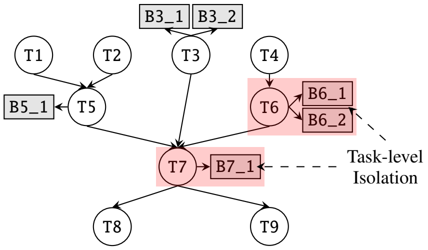


Fig. 4. Hierarchical addressing for the job in Fig. 3. Jiffy provides task-level resource isolation for far-memory under each task address-prefix (§3.1). Note that block addresses are only assigned to address-prefixes with currently allocated blocks; for tasks `T1`, `T2` and `T4`, blocks are not stored in Jiffy, but read from persistent storage.

3.1 Hierarchical Addressing

Analytics job are usually organized around multiple stages or a directed acyclic graph. In serverless analytics, where compute elasticity is a first-class primitive, each job may execute tens to thousands of individual tasks [6–20]. Thus, performing fine-grained resource allocation requires an efficient mechanism to keep an up-to-date mapping between tasks and memory blocks allocated to individual tasks. Moreover, the number of tasks reading and writing to the shared memory can change rapidly. Under such high concurrency and churn, it becomes important to provide isolation at the granularity of individual tasks: arrival and departure of a task should not affect the resources allocated to other tasks, even from the same job (since it can degrade the *overall* job performance). In this subsection, we describe Jiffy's hierarchical addressing — a simple, effective, mechanism that enables Jiffy to maintain a mapping between individual tasks and memory blocks allocated to these tasks, as well as provide isolation at individual task granularity.

Motivated by the Internet hierarchical IP addressing mechanism that captures *network structure*, Jiffy employs a hierarchical addressing mechanism that captures *execution structure* in analytics jobs. Specifically, Jiffy organizes intermediate data for analytics jobs within a "virtual" address hierarchy to capture the dependencies between intermediate data for different tasks. We provide an example below, but conceptually, internal nodes in the hierarchy correspond to tasks in the DAG, while leaf nodes correspond to Jiffy blocks storing

[2] Functions refer to a basic computation unit in serverless architectures, *e.g.*, Amazon Lambdas [1], Google Functions [3], Azure Functions [2], etc.

intermediate data generated by the tasks. Blocks form the final layer of the hierarchy: block addresses are defined by the path used to reach it in the hierarchy. The immediate address prefix of a block, therefore, identifies the task that generated it. Finally, the edges between internal nodes capture the dependencies between the intermediate data generated by them. To construct the address hierarchy, Jiffy uses the execution plan for a job (*e.g.*, using AWS Step Function and Azure Durable Function, or via explicit workflow specification from the job). Otherwise, Jiffy initializes the hierarchy to a single node, and *deduces* the rest on-the-fly based on the intermediate data dependencies between the job's tasks (during registration of individual tasks using Jiffy API §4.1), albeit by using more computation at the control plane; this allows Jiffy to support dynamic query plans, where the DAG is not known a priori.

Example. Fig. 4 shows the address hierarchy for the job from Fig. 3. The internal nodes `T1`-`T9` correspond to tasks in the DAG, while leaf nodes `B3_1`, `B3_2`, etc., correspond to the data blocks allocated to them by Jiffy for storing their intermediate data. Edges (`T1`, `T5`) and (`T2`, `T5`) in the address hierarchy indicate that the intermediate data in `T5` depends on the intermediate data from both `T1` and `T2`. The complete address of block `B6_2` under `T6` would be `T4.T6.B6_2`, while the address-prefix `T4.T6` identifies all blocks allocated to `T6`. Note that a block can have multiple addresses, *e.g.*, block `B7_1` can be addressed using `T4.T6.T7.B7_1`, `T3.T7.B7_1`, `T2.T5.T7.B7_1` and `T1.T5.T7.B7_1`. This is similar to inode hierarchy in POSIX filesystems — just as an inode may be linked by many directories, and thus may have many pathnames, a block may have many addresses.

Organizing intermediate data across an address hierarchy allows Jiffy to manage resource allocations for an address-prefix independent of other prefixes. Specifically, if the memory in a specific address-prefix spills over to persistent storage, it does not affect the performance of other address-prefixes. Moreover, Jiffy ensures that once a block is allocated to an address-prefix, it will not be reclaimed until the application either explicitly reclaims it, or stops renewing leases for it (§3.2), affording *isolation* at address-prefix granularity. Since address-prefixes correspond to tasks in Jiffy address hierarchy, this enables task-level isolation regardless of task concurrency and churn. This is similar to virtual memory, where each process is is assigned its own virtual address space that enables isolation at process granularity; Jiffy does this at individual task granularity using hierarchical addressing that captures the execution structure of the job.

We outline two important design issues. First, Jiffy's fine-grained resource allocation should be decoupled from the policies required to enforce desired system behavior. For instance, algorithms to achieve fairness in resource allocation across various jobs or tenants can be easily integrated on top of Jiffy allocation mechanism. This is orthogonal to Jiffy's goals of enabling fine-grained sharing (where the overall goal is high resource utilization). Second, address translation — the mapping from virtual addresses to physical memory blocks — is performed similar to Pocket [8] at the centralized metadata server. Thus, unlike hardware address translation that imposes a limit on the size (depth and breadth) of the execution DAG, Jiffy can easily perform addressing for arbitrary DAGs. Jiffy's hierarchical addressing and fine-grained resource allocation does introduce additional complexity at the controller; we evaluate the scalability of our controller implementation in the evaluation section and demonstrate that Jiffy can still scale to ~45K requests per second per core, which is large enough for most realistic deployments.

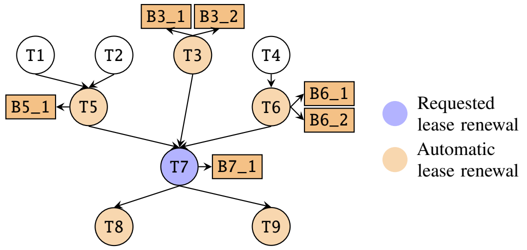


Fig. 5. Lease Renewal via Address Hierarchy. Hierarchical addressing simplifies lease renewal in Jiffy (§3.2), since lease renewal for an address-prefix automatically implies renewals for all parent and descendent address-prefixes in the hierarchy.

Block sizing. Similar to page-size in traditional virtual memory, block-size in Jiffy exposes well-known tradeoffs between the amount of metadata that needs to be stored at the control plane, and memory utilization. In particular, larger block-sizes reduce the amount of per-block metadata at the control plane, at the cost of potentially reduced memory utilization from data fragmentation within blocks, and vice versa. We note that Jiffy does not suffer the traditional overheads of higher I/O with larger blocks, since it supports fine-grained access within blocks via its data structure interface (§4.1). Moreover, Jiffy also controls under-utilization within a block via data repartitioning, as described in §3.3. While the system block size can be configured during initialization in Jiffy (§6.6), it employs a block size that is typically used for files in analytics frameworks (*e.g.*, 128MB in HDFS [59]) for compatibility.

Isolation granularity. Since the nodes in the address hierarchy correspond to tasks, Jiffy provides task-level isolation. It is, however, possible to provide finer or coarser-grained isolation by simply adding another layer to the hierarchy (*e.g.*, for isolation at the granularity of tables in data lakes) or removing a layer from the hierarchy (*e.g.*, for stage-level isolation in MapReduce frameworks). We chose task-level isolation as our default since most analytics frameworks stand to benefit from task-level isolation, but do not require finer-grained isolation. Individual applications can, however, chose to create custom hierarchies using Jiffy API, as we outline in §4.1.

3.2 Data Lifetime Management

Existing far-memory systems for serverless analytics that perform job-level resource allocation, also manage data lifetimes at job granularity — reclaiming storage when the job explicitly deregisters. A unique feature of serverless analytics is that a task's intermediate data is decoupled from its execution: while its execution occurs at the serverless compute platform, the data generated and consumed by it would reside at the far-memory system. This also results in the decoupling of their fault domains: with standard mechanisms (*e.g.*, reference counting approaches [60–62]), when the task fails, its corresponding intermediate data becomes dangling state at the far-memory system. To avoid the resulting inefficiency, we need additional mechanisms to efficiently perform task-level data lifetime management.

Jiffy achieves this by integrating well-known lease management mechanisms [63–65] with hierarchical addressing to enable lifetime management of intermediate data. In particular, Jiffy associates each address-prefix in a job's hierarchical addressing with a lease, and only keeps its data in memory as long as its lease is renewed. Consequently, a job periodically renews leases for the address-prefixes of tasks that are currently running. Jiffy tracks the time a lease was last renewed for each node in the address hierarchy, and updates it for the relevant nodes when a new renewal request for a particular address-prefix is received. Finally, on lease expiry for a particular address-prefix, Jiffy reclaims all memory allocated to it after flushing the data to persistent storage. This ensures that even if a lease expires due to network delays between the task and Jiffy, the data is not lost.

The new aspect of lease management in Jiffy is that it exploits the DAG-based hierarchical addressing to determine dependencies between leases. On receiving a lease renewal request from a task that is currently running, Jiffy renews leases not only for its address-prefix, but also for the prefixes of tasks that they depend on (i.e., parent nodes in the hierarchy), and for all the prefixes of tasks that depend on it (i.e., all descendant nodes in the hierarchy). This significantly reduces the number of lease renewal messages that a job has to send. More over, this also ensures that while a task is running, not only is its own intermediate data kept in memory, but also the data for all the tasks that it depends on, and all the tasks that depend on it. In particular, if a task fails but its dependent task is still alive and renewing leases, then Jiffy will still keep the data corresponding to the failed task in memory so that the dependent task can continue operating on it.

Jiffy's leasing mechanism finds a favorable tradeoff between age-based eviction (*e.g.*, in caching approaches, where jobs have no control on data lifetime) and explicit acquisition and release (where jobs have full control, but job failures could lead to orphaned state). Jiffy's mechanism not only provides jobs control over the lifetime of their memory resources (via explicit leases), but also ties the fate of the allocated resources to the job — if a lease is not renewed (*e.g.*, due to job or task failure, or if resources are no longer needed), Jiffy reassigns resources to other jobs or tasks on lease expiry.

Example. In Fig. 3, task `T7` periodically renews leases for the prefix `T4.T6.T7`[3] during its execution — Jiffy keeps the intermediate data for the blocks under it in memory as long as the job renews leases for it. Moreover, a lease renewal for task `T7`'s prefix also renews leases for its parent tasks prefixes (i.e., for `T3`, `T5`, `T6`) and for its descendent task prefixes (i.e., for `T8`, `T9`), as shown in Fig. 5. Thus, renewing `T7`'s lease ensures that `T7` can still use the intermediate data generated by its parent tasks; moreover, if any of `T7`'s downstream tasks are active, their intermediate data is automatically kept in memory as well. Note that leases for tasks `T1`, `T2` and `T4` are not automatically renewed, since they are no longer active and `T7` does not require the data generated by them.

Lease duration. Lease duration in Jiffy exposes a tradeoff between control plane bandwidth and system utilization *over time*. Specifically, longer lease durations reduce the network traffic to the control plane since jobs renew their leases at coarser granularities, but reduce system utilization since Jiffy does not reclaim (potentially unused) resources from jobs until their leases expire. We evaluate Jiffy's sensitivity to lease durations in §6.6.

3.3 Flexible Data Repartitioning

Decoupling compute tasks from their intermediate data in serverless analytics makes it challenging to efficiently achieve memory elasticity at fine granularities. Specifically, as memory is allocated/deallocated to a task, the intermediate data needs to be repartitioned across the remaining memory blocks. However, the decoupling of compute tasks from their intermediate data and the large number of concurrent tasks makes it impractical to offload this repartitioning to the application. For instance, many existing serverless analytics approaches [7, 8] employ key-value stores to store intermediate data. In such a setting, if the compute task were to repartition the intermediate data on memory scaling, it would have to first read the key-value pairs from the store over the network, compute the data partitions across the new memory allocation, and write back the data to the store. This would incur significant network latency and bandwidth overheads for the task.

As we discuss in §5, Jiffy already implements standard data structures used in data analytics frameworks — *e.g.*, files [10, 16–18, 20], to key-value pairs [6–8, 11, 13, 15, 19] to queues [9, 12]. Analytics jobs using these data structures can offload repartitioning of intermediate data upon resource allocation/deallocation to Jiffy. Each block allocated to a Jiffy data structure tracks the fraction of the block memory capacity that is currently being used to store data. Whenever the usage grows above a high threshold, Jiffy, in turn, allocates a

[3] Note that since task `T7` has four different address-prefixes, the job can renew leases for its data using any of them.

Table 1. Jiffy User-facing API. See §4.1 for details.

	API	Description
	`connect(jiffyAddress)`	Connect to Jiffy.

Address Hierarchy	`createAddrPrefix(addr, parent, optionalArgs)`
	`createHierarchy(dag, optionalArgs)`
	`flushAddrPrefix(addr, externalPath)`
	`loadAddrPrefix(addr, externalPath)`
		Create address-prefix `addr` with given `parent` address-prefix and `optionalArgs` (*e.g.*, initial capacity), or, create address hierarchy from execution plan provided as a DAG `dag`.
		Flush/load data in address-prefix to external persistent store.

	`leaseDuration = getLeaseDuration(addr)`
	`renewLease(addr)`
		Get the lease duration associated with address-prefix `addr`.
		Send lease renewal request for address-prefix `addr`.

Data Structure	`ds = initDataStructure(addr, type)`	Initialize data structure of given `type` in address-prefix `addr` and get handle `ds` that encapsulates physical locations of allocated blocks.
	Data structure-specific interface implemented using block API (Fig. 6).	See Table 2 in §5 for examples.

	`listener = ds.subscribe(op)`
	`notif = listener.get(timeout)`
		Subscribe to notifications for operations of type `op` on `ds`.
		Get latest notification; waits `timeout` seconds for response.

new block to the corresponding address-prefix[4]. Subsequently, the overloaded block triggers data structure-specific repartitioning to move part of its data to the new block. Similarly, when the block usage drops below a low threshold, Jiffy identifies another block in the address-prefix with low-usage with which the block can merge its data. The block then conducts the required repartitioning, after which Jiffy deallocates it. Note that by having the target block conduct the repartitioning instead of the compute task, Jiffy avoids the network and computational overheads for task itself. Finally, we note that data repartitioning occurs asynchronously in Jiffy: data access operations across data structure blocks can proceed even while repartitioning is in progress. This allows Jiffy to ensure application performance is minimally impacted due to data repartitioning (§6.3).

Data structures included in Jiffy already allow us to implement serverless incarnations of several powerful distributed programming frameworks on top of Jiffy: MapReduce [53, 67], Dryad [54], StreamScope [55] and Piccolo [56]. We note that data structures used in analytics frameworks — files, queues, key-value stores — require very simple repartitioning mechanisms (unlike data structures like B-trees or other ordered trees that are not used in data analytics frameworks). As such, serverless applications employing these programming models can run on Jiffy and leverage its flexible data repartitioning without any modification.

Thresholds for elastic scaling. The high and low thresholds for elastic scaling in Jiffy expose a tradeoff between the data plane network bandwidth and task performance on one hand, and system utilization on the other. Specifically, if the high and low thresholds are set high and low enough, respectively, then elastic scaling is triggered rarely, reducing the amount of network traffic due to data repartitioning. At the same time, extreme threshold values also negatively affect system utilization within the blocks, *e.g.*, lower low-thresholds result in larger number of nearly empty blocks. We evaluate Jiffy's sensitivity to the choice of thresholds in §6.6.

4 Jiffy Implementation

Jiffy inherits Pocket's scalable and fault-tolerant metadata plane, system-wide capacity scaling, analytics execution model, etc. However, Jiffy implements hierarchical addressing, lease management and efficient data repartitioning (§3) to resolve unique challenges introduced by serverless environments. We now describe Jiffy interface (§4.1) and implementation (§4.2.1) focusing on these new features.

4.1 Jiffy Interface

We describe Jiffy interface in terms of its user-facing API (Table 1) and internal API (Fig. 6).

User-facing API. Jiffy's user-facing interface (Table 1) is divided along its two core abstractions: *hierarchical addresses* and *data structures*. Jobs add a new address-prefix to their address hierarchy using `createAddrPrefix`, specifying the parent address-prefix, along with optional arguments such as initial capacity. Jiffy also provides a `createHierarchy` interface to directly generate the complete address hierarchy from the application's execution plan (i.e., DAG), and `flush`/`load` interfaces to persist/load address-prefix data from external storage (*e.g.*, S3). Jiffy provides three built-in data structures that can be associated with an address-prefix (via `initDataStructure`), and a way to define new data structures using its internal API.

Similar to existing systems [52, 68], data structures also expose a notification interface, so that tasks that consume intermediate data can be notified on data availability. For instance, a task can `subscribe` to write operations on its parent task's data structure, and obtain a `listener` handle. Jiffy asynchronously notifies the `listener` upon a write to the data structure, which the task can get via `listener.get()`.

[4] Similar to existing systems [8, 50, 52, 66], Jiffy can trivially scale its cluster capacity: if the number of free blocks available increase/decrease beyond a certain threshold, Jiffy adds/removes servers to adjust physical memory resources. Here, we focus only on fine-grained elasticity.

```
block = ds.getBlock(op, args) // Get block
block.writeOp(args) // Perform write
data = block.readOp(args) // Perform read
block.deleteOp(args) // Perform delete
```

Fig. 6. Jiffy Internal API. The block interface is used internally in Jiffy to implement the data structure APIs (§5).

Internal API. The data layout within blocks in Jiffy is unique to the data structure that owns it. As such, Jiffy blocks expose a set of data structure *operators* (Fig. 6) which uniquely defines how data structure requests are *routed* across their blocks, and how data is *accessed* or *modified*. These operators are used internally within Jiffy for its built-in data structures (§5) and not exposed to jobs directly.

The `getBlock` operator determines which block an operation request is routed to based on the operation type and operation-specific arguments (*e.g.*, based on key hashes for a KV-store), and returns a handle to the corresponding block. Each Jiffy block exposes `writeOp`, `readOp` and `deleteOp` operators to facilitate data structure-specific access logic (*e.g.*, `get`, `put` and `delete` for KV-store). Jiffy executes individual operators *atomically* using sequence numbers, but does not support atomic transactions that span multiple operators.

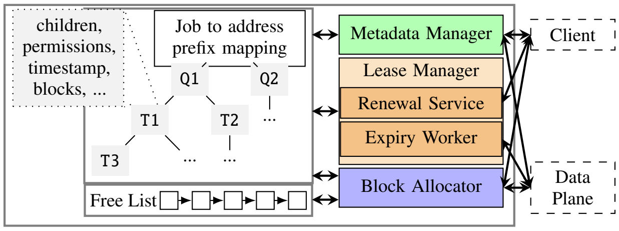


Fig. 7. Jiffy controller. See §4.2.1 for details.

4.2 System Implementation

Jiffy's high-level design components similar to Pocket's, except for one difference: Jiffy combines the control and metadata planes into a unified control plane. We found this design choice allowed us to significantly simplify interactions between the control and metadata components, without affecting their performance. While this does couple their fault-domains, standard fault-tolerance mechanisms are still applicable to the unified control plane.

4.2.1 Control plane

The Jiffy controller (Fig. 7) maintains two pieces of system-wide state. First, it stores a *free block list*, which lists the set of blocks that have not been allocated to any job yet, along with their corresponding physical server addresses. Second, it stores an address hierarchy per-job, where each node in the hierarchy stores a variety of metadata for its address-prefix, including access permissions (for enforcing access control), timestamps (for lease renewal), a block-map (to locate the blocks associated with the address-prefix in the data plane), along with metadata to identify the data structure associated with the address-prefix and how data is partitioned across its blocks. The mapping between jobIDs (which uniquely identify jobs) and their address hierarchies is stored in a hash-table at the controller.

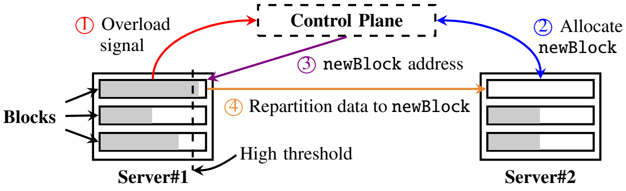


Fig. 8. Data repartitioning on scaling up capacity. Scaling down capacity employs a similar approach (§4.2.2).

Block allocator. When a job creates an address-prefix in Jiffy, the block allocator at the control plane assigns it the number of blocks corresponding to the requested initial capacity from its pool of free blocks. While assigning the blocks, the controller updates its state: the free block list, access permissions and block-map for that address-prefix. Assignment of blocks across address-prefixes is akin to virtual memory in traditional operating systems: Jiffy *multiplexes* its physical memory pools at the data plane across different prefixes at block-granularity, while individual tasks operate under the illusion that their prefixes have infinite memory resources.

Metadata manager. The metadata manager tracks the partitioning information specific to different data structures (§5) and assists clients in maintaining a consistent view of how the data is organized across the blocks allocated to each data structure. We defer the discussion of data structure-specific metadata stored at the control plane to §5, but note that this metadata is updated whenever blocks allocated to an address-prefix is scaled. A client detects that a scaling has occurred when it queries the data plane, and updates its view of the partitioning metadata by querying the control plane.

Lease manager. The lease manager implements lifetime management in Jiffy. It comprises a lease renewal service that listens for renewal requests from jobs and updates the lease renewal timestamp of relevant nodes in its address hierarchy, and a lease expiry worker that periodically traverses all address hierarchies, marking nodes with timestamps older than the associated lease period as expired.

Controller scaling and fault tolerance. In order to scale the control plane, Jiffy can employ multiple controller servers, each managing control operations for a non-overlapping subset of address hierarchies (across jobs) and blocks (across memory servers at the data plane). Jiffy employs hash-partitioning to distribute both address-prefixes and memory blocks (via their blockIDs) across controller servers. Moreover, Jiffy employs the same approach to scale its control plane to multiple cores on a multi-core server. Jiffy adopts

Table 2. Jiffy Data Structure Implementations. See §5 for details.

Data Structure		Operators				
		`writeOp`	`readOp`	`deleteOp`	`getBlock`	`repartition`
Built-in	File (§5.1)	`write`	`read`	-	Route to block based on file offsets.	Not required
	FIFO Queue (§5.2)	`enqueue`	`dequeue`		`enqueue` to tail, `dequeue` to head block.	Not required
	KV-Store (§5.3)	`put`	`get`	`delete`	Route to block based on key hash.	Hash-based repartitioning
Custom data structures.						

primary-backup based mechanisms from prior work [8, 69] at each controller server for fault-tolerance.

4.2.2 Data plane

Jiffy data plane is responsible for two main tasks: providing jobs with efficient, data-structure specific atomic access to data, and repartitioning data across blocks allocated by the control plane during resource scaling. It partitions the resources in a pool of memory servers across fixed sized blocks. Each memory server maintains, for the blocks managed by it, a mapping from unique blockIDs to pointers to raw memory allocated to the blocks, along with two additional metadata: data structure-specific operator implementations as described in §4.1, and a subscription map that maps data structure operations to client handles that have subscribed to receive notifications for that operation.

We implement a high-performance RPC layer at the data plane using Apache Thrift [70] for interactions between clients and memory servers. While Thrift already provides low-overhead serialization/deserialization protocols, we add two key optimizations at the RPC layer. First, our server-side implementation employs asynchronous framed IO to multiplex multiple client sessions, permitting requests across different sessions to be processed in a non-blocking manner for lower latency and higher throughput. Second, while our client-side library is implemented in Python for compatibility with AWS lambda, it employs thin Python wrappers around Thrift's C-libraries to minimize performance overheads.

Data repartitioning for a Jiffy data structure is implemented as follows: when a block's usage grows above the high threshold, the block sends a signal to the control plane, which, in turn, allocates a new block to the address-prefix and responds to the overloaded block with its location. The overloaded block then repartitions and moves part of its data to the new block (see Fig. 8); a similar mechanism is used when the block's usage falls below the low threshold.

For applications that require fault tolerance and persistence for their intermediate data, Jiffy supports chain replication [71] at block granularity, and synchronously persisting data to external stores (*e.g.*, S3) at address-prefix granularity.

5 Programming Models on Jiffy

We now describe how Jiffy's built-in data structures (Table 2) enable many distributed programming frameworks atop serverless platforms (§5.1-§5.3).

5.1 Map-Reduce Model

A Map-Reduce (MR) program [53] comprises map functions that process a series of input key-value (KV) pairs to generate intermediate KV pairs, and reduce functions that merge all intermediate values for the same intermediate key. MR frameworks [53, 67, 72] parallelize map and reduce functions across multiple workers. Data exchange between map and reduce workers occurs via a shuffle phase, where intermediate KV pairs are distributed in a way that ensures values belonging to the same key are routed to the same worker.

MR on Jiffy executes map/reduce tasks as serverless tasks. A master process launches, tracks progress of, and handles failures for tasks across MR jobs. Jiffy stores intermediate KV pairs across multiple shuffle files, where shuffle files contain a partitioned subset of KV pairs collected from all map tasks. Since multiple map tasks can write to the same shuffle file, Jiffy's strong consistency semantics ensures correctness. The master process handles explicit lease renewals.

Jiffy Files. A Jiffy file is a collection of blocks, each storing a fixed-sized chunk of the file. The controller stores the mapping between blocks and file offset ranges managed by them at the metadata manager; this mapping is cached at clients accessing the file, and updated whenever the number of blocks allocated to the file is scaled in Jiffy. The `getBlock` operator forwards requests to different file blocks based on the offset-range for the request. Files support sequential `read`s, and `write`s via append-only semantics. For random access, files support `seek` with arbitrary offsets. Jiffy uses the provided offset to identify the corresponding block, and forwards subsequent `read` requests to it. Finally, since files are append-only, blocks can only be added to it (not removed), and do not require repartitioning when new blocks are added.

5.2 Dataflow and Streaming Dataflow Models

In the dataflow programming model, programmers provide DAGs to describe an application's communication patterns. DAG vertices correspond to computations, while data channels form directed edges between them. We use Dryad [54] as a reference dataflow execution engine, where channels can be files, shared memory FIFO queues, etc. Dryad runtime schedules DAG vertices across multiple workers based on their dataflow dependencies. A vertex is scheduled when all its input channels are ready: a file channel is ready if all its data items have been written, while a queue is ready if it has any data item. Streaming dataflow [55] employs a similar approach, except channels are continuous event streams.

Dataflow on Jiffy maps each DAG vertex to a serverless task, while a master process handles scheduling, fault-tolerance and lease renewals for Jiffy. We use Jiffy FIFO queues and files as data channels. Since queue-based channels are considered ready as long as some vertex is writing to it, Jiffy allows downstream tasks to efficiently detect if items produced by upstream tasks are available via notifications.

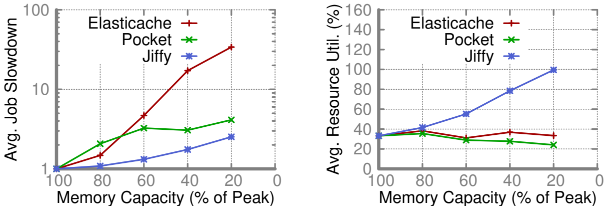
Jiffy Queues. A FIFO queue in Jiffy is a continuously growing linked-list of blocks, where each block stores multiple data items, and a pointer to the next block in the list. The queue size can be upper-bounded (in number of items) by specifying a `maxQueueLength`. The controller only stores the head and the tail blocks in the queue's linked list, which the client caches and updates whenever blocks are added/removed. The queue supports `enqueue`/`dequeue` to add/remove items. The `getBlock` operator routes `enqueue` and `dequeue` operations to the current tail and head blocks in the link-list, respectively. While, blocks can be both added and removed from a queue, queues do not need subsequent data repartitioning. Finally, the queue leverages Jiffy notifications to asynchronously detect when the there is data in the queue to consume, or space in the queue to add more items, via subscriptions to `enqueue` and `dequeue`, respectively.

5.3 Piccolo

Piccolo [56] is a data-centric programming model that allows distributed compute machines to share distributed, mutable state. Piccolo kernel functions specify sequential application logic and share state with concurrent kernel functions via a KV interface, while centralized control functions create and coordinate both shared KV stores and kernel function instances. Concurrent updates to the same key in the KV store are resolved using user-defined accumulators.

Piccolo on Jiffy runs kernel functions across serverless tasks, while control tasks run on a centralized master. The shared state is stored across Jiffy's KV-store data structures (described below). KV-stores may be created per kernel function, or shared across multiple functions, depending on the application needs. The master periodically renews leases for Jiffy KV-stores. Like Piccolo, Jiffy checkpoints KV-stores by flushing them to an external store.

Jiffy KV-store. The Jiffy KV-store hashes each key to one of H hash-slots in the range $[0, H\text{-}1]$ (H=1024 by default). The KV-store shards KV pairs across multiple Jiffy blocks, such that each block owns one or more hash-slots in this range. Note that a hash-slot is completely contained in a single block. The controller stores the mapping between the blocks and the hash slots managed by them; this metadata is, again, cached at the client and updated during resource scaling. Each block stores KV pairs that hash to its slots as a hash-table — Jiffy employs cuckoo hashing [73] for highly concurrent KV operations. The KV-store supports typical `get`, `put`, and `delete` operations as implementations of `readOp`, `writeOp` and `deleteOp` operators. The `getBlock` operator routes requests to KV-store blocks based on key-hashes.



(a) Job performance (b) Resource utilization

Fig. 9. Fine-grained task-level elasticity in Jiffy enables (a) better job performance, and (b) higher resource utilization under constrained capacity. In (a), the slowdown is computed relative to the job completion time with 100% capacity (for this data point, Elasticache performance was 30% worse than Pocket, and Pocket performance was 5% worse than Jiffy). See §6.1 for details.

Unlike files and queues, data needs to repartitioned for the KV-store when a block is added or removed. When a block is close-to-full, Jiffy reassigns half of its hash-slots to a new block, moves the corresponding key-value pairs to it, and updates the block-to-hash-slot mapping at the controller. Similarly, when a block is nearly empty, its hash-slots are merged with another block.

6 Evaluation

Jiffy is implemented in 25K lines of C++, with client libraries in C++, Python and Java (~1K LOC each). In this section, we evaluate Jiffy to demonstrate its benefits (§6.1, §6.2) and to understand the contribution of individual Jiffy mechanisms to its overall performance (§6.3). We evaluate Jiffy controller overheads in §6.4, additional serverless workloads in §6.5, and Jiffy's sensitivity to various system parameters in §6.6.

Experimental setup. Unless otherwise specified, each evaluated system is deployed across 10 m4.16xlarge EC2 [74] instances, while serverless applications are deployed across AWS Lambda [74] instances. Since Jiffy leverages Pocket's design, it supports addition of new instances to increase the system capacity. However, our experiments do not evaluate overheads for doing so, since, it is orthogonal to Jiffy's goals; we specifically focus on multiplexing available capacity for higher utilization and to reduce the need for adding more capacity. Jiffy employs 128MB blocks, 1s lease duration and 5% (low) and 95% (high) as thresholds for data repartitioning.

6.1 Benefits of Jiffy

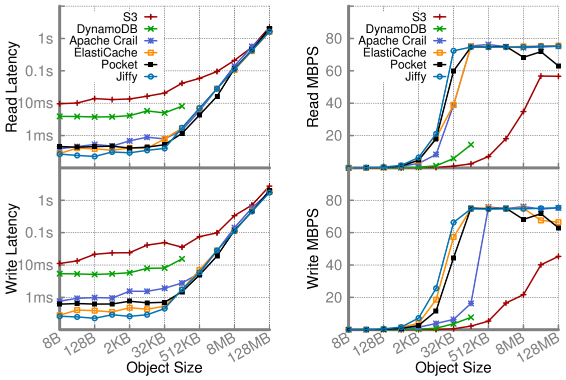
Jiffy enables fine-grained resource allocation for serverless analytics. We demonstrate the benefits of this approach to job performance and resource utilization for roughly 50,000 jobs across 100 randomly chosen tenants over a randomly selected 5 hour window in the Snowflake workload[5] [20]. We compare Jiffy (with the MR programming model, §5) against

[5] We were unable to evaluate the entire 14 day window with > 2000 tenants due to intractable cost overheads.

Elasticache [66] and Pocket [8]. Elasticache represents systems that provision resources for *all* jobs. Since Elasticache does not support multiple storage tiers, if available capacity is insufficient, jobs must write their data to external stores like S3 [75]. Pocket, on the other hand, reserves and reclaims resources at *job* granularity; if available capacity is insufficient, Pocket allocates resources on secondary storage (SSD). Note that Pocket's utilization can sometimes be *lower* than Elasticache, since it provisions for the peak of each job *separately*, sacrificing utilization for job-level isolation. Finally, we place Pocket's control and metadata services on the same server to ensure a fair comparison with Jiffy's unified control plane.

Impact of fine-grained elasticity on job performance. We demonstrate this impact by constraining the amount of available capacity at the intermediate store for the Snowflake workload. Fig. 9(a) shows the average job slowdown as the capacity is reduced to a certain percentage of the peak usage for the workload within the evaluated time window (i.e., across all jobs). Note that the peak usage can be several orders of magnitude larger that the average requirements for each tenant, so provisioning for the peak would be quite wasteful; ideally, we want the provisioned capacity to be as small as possible without much degradation in performance. Unfortunately, with Elasticache, job performance suffers significantly as the intermediate data grows larger than capacity (4.7× slowdown at 60% of peak and 34× slowdown at 20%), since the data must now be accessed from S3. With Pocket, the data spills to SSD when the allocated capacity at the DRAM-tier (during job registration) is insufficient. While the slowdown is less severe due to its efficient tiered-storage, jobs still experience a 3.2× slowdown at 60% of peak and $>$ 4.1× slowdown at 20% of peak. Finally, Jiffy observes much lower job performance degradation with constrained capacity (1.3× at 60% of peak and $<$ 2.5× at 20% of peak). In particular, Jiffy improves job execution time by $1.6-2.5\times$ compared to Pocket at different memory capacities. This is because task-level elasticity and lease based reclamation of memory allows Jiffy to efficiently multiplex capacity across multiple jobs at a much finer granularity than Pocket. This, in turn, significantly reduces data spilling over to a slower storage tier in Jiffy compared to Pocket. We confirm this intuition further by studying the impact of fine-grained elasticity on resource utilization next.

Impact of fine-grained elasticity on resource utilization. Fig. 9(b) shows the resource utilization across the compared systems under constrained capacity. While the resource utilization for Elasticache and Pocket either decreases or remains the same as the system capacity is reduced, resource utilization *improves* for Jiffy. This is because Pocket and Elasticache provision capacity (at job or coarser granularity), and the unused capacity is wasted, regardless of the total system capacity. In contrast, Jiffy is able to better multiplex the available capacity across various jobs owing to its fine-grained elasticity and lease-based reclamation of unused capacity. By making better use of available capacity, Jiffy ensures that a much smaller fraction of data spills over to SSD, resulting in better performance in Fig. 9(a).



(a) Latency

(b) MBPS

Fig. 10. Jiffy performance comparison with existing systems (§6.2). Despite providing the additional benefits demonstrated in §6.1, Jiffy performs as well as state-of-the-art systems commonly used for intermediate data storage in serverless analytics.

6.2 Performance Benchmarks for Six Systems

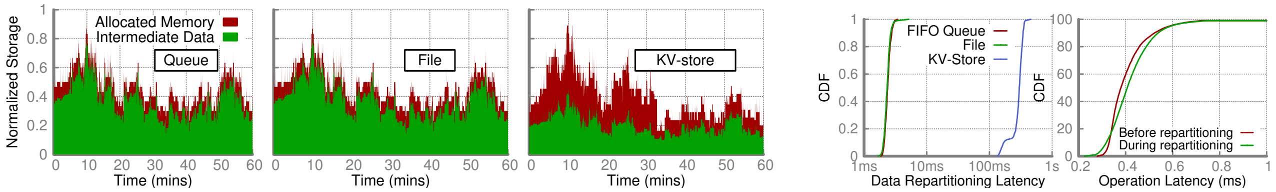
We now compare Jiffy performance (using its KV-Store data structure) against five state-of-the-art systems commonly used for intermediate data storage in serverless analytics: S3, DynamoDB, Elasticache, Apache Crail and Pocket. Since only a subset of the compared systems support request pipelining, we disable pipelining for all of them.

To measure latency and throughput for the above systems, we profiled synchronous operations issued from an AWS Lambda instance using a single-threaded client. Fig. 10 shows that in-memory data stores like Elasticache, Pocket and Apache Crail achieve low-latency (sub-millisecond) and high-throughput. In contrast, persistent data stores like S3 and DynamoDB observe significantly higher latencies and lower throughput; note that DynamoDB only supports objects up to 128KB. Jiffy matches the performance achieved by state-of-the-art in-memory data stores, while additionally providing the benefits outlined in §6.1. Note that Jiffy's performance gains over Pocket and Elasticache are due to (a) its optimized RPC layer (§4.2.2), and, (b) its use of cuckoo hashing in the KV-Store data structure (§5.3).

6.3 Understanding Jiffy Benefits

Fig. 9 already shows how fine-grained elasticity in Jiffy allows it to achieve performance and resource utilization gains over the compared state-of-the-art systems. As noted earlier, this fine-grained elasticity is enabled by hierarchical virtual addressing combined with with flexible data lifetime management and data repartitioning in Jiffy. In this section, we evaluate their impact in isolation.

Fine-grained elasticity via lifetime management. Unlike traditional storage systems, Jiffy's lease-based data lifetime



(a) Efficient lifetime management

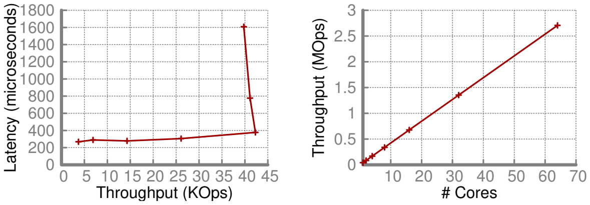
(b) Efficient data repartitioning

Fig. 11. Jiffy data lifetime-management and data repartitioning. (a) Jiffy enables fine-grained elasticity via lease-based lifetime management for its built-in data structures, FIFO Queue (left), File (center) and KV-store (right), reclaiming resources from tasks as soon as their leases expire. (b) Jiffy facilitates efficient data repartitioning for its data structures when their allocation is scaled up, with repartitioning for a single block completing in 2-500ms (left). Moreover, Jiffy latency for 100KB `gets` is minimally impacted during KV-store repartitioning. Note: plots in (a), (b) share a common y-axis; x-axis for (c, left) is in log-scale.

management allows it to reclaim unused resources from jobs, and potentially assign them to other jobs that might need them. Coupled with fine-grained resource allocations and efficient data repartitioning, this enables fine-grained elasticity for serverless jobs running on Jiffy. To understand how, we evaluate memory allocation across different Jiffy data structures (Fig. 11(a)) when subjected to the Snowflake workload.

FIFO queue and file observe seamless elasticity in allocated resources as intermediate data is written to them since they do not require repartitioning. The allocated capacity exceeds the intermediate data size for the data structures by only a small amount; this accounts for the additional metadata stored at each of the blocks (*e.g.*, object metadata for the items enqueued in the FIFO queue, etc.), along with unused space within the head/tail blocks. For the KV-store, the inserted keys were sampled from a Zipf distribution over the keyspace since the Snowflake dataset does not provide access patterns. Due to the skew, a few Jiffy blocks receive most of the key-value pairs, and repeatedly split across newly allocated blocks when their used capacity grows too high. The allocated capacity is therefore higher than the dataset size, since the used capacity is low for most blocks owing to the Zipf key sampling; this corresponds to the worst-case for the KV-Store. However, Jiffy's lease mechanism reclaims resources allocated to the data structures soon after their utility is over, ensuring that the overheads are short-lived.

Efficient elastic scaling via flexible data repartitioning. A key contributor for the fine-grained resource elasticity achieved by Jiffy is its flexible but efficient data repartitioning approach. Fig. 11(b) shows the CDF of data repartitioning latency per block across the three data structures, when subjected to the Snowflake workload from above. The data repartitioning latency shown here corresponds to the total time taken from the detection of an overloaded/underloaded block to the end of data repartitioning. The memory server takes ~1-1.5ms to connect to the controller, and two round-trips (100-200μs in EC2) to trigger allocation/reclamation of data blocks and update for partitioning metadata at the controller. Unlike FIFO Queue and File, KV-Store also requires repartitioning data across blocks. However, since repartitioning a single block only requires moving only half the block capacity (~64MB), Jiffy is able to repartition the data in a few hundred milliseconds over 10Gbps links. As such, Jiffy repartitions data within a block for its built-in data structures with very low latency (2-500ms).



(a) Controller throughput vs. latency on a single CPU core.

(b) Controller throughput scaling with multiple cores.

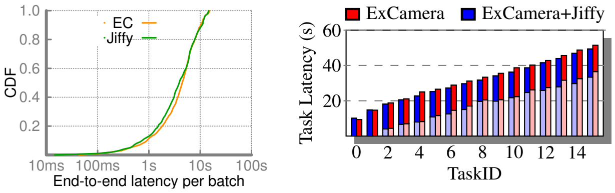
Fig. 12. Jiffy controller performance. Details in §6.4.

Finally, we also note that Jiffy does not block data structure operations during data repartitioning. In fact, these operations are minimally impacted during this period: Fig. 11(b) shows that the CDF for 100KB `get` operations on the KV-Store prior to and during scaling are almost identical.

6.4 Controller Overheads

Jiffy adds several components at the controller compared to Pocket, including all of metadata management, lease management and handling requests for data repartitioning. As such, we expect its performance to be lower than Pocket's metadata server. We deem this to be acceptable as long as it can still handle control plane request rates typically seen for real world workload, *e.g.*, a peak of a few hundred requests per second, including lease renewal requests, for all of our evaluated workloads and those evaluated in [8].

Fig. 12(a) shows the throughput-vs-latency curve for Jiffy controller operations on a single CPU core of an m4.16xlarge EC2 instance. The controller throughput saturates at roughly 42 KOps, with a latency of 370us, which is more than sufficient to handle control plane load for real-world workloads. In addition, the throughput scales almost linearly with the number of cores, since each core can handle requests independent of other cores for a distinct subset of virtual address hierarchies (Fig. 12(b)); in particular, with 64 cores, Jiffy can handle ~ 2.7 million concurrently running tasks — far more



(a) Streaming word-count (b) Encoding videos on ExCamera

Fig. 13. For streaming word-count and ExCamera applications, Jiffy performance is comparable to systems with over-provisioned capacity. Details in §6.5.

than the total number of tasks in the entire Snowflake workload. Finally, the controller readily scales to multiple servers by partitioning the set of address hierarchies across them (§4).

Storage overheads. The task-level metadata storage in Jiffy has an overhead of only 64 bytes of fixed metadata per task and 8 bytes per block. For the default 128MB blocks used in Jiffy, the metadata storage overhead is a tiny fraction of the total storage ($< 0.00005 - 0.0001\%$).

6.5 Additional Applications on Jiffy

The Snowflake workload evaluated in §6 shows Jiffy performance a SQL application (MapReduce model, §5.1). We now evaluate Jiffy for two additional serverless applications, which make use of other models.

Streaming word-count. This workload comprises 50 partition tasks that split input sentences randomly sampled from the Wikipedia dataset [76] into words and partition them based on their string hashes, and 50 count tasks that collect words within a partition and compute their counts. The job employs queues as data channels (Dataflow model) and stores word counts in a KV-store (Piccolo model). We compare Jiffy performance with Elasticache (both hosted on 5 m4.16xlarge EC2 instances), since both support queue and KV-store models. Fig. 13(a) shows the CDF of end-to-end latency for 64-sentence batches. Despite its benefits (§6.3), Jiffy can match the performance of an over-provisioned Elasticache cluster.

Video encoding. ExCamera [12] is a video processing framework that facilitates fine-grained parallelism for video encoding on AWS Lambda. It performs encoding using serverless tasks that exchanged state via a dedicated rendezvous server that forwards messages between them. We compare the rendezvous server approach with state exchange via Jiffy queues in ExCamera, both being hosted on a single m4.16xlarge instance. Fig. 13(b) shows ExCamera task latencies for uncompressed 4k raw frames from [77]. Compared to ExCamera, Jiffy reduces task wait times (lighter shade) by 10-20% via queue notifications.

6.6 Sensitivity Analysis

We now analyze Jiffy's sensitivity to various system parameters, including block size (§3.1), lease duration (§3.2) and thresholds for data repartitioning (§3.3). We use files as our underlying data structure, and use the Snowflake workload from Fig. 1. These results can be contrasted directly with Fig. 11(a) (center), which corresponds to our default system parameters (128MB blocks, 1s lease duration and 95% block usage as repartition threshold). For each parameter that we vary, the others remain fixed at their default values.

Block size. The block size in Jiffy exposes a tradeoff between the amount of metadata that needs to be stored at the control plane and resource utilization (§3.1). We confirm this in Fig. 14(a), where increasing the block size from 32MB to 512MB increases the disparity between allocated and used capacity, and therefore decreases the resource utilization. The default block size in Jiffy is set to 128MB since: (1) it achieves high utilization with low metadata overhead (a few megabytes for terabytes of application data), and (2) it is the default block size in most analytics platforms.

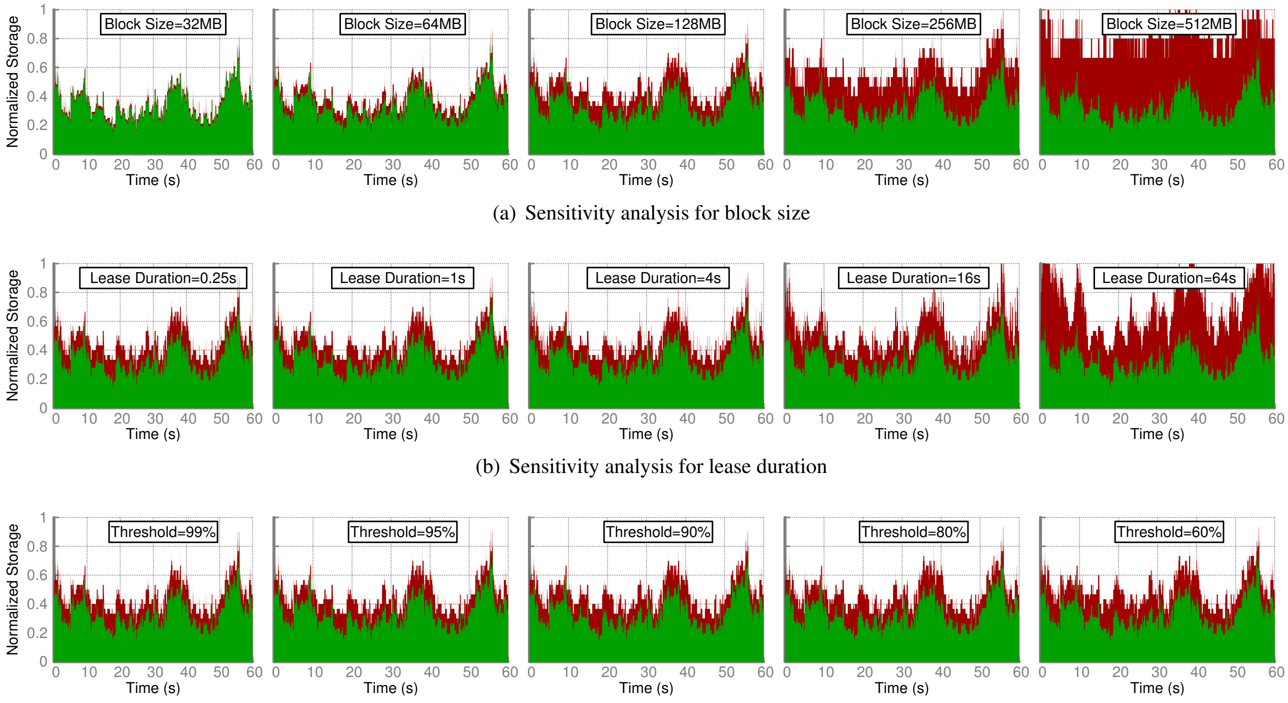
Lease duration. As shown in Fig. 14(b), lease duration in Jiffy affects resource utilization over time. As we increase lease durations from 0.25 seconds to 64 seconds, resource utilization decreases since Jiffy does not reclaim (potentially unused) resource resources from jobs until their leases expire. At the same time, if we keep lease duration too low, applications would renew leases too often, resulting in higher traffic to the controller. We find a lease duration of 1s to be a sweet spot, ensuring high resource utilization, while ensuring the traffic to the controller for even thousands of concurrent applications is only a few thousand requests per second — well within Jiffy controller's limits on a single CPU core.

Repartition threshold. Finally, Fig. 14(c) shows the impact of (high) repartition threshold on resource utilization. As expected, lower repartition threshold leads to lower utilization, since it triggers premature allocation of new blocks to most files in the workload. Note that since the size of the block (128MB) is much smaller than the amount of data written to each file in the workload (often several gigabytes), this overhead is relatively small when compared to effect of other parameters. Since a large value of high repartitioning threshold results in more frequent block allocation requests to the controller, our default value of 95% provides a sweet spot between resource utilization and number of allocation requests.

7 Related Work

We discussed intermediate storage systems for serverless analytics in §1 and §6.2; we now discuss other related systems.

Pocket [8] has shown how existing designs for in-memory key-value stores [49–52, 78–82], distributed [83–86] and disaggregated memory systems [31, 32, 87, 88], and storage systems with flexible interfaces [89–96] can be extended to facilitate three key goals for intermediate data storage in serverless analytics: low-latency/high-throughput, storage resource sharing and resource elasticity. Jiffy strives for complementary goals of resolving specific challenges arising from



(a) Sensitivity analysis for block size

(b) Sensitivity analysis for lease duration

(c) Sensitivity analysis for (high) repartition threshold

Fig. 14. Jiffy sensitivity analysis for (a) block size (b) lease duration and (c) repartition threshold for the file data structure. Green area corresponds to used capacity, while red area corresponds to allocated capacity under Jiffy. See §6.6 for details.

adapting virtual memory-based allocation to serverless environments (§3) to achieve task-level elasticity, isolation and lifetime management. However, Jiffy is flexible enough to be implemented atop most such systems to achieve these goals.

Our evaluation employs publicly released datasets from Snowflake's production clusters [20]. Snowflake itself neither performs task-level resource allocation, nor does it provide isolation across tasks. In fact, Snowflake's ephemeral storage is not shared across tenants, or even tasks running on separate compute nodes. Due to the above reasons, Snowflake does not need to perform data lifetime management either. In contrast, Jiffy is designed for multi-tenant environments, and provides lifetime management for serverless analytics.

Other recent storage systems have also explored fine-grained resource sharing. Pisces [97] provides per-tenant *performance* isolation in a multi-tenant cloud storage system, but does not share *storage capacity* across tenants. Memshare [98] facilitates memory sharing across multiple tenants, but operates under a KV cache setting, i.e., under high contention, it evicts KV pairs that contribute less to overall system hit-rate. In contrast, Jiffy focuses on more general data models that support fine-grained memory elasticity via efficient data repartitioning, and allows applications control over the data that resides in memory via leasing.

8 Conclusion

We have presented Jiffy, a far-memory system for storing ephemeral state that matches the instantaneous capacity demands for stateful serverless analytics jobs. Jiffy resolves unique challenges introduced by serverless environments using a combination of hierarchical addressing, efficient data lifetime management via leasing, and flexible data repartitioning. Jiffy supports rich data models that enable several powerful distributed programming frameworks on serverless platforms. Our evaluation shows that Jiffy improves job execution time by $1.6-2.5\times$ for production workloads.

Acknowledgements

We would like to thank our shepherd Flavio Junqueira and anonymous EuroSys reviewers for their valuable comments and insightful feedback. We are also grateful to Midhul Vuppalapati for helping us parse and analyze the Snowflake traces. This work is supported in part by NSF Awards 2047220, 1704742, 1730628, 1763810, 1838733 and their REU supplements, as well as a Sloan Fellowship, a NetApp Faculty Fellowship and gifts from Amazon Web Services, Ant Group, Ericsson, Futurewei, Google, Intel, Meta, Microsoft, Scotiabank, and VMware.

References

[1] AWS Lamda. https://aws.amazon.com/lambda/.

[2] Azure Functions. https://azure.microsoft.com/en-us/services/functions.

[3] Google Cloud Functions. https://cloud.google.com/functions.

[4] State of the Serverless Community Survey Results. https://serverless.com/blog/state-of-serverless-community.

[5] 2018 Serverless Community Survey: huge growth in serverless usage. https://bit.ly/2Mu5TCR.

[6] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. In *SIGMOD*, 2020.

[7] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[8] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[9] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. In *CLOUD*, 2018.

[10] Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda.

[11] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *SoCC*, 2019.

[12] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI*, 2017.

[13] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *SoCC*, 2017.

[14] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[15] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *ATC*, 2019.

[16] Amazon. Amazon Athena. https://aws.amazon.com/athena.

[17] Amazon. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless.

[18] Azure. Azure SQL Data Warehouse. https://azure.microsoft.com/en-us/services/sql-data-warehouse.

[19] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[20] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[21] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *NSDI*, 2021.

[22] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *SOSP*, 2021.

[23] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[24] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[25] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. In *ASPLOS*, 2014.

[26] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. Memory disaggregation: Research problems and opportunities. In *ICDCS*, 2019.

[27] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[28] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[29] Ahmad Samih, Ren Wang, Christian Maciocco, Mazen Kharbutli, and Yan Solihin. *Collaborative Memories in Clusters: Opportunities and Challenges*. 2014.

[30] Krste Asanović. Firebox: A hardware building block for 2020 warehouse-scale computers. 2014.

[31] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[32] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, 2017.

[33] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[34] Amanda Carbonari and Ivan Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets*, 2017.

[35] High Throughput Computing Data Center Architecture. http://www.huawei.com/ilink/en/download/HW_349607.

[36] The Machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/.

[37] Intel Rack Scale Design: Just what is it? https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it/.

[38] Facebook's Disaggregated Racks Strategy Provides an Early Glimpse into Next Gen Cloud Computing Data Center Infrastructures. https://dcig.com/2015/01/facebooks-disaggregated-racks-strategy-provides-early-glimpse-next-gen-cloud-computing.html.

[39] Rack-scale Computing. https://www.microsoft.com/en-us/research/project/rack-scale-computing/.

[40] In Bid for Major Carriers and Service Providers, Dell EMC Rack Scale Infrastructure Offers 'Hyperscale Principles'. https://www.enterpriseai.news/2017/09/12/bid-major-carriers-service-providers-dell-emc-rack-scale-infrastructure-offers-hyperscale-principles/.

[41] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In *OSDI*, 2018.

[42] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP$\approx$RDMA: CPU-efficient remote storage access with i10. In *NSDI*, 2020.

[43] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *OSDI)*, 2021.

[44] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP*, 1988.

[45] Brett Fleisch and Gerald Popek. *Mirage: A coherent distributed shared memory design*. 1989.

[46] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *TOCS*, 1988.

[47] Partha Dasgupta, Richard J LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *Computer*, 1991.

[48] John B Carter, Dilip Khandekar, and Linus Kamb. Distributed shared memory: Where we are and where we should be headed. In *HotOS*, 1995.

[49] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *NSDI*, 2014.

[50] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS OSR*, 2010.

[51] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[52] Redis. http://www.redis.io.

[53] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 2008.

[54] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS OSR*, 2007.

[55] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: continuous reliable distributed processing of big data streams. In *NSDI*, 2016.

[56] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, 2010.

[57] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[58] TPC-DS. http://www.tpc.org/tpcds/.

[59] Hadoop Distributed File System. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[60] Paul R. Wilson. Uniprocessor garbage collection techniques. In *IWMM*, 1992.

[61] David I Bevan. Distributed garbage collection using reference counting. In *PARLE*, 1987.

[62] K G Cassidy. Feasibility of automatic storage reclamation with concurrent program execution in a lisp environment. master's thesis. 1985.

[63] Cary Gray and David Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*. 1989.

[64] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *OSDI*, 2006.

[65] R. Droms. RFC 2131: Dynamic Host Configuration Protocol. https://www.ietf.org/rfc/rfc2131.txt, 1997.

[66] Amazon ElastiCache. https://aws.amazon.com/elasticache.

[67] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[68] Amazon Simple Notification Service (SNS). https://aws.amazon.com/sns.

[69] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *ATC*, 2010.

[70] Apache Thrift. https://thrift.apache.org/.

[71] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[72] Apache Hadoop. https://hadoop.apache.org/.

[73] libcuckoo. https://github.com/efficient/libcuckoo.

[74] Amazon EC2. https://aws.amazon.com/ec2/.

[75] Amazon S3. https://aws.amazon.com/s3.

[76] Wikipedia Dataset. https://en.wikipedia.org/wiki/Wikipedia:Database_download.

[77] Ton Roosendaal. Sintel. In *ACM SIGGRAPH CAF*, 2011.

[78] MemCached. http://www.memcached.org.

[79] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.

[80] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in anna. *VLDB*, 2019.

[81] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *NSDI*, 2015.

[82] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[83] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *SOSP*, 1991.

[84] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 1989.

[85] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 1991.

[86] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *WTEC*, 1994.

[87] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *ATC*, 2018.

[88] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *SOSP*, 2021.

[89] Postgres: User defined Functions. https://www.postgresql.org/docs/8.0/xfunc.html.

[90] Oracle: User defined Functions. https://docs.oracle.com/cd/B19306_01/server.102/b14200/functions231.htm.

[91] SQL Server: User defined Functions. https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions.

[92] Postgres: Stored Procedures. https://www.postgresql.org/docs/11/sql-createprocedure.html.

[93] Oracle: Stored Procedures. https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.htm.

[94] SQL Server: Stored Procedures. https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-2017.

[95] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[96] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[97] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, 2012.

[98] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *ATC*, 2017.