Research Statement

Seo Jin Park

The goal of my research is to build efficient large-scale parallel systems that can accelerate big data applications by 100–1000× without incurring a hefty cost. My work is motivated by two growing trends: big data and cloud computing. Today, many businesses and web services store staggering quantities of data in the cloud and lease relatively small clusters of instances to run analytics queries, train machine learning models, and more. However, the exponential data growth, combined with the slowdown of Moore's law, makes it challenging (if not impossible) to run such big data processing tasks in real-time. Most applications run big data workloads on timescales of several minutes or hours, and resort to complex, application-specific optimizations to reduce the amount of data processing required for interactive queries. This design pattern hinders developer productivity and restricts the scope of applications that can use big data.

My research aims to enable interactive, cost-effective big data processing through "flash bursts". *Flash bursts enable an application to use a large portion of a shared cluster for short periods of time.* This could allow big data applications to complete significantly faster, with cost comparable to leasing a few instances for a longer period of time. A flash-burst-capable cloud could enable new real-time applications that use big data in ad hoc ways, without relying on query prediction and precomputation; e.g., a security intrusion detection system could analyze large-scale logs from many machines in real-time as threats emerge. It could also improve developer productivity (e.g., a machine learning researcher could train models and iterate on new ideas quickly). Finally, flash bursts could reduce costs, since applications would no longer need to precompute results for all potential queries.

The main challenge to flash bursts is Amdahl's law. Flash bursts launch small tasks per node that run for milliseconds at a time. With such small task granularity, previously negligible overheads (e.g., synchronization, I/O processing, cache misses, etc.) can turn into significant bottlenecks. In addition, with more nodes, the communication overhead for coordinating many nodes grows and limits scaling. My research takes a two-pronged approach to tackle this challenge: restructuring important applications (e.g., data analytics, DNN training) to use flash bursts efficiently [1, 2] and making distributed systems more efficient for flash bursts [3, 4, 5, 6, 7, 8, 9, 10]. My high-level approach has been to: (1) rethink distributed protocols (or application logic) to minimize protocol/algorithm-driven overheads (e.g., extra message delays) and (2) develop highly optimized implementations of them to eliminate even sub-microsecond overheads (e.g., cache misses). Throughout this process, I build systems and use in-depth measurements to deeply understand bottlenecks and uncover opportunities for improvement.

1 Ensuring big data applications can use flash bursts efficiently

The most crucial premise of flash bursts is that applications must be able to utilize thousands of servers in parallel over short periods of time. Even with the perfect system infrastructure (e.g., instant application and data loading, perfect isolation from other workloads, etc), applications may not be able to efficiently harness the computing power of many servers for short periods if their core algorithms have poor parallelism or suffer from high overheads with small task granularity.

To learn whether common applications can benefit from flash bursts or not, I have conducted three case studies: SQL query processing, distributed sorting, and deep neural network training. For these studies, I assumed an ideal cloud infrastructure: data collocated with compute nodes, no concurrent jobs causing interference, etc.

Data analytics. For the first application, I chose sorting and SQL query processing within one or ten milliseconds, which I call **MilliSort** and **MilliQuery** (**NSDI'21**) [1]. Since the time budget of 1 ms is extremely limited (e.g., 1 ms only allows 10000 sequential cache misses or 200 sequential RPCs), I redesigned

both applications from scratch to avoid cache misses and sequential RPCs. In addition, for MilliSort, I invented a new distributed sorting algorithm that uses a hierarchical form of key range partitioning, which is 200× faster than the previous systems [11] (100 ms → 0.5 ms for 280 nodes). Thanks to this careful redesign, MilliSort performs very efficiently even at the extreme 1 ms timescale. MilliSort's throughput per node reaches 63.6% of the idealized upper bound, where the partitioning cost is zero and data is shuffled at full network bandwidth with perfect balance. This is on par with the efficiency of running a state-of-the-art sorting system [12] for 100 seconds.

By measuring the performance of MilliSort and MilliQuery, I gained a basic understanding of the limits of flash bursts. First, the smallest practically possible timescale of flash burst jobs was 0.5 – 1ms. Below that, both applications are better off running at a small scale. Second, the total data that can be processed increases quadratically with the time budget; which is because both the optimal cluster size and the amount of data per node increase about linearly with the time budget. Third, The primary obstacle to scalability is per-message costs, which lead to inefficient data shuffles and high coordination overhead.

Efficient GPU scaling. The next application I chose was efficient strong scaling for deep neural network (DNN) training. Scaling DNN training to hundreds or thousands of GPU nodes poses a significant challenge to scaling efficiency. At large scales, the conventional scaling strategy of increasing global batch size doesn't reduce overall training time to accuracy. For continued improvement on time to accuracy, we must consider "strong scaling" strategies that hold the global batch size constant and allocate smaller batches to each GPU. Unfortunately, small-batch training often underutilizes modern GPUs with many compute cores. Thus, we have had to make an unfortunate choice between high cluster utilization or fast training.

To address this challenge, I built **DeepPool (MLSys'22)** [2], which enables both good cluster throughput (2.2–2.4× over baseline) and fast training. DeepPool incorporates two key ideas. First, I introduced a *"burst parallel training planner"* to dynamically adjust the number of GPUs allocated to each layer, so that layers with less parallelism can use fewer GPUs. This increases overall cluster efficiency because it frees up underutilized GPUs for use by other training tasks. Second, I propose a new collection of GPU multiplexing techniques that allow a background training task to reclaim the GPU cycles unused by the large-scale foreground task. On three popular image classification workloads, DeepPool can achieve a 2.2–2.4× improvement in total cluster throughput over standard data parallelism with a single task (about 75-82% of the maximum possible throughput with large batch training). At the same time, our GPU multiplexing techniques reduce degradation in throughput of foreground jobs caused by collocation with a background job by 20–39%.

2 Making distributed systems more efficient to flash bursts

Flash bursts have many characteristics that pose challenges to systems: more remote operations, high fanouts, and shorter time budgets. For efficient scaling to thousands of nodes, we must minimize overheads in remote procedure calls and suppress tail latency spikes. The second part of my research on flash bursts is improving distributed systems so that they can work more efficiently with granular and bursty jobs.

Replication / consensus overhead. Flash bursts are more susceptible to machine failures since any individual machine failure may stop the entire application. To provide high availability for such applications, systems must rely on replication to mask individual machine failures. However, replication introduces a significant latency overhead because it requires round-trip communication to one or more additional servers. Within a datacenter, replication can easily double the operation latency (1 RTT → 2 RTTs) compared to an unreplicated system.

The reason for 2 RTTs is because most systems perform ordering of operations before replication. A client first sends an operation to a server that orders the operation with respect to other concurrent operations (and usually executes it as well); then that server issues replication requests to other servers, ensuring a consistent ordering among replicas. As a result, the minimum latency for an operation is two round-trip

times (RTTs).

I found that the overhead for replication can be mostly removed if we take advantage of the fact that most operations are commutative, so their order of execution doesn't matter. To implement the idea, I developed **Consistent Unordered Replication Protocol (CURP)** (**NSDI'19**) [3], which supplements a system's existing replication mechanism with a lightweight form of replication without ordering. A client replicates each operation to one or more temporary request storage nodes, which I call *witnesses*, in parallel with sending the request to the primary server; the primary can then execute the operation and return to the client without waiting for normal replication, which happens asynchronously. This allows operations to complete in 1 RTT, as long as all witnessed-but-not-yet-replicated operations are commutative. When applied to the RAMCloud [13] key-value store, CURP reduces write latency by half (only a 1 µs overhead relative to RAMCloud without replication) and increases throughput by 3.8x.

My follow-up work, **ARGUS** (**APNET'19**) [4], moved CURP's temporary backup servers from user-level processes to SmartNICs. This approach brings about two benefits: better tail latency and saving of CPU resources. Using Linux processes for backup servers results in high tail latency due to various software overheads (e.g., networking stack and context switching overhead). Using more backups makes the problem even worse since clients must wait for replication to complete at all backup servers. This undesirable effect is known to be especially problematic in public clouds. ARGUS avoids this issue by taking advantage of SmartNICs' proximity to the wire, minimal software overhead, and line-rate throughput.

Although the idea of exploiting commutativity for 1 RTT replication generally works well within a datacenter, it doesn't work as well in wide-area networking environments. High network delays make it more difficult to exploit commutativity since they increase the window of time during which operations can conflict with each other. Those conflicting operations must take the standard 2-RTT slow path. I mitigated this problem with a technique called **Timestamp-Ordered Queueing (TOQ)** (**NSDI'21**) [5]. With the observation that conflicts occur primarily because different replicas process operations at different times, we modified EPaxos to use synchronized clocks to ensure that all quorum replicas process a given operation at the same time. This reduces conflict rates by as much as 50% without introducing any additional latency.

Consistency / concurrency control overhead. I also worked on minimizing the latency overhead of consistency layers of RPCs. For an RPC, a client sends an RPC request and waits for the return value from the server. If a client doesn't receive a response, it has no idea what happened to its request. The traditional "solution" to this problem is to retry the request after some timeout period. But this may cause re-execution of the RPC request, resulting in consistency issues, such as double increments or overwriting later operations. To prevent RPC re-executions, I designed **Reusable Infrastructure for Linearizability (RIFL)** (**SOSP'15**) [7]. RIFL records the results of completed RPCs durably and returns the saved results without re-executing if an RPC is retried after it has completed. RIFL guarantees safety even in the face of server crashes and system reconfigurations such as data migrations. I designed and implemented RIFL to be scalable (10,000's of servers and 1 million clients) and low-latency (0.5 µs overhead).

I reused RIFL beyond simple RPC retry handling. I extended RIFL to implement a new low-latency distributed ACID transaction mechanism (20 µs commit latency), to prevent duplicate replay from both witnesses and backups in CURP, and to ensure consistency of hierarchical RPCs on multi-layer replicated servers in Nu [10].

Programmability and flexible scaling. Writing a distributed application is a complex job, especially for flash bursts. From my MilliSort experience, I learned that manually managing communication among many servers is very complicated and not flexible enough to adjust scaling dynamically. My recent work **Nu (one-shot revision to NSDI'23)** [10] addresses those difficulties by realizing *logical processes*, a new abstraction that splits the classic UNIX process into many atomic units of state called proclets that can be independently and quickly migrated. With Nu, implementing distributed applications becomes as easy as writing single-process programs. But, thanks to fine-grained proclets, local processes may migrate parts of

their state and the corresponding compute load to other servers seamlessly and quickly (a few microseconds).

Debugging/preventing high tail latency. A flash burst server must rely on fanouts of RPCs to efficiently communicate with multiple other servers. A server using a fanout must wait until the last RPC returns, so keeping the tail latency low is important for the overall performance. **Breakwater** (**OSDI'20**) [8] is an overload control system that prevents request queue build-ups at servers so that a server's tail latency is always under the target SLO. **NanoLog** (**ATC'18**) [9] is a prinf-like event logging system with just 8 to 18 nanosecond overhead per log. The low-overhead logging enables tracing each RPC's lifetime in detail, which helps to debug performance.

3 Future work

For the next five years, I will continue working on bringing flash bursts to reality. On the application side, I will collaborate with potential customers (both industry and academics using big data). On the system side, I will work more closely with cloud providers. As the immediate next steps, there are a few specific directions I plan to pursue.

Straggler avoiding scheduler and smart replication. A single straggler node can hurt the performance of an entire flash burst application. Stragglers occur mostly due to resource contention, so a cluster scheduler should ensure that all assigned nodes will have enough resources available. A flash burst scheduler should solve the scheduling of untraditional jobs: a job needs many nodes together, its performance depends on the straggler node, and the scheduler may reduce scale if necessary. In addition, flash burst applications can easily be bottlenecked by initial data loading. If all nodes load data from a single storage server, the egress bandwidth of the storage server will become a bottleneck. To avoid that, an application's data must be properly scattered and replicated.

Energy-efficient scaling. Datacenter energy consumption is becoming a more serious issue, and I believe that flash bursts will open up new opportunities to save energy. For example, a big data workload may run on slow-but-energy-efficient hardware and compensate for the slowdown with scaling to a larger cluster. Also, latency-critical applications (e.g., web services) have had to precompute for future big data queries which may or may not be used later. Flash bursts will allow transforming preprocessing for all potential queries into on-demand flash bursts, providing an opportunity for saving energy and other resources.

Offloading to programmable networking hardware. In the last few years, we have started putting more functionality in network devices, and such networking hardware offloading is becoming more important to keep up with rapidly improving network speeds. I am particularly interested in programmable NICs and switches. I previously showed programmable NICs can be used for fast replication by leveraging NICs' proximity to network wires. In general, I am looking forward to seeing the new networking hardware, and I plan to investigate how to best use them for tail-latency-sensitive workloads like flash bursts.

Performance debugging tools. The last direction is building tools for debugging the performance of flash bursts, especially for diagnosing tail latencies. Flash burst applications often use large fan-out/fan-in communication, so keeping tail latency low is critical for the application performance. However, it's difficult to debug tail latency on large-scale systems, especially on production systems due to the long deployment cycle. Even for research systems like Shenango or RAMCloud, it often takes several days of engineering just to pinpoint the source of tail latency. With the new monitor/diagnosis tool, I will automate the tiresome process of instrumentation, deployment, and performance measurement, so developers will see the source of tail latency spikes and relevant information as the latency spike is detected during a production run.

In conclusion, there are many interesting challenges on the way to bringing flash bursts to clouds. By collaborating with the research community and cloud providers, whose interest in serverless computing is growing, I will make flash bursts practical and widespread within five to ten years.

References

[1] Y. Li, S. J. Park, and J. Ousterhout, "MilliSort and MilliQuery: Large-scale data-intensive computing in milliseconds," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 593–611, USENIX Association, Apr. 2021.

[2] S. J. Park, J. Fried, S. Kim, M. Alizadeh, and A. Belay, "Efficient strong scaling through burst parallel training," in *5th Conference on Machine Learning and Systems (MLSys'22)*, To appear.

[3] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 47–64, USENIX Association, Feb. 2019.

[4] S. Choi, S. J. Park, M. Shahbaz, B. Prabhakar, and M. Rosenblum, "Toward scalable replication systems with predictable tails using programmable data planes," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pp. 78–84, 2019.

[5] S. Tollman, S. J. Park, and J. Ousterhout, "EPaxos revisited," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 613–632, USENIX Association, Apr. 2021.

[6] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, (Renton, WA), USENIX Association, Apr. 2022.

[7] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, "Implementing linearizability at large scale and low latency," in *Proc. 25th ACM Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), ACM, 2015.

[8] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload control for μs-scale RPCs with breakwater," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 299–314, 2020.

[9] S. Yang, S. J. Park, and J. Ousterhout, "NanoLog: a nanosecond scale logging system," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 335–350, 2018.

[10] Z. Ruan, S. J. Park, M. K. Aguilera, A. Belay, and M. Schwarzkopf, "Nu: Achieving microsecond-scale resource fungibility with logical processes," 2021.

[11] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, "CloudRAMSort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 841–850, ACM, 2012.

[12] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub, "Tencent sort."

[13] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Transactions on Computer Systems*, vol. 33, pp. 7:1–7:55, Aug. 2015.