Language Runtime Snapshots in Junction for Fast Serverless Start

Baltasar Dinis Benjamin Holmes

Abstract

Junction, an experimental library operating system, has a lot of potential as a serverless substrate. It inherits Caladan's [4] *scalability* and hability to *handle bursty workflows*. By using kernel-bypass techniques Junction can transparently speed up applications. Additionally, by using Linux container primitives (`seccomp` [3], `chroot` [7] and `cgroups` [6]) and relying only on 12 system calls it can provide *strong isolation*. Finally, by exposing the same system call interface, it is *compatible* with pre-existing applications, not requiring recompilation or code changes.

However, Junction still suffers from significantly high start up times (ranging from 10*ms* to 175*ms* for even simple programs, depending on the language runtime). This lack of fast starts is a key missing piece for Junction to realize its full potential as serverless substrate. We make the key observation that a significant portion of the startup time is in application code and that by adding a snapshot/restore feature to Junction we would be able to skip that overhead altogether. We have implemented this feature in Junction and show that it indeed reduces startup times up to 90%.

1 Background and Motivation

1.1 Requirements for a Serverless Substrate

Serverless computing is a novel paradigm that arose from experience of more traditional, VM-based, public cloud offerings. From a provider perspective, most resources in the datancenter are *stranded* (i.e., allocated but not used). Alibaba [5] reports that CPU usage is higher than 60% under 3% of the time (Figure 1). This is wasteful from both an economic and environmental perspective: power consumption from an idel machine is only 50% that of a fully utilized machine [1] (Figure 2).

The underlying reason for this underutilization is the difficulty cloud clients have in correctly provision resources. Due to varying loads, sometimes extremely bursty ones, clients must *over-provision* resources for peak load, leaving them stranded when that load is not met.

Serverless aims to solve these problems by providing a different interface for the cloud: *functions*. A client partitions their system/application into a set of functions. The number of concurrent invocations of these functions scale with load, eliminating stranded resources. To service these

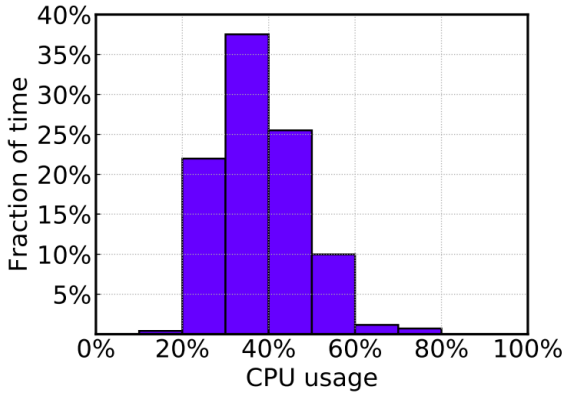

Figure 1: CPU utilization as a fraction of time in Alibaba's datacenter [5].

workloads efficiently, a servereless substratre must meet strict requirements, namely:

1. **Scalability:** It must be possible to handle numerous concurrent invocations.
2. **Handling Burstiness:** It must be possible to launch a lot of concurrent invocations.
3. **Isolation:** Function invocations must be isolated, both from a security perspective, as well as from a performance perspective (i.e., the performance impact of the overall load the datacenter has on the function should be minimized).
4. **Fast Starts:** Functions tipically run for a very short period of time. As such, the start up time becomes a significant factor in overall function latency. Starting functions fast becomes a necessity.

1.2 Kernel-Bypass for Serverless

Serverless computing is a strong candidate to benefit from kernel-bypass systems. Kernel-bypass systems allow applications to avoid kernel crossings and save significantly on both latency and throughput by interacting directly with the hardware. This manifests in savings across the board; startup time can be reduced significantly by avoiding system calls and handling I/O in userspace allows users' functions to experience significantly less latency per-request.

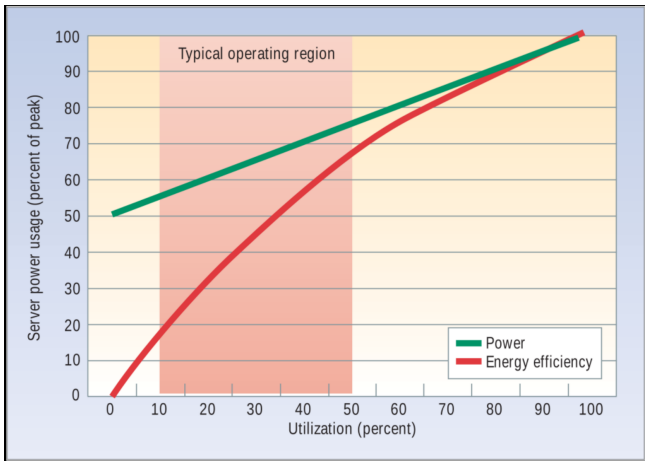

Figure 2: Under utilization is energy-wasteful [1].

Existing systems rely on allocating busy spinning cores per-instance and large per-core packet buffers to avoid drops. This clashes with the conditions necessary for serverless because it prevents scalability. Similarly, existing systems lack strong isolation and must be run as root or simply are not targetted at a multi-tenant cloud. Finally, existing systems require applications to link with a custom runtime to gain access to kernel-bypass features.

Junction addresses the problems with existing kernel-bypass systems to meet the requirements for serverless:

1. **Scalability** is achieved by leveraging modern NIC features to configure the NIC to share a queue of recieve buffers across multiple cores and relying on a centralized scheduler core.

2. **Handling Burstiness** is achieved by building on top of Caladan [4] to adapt to load on the microsecond timescale.

3. **Isolation** is achieved by reimplementing the majority of the Linux system call interface in userspace and only exposing a dozen syscalls to the host kernel.

We now discuss Caladan and Junction in greater detail.

1.3 Caladan

Caladan is a CPU scheduler that achieves *fast core allocation* (on the order of tens of microseconds) to achieve performance isolation. Caladan relies on a scheduler core to make centralized scheduling decisions and reallocate cores at a microsecond scale, making it possible to handle burstiness efficiently. Caladan levarges busy polling of shared memory regions to collect control signals and enable asynchronous core scheduling. Caladan also enables work stealing across cores and offloads scheduling work to the schedulee rather than the scheduler to maximize resource utilization.

By leveraging Caladan, Junction can handle burstiness efficiently and offer strong performance isolation.

1.4 Junction

Junction is a library operating system (libOS) built for containing applications in a multi-tenant cloud setting. It leverages kernel bypass hardware like NIC queues and new CPU features to emulate the Linux kernel system call interface in userspace. This reduces the number of system calls made to the host kernel to about a dozen that are needed mostly for multiplexing resources like CPU and memory. Since most of the OS functionality is handled in userspace, Junction provides strong security isolation and high performance by significantly reducing kernel crossings. Moreover Junction offers a familiar Linux API, allowing for unmodified binaries to run.

Of the requirements we identified for an ideal serverless substrate, the one that Junction does not address is fast starts. The initialization time of serverless functions remains an important metric because startup time is often on the same order of magnitude as a function's runtime, so reducing startup time is key to minimizing latency and maximizing the time spent doing useful work in datacenters. Startup time includes both the initialization time of the container system and the time it takes to initialize the environment for a function to run. Because serverless function are typically written in high level languages, a large portion of their startup time is runtime initialization and library loading.

Junction initialization workflow. When a new Junction instance is launched, it first `chroot()`s itself to restrict host file system access, and enables a `seccomp` filter that restricts all but the dozen syscalls it does not emulate. Junction also runs on top of the Caladan scheduler to provide interference-aware, low latency scheduling for unmodified Linux applications. During startup, shared memory channels are initialized between the new Junction instance and a global Caladan instance running alongside the host kernel. Finally, Junction's built in ELF loader loads and executes a program. While Junction's startup is ligtweigt, it does not avoid the cost of environment initialization. To unlock Junction's true potential as the isolation solution for a blazingly fast serverless system, it should support *warm start* to launch an application from a pre-initialized environment.

2 Design

We design a warm start solution for Junction with the following elements: *snapshotting* and *restoring*.

Snapshotting. During snapshot creation, Junction will initialize the environment needed to run a function, likely a

language runtime. Before any function specific code would execute, Junction will pause the running process, checkpoint its state, and save it to a file. We call this the *base snapshot* which will be used to quickly run a function without redundant setup. For our prototype, we propose the addition of a new call into the Junction kernel, `screate()` to handle saving the state of the calling process to a file. A serverless provider may support a large number of runtimes across languages and versions, so the ability to save a base snapshot to disk is important because the alternative is to keep one paused Junction instance in memory for each supported language runtime, tying up a significant amount of memory.

Restoring. To load a snapshot, we will have to modify Junction's intialization workflow to set up all the required connections with Caladan that existed at snapshot creation time, so that its environment is fully constructed after the snapshot is loaded. An added benefit of Junction's design is that instances can share mappings in the Linux page cache by virtue of each instance being encapsulated by a normal Linux process. With this in mind, our design for *snapshot loading* allows Junction instances to share read only mappings of the same base snapshot backed by the page cache, in the case of multiple invocations that share a runtime. This will further reduce the I/O time to launch from a snapshot.

2.1 Snapshot Format: ELF + Metadata

We have identified that using ELF as the underlying snapshot format will be extremely benefitial: we can copy the processes memory directly into ELF segments. All additional metadata is serialized under a custom format on a separate file. This not only simplifies creating snapshots, by reducing the ammount of data we need to serialize, but also means loading will become simpler: we need only parse the metadata and map the remaining segments.

2.2 Snapshotting API

We have identified three possible ways to trigger a snapshot in Junction.

Signal-based. After receiving a custom signal, the process would start snapshotting itself. This is attractive because it would allow binaries not to be modified. Unfortunately, Junction's signal support is still limited.

Runtime-triggered. Junction itself could hardcode a timeout after which a snapshot would be triggered. Although an instersting possibility, we consider this future work.

Custom system call. The system call interface is an attractive place to introduce a snapshot trigger. Not only does

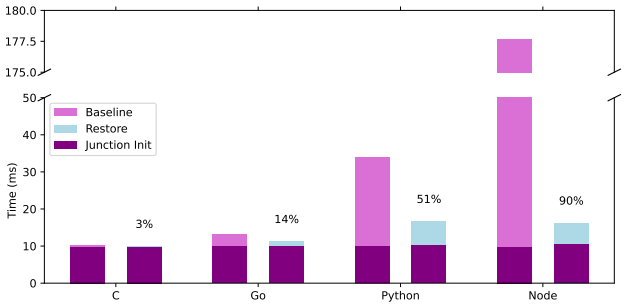

Figure 3: Comparison between startup time with and without snapshotting. There are gains across all languages. Gains are more significant when the startup costs arer higher.

Junction already have support for saving registers when entering a system call, but it allows for flexibility in testing our prototype. For this reason we chose to implement a new `snapshot` system call, which offers a `fork()` inspired interface, by returning different values depending on whether it returns from sucessful snaphsot or a restore.

2.3 Restoring

To restore from a snapshot, Junction first creates a harness for the process from the serialized metadata. Then, it can load the ELF file with all the memory mappings. Subsequently, it can restore the registers and return from the system call.

3 Implementation

We implemented our prototype in around 2500 lines of C++ code, within the Junction codebase. We added support for virtual memory areas to Junction, added logic to snapshot a single threaded process and modified the loading sequence to take an optional snapshot and restore from it.

4 Evaluation

We aim to evaluate our prototype by answering the following research questions:

1. Can snapshotting improve startup times across different programming languages? (§4.1)

2. Can snapshotting improve startup time when using libraries? (§4.2)

We ran our experiments on a Intel Xeon Silver 4210R Server CPU running Ubuntu Server 23.04 using Linux (version 6.2.0) as its host kernel.

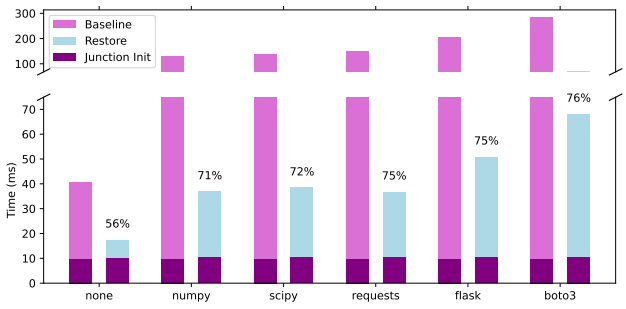

Figure 4: Comparison between startup time with and without snapshotting for different python libraries. `numpy`, `scipy`, `requests` and `flask` refer to common Python libraries for numerical computation and HTTP requests; `boto3` is AWS's SDK and `none` refers to a Python program that imports no libraries.

4.1 Restore performance across Languages

We implemented a simple Hello World style program in C, Go, Python and Node.js. We snapshotted these programs and compared the startup time from the original program with the startup time when restoring from a snapshot. As seen in Figure 3, the speedups range from 3% (for C) to 90% (for Node). We also observe that Go, which bundles the runtime with the compiled artifact, has a smaller startup cost (originally and after restoring) when compared to Python and Node, which run in an interpreter that dinamically loads the program (and thus can benefit more from snapshotting).

4.2 Restore performance when loading libraries

Hello World programs show the minimum possible startup time. However, realistic applications load libraries when initializing, which increases startup time. To characterize the speedup that snapshot/restore can achieve in this setting, we ran an experiment where we measured the speedup of restoring snapshots of common Python libraries. As Figure 4 shows, the startup time is consistently reduced by around 75%.

4.3 Discussion

Junction Initialization. Junction adds a 10*ms* startup time to all example programs. In the future, we wish to understand this bottleneck better and find ways to move that cost to before application instantiation time.

Absolute Latency. Looking past Junction's initialization time, both Python and Node have a baseline $\sim 8ms$ overhead in the hello world programs. Moreover, Python's `boto3` library has a staggering $\sim 60ms$ initialization cost. We have yet to conduct a thorough investigation into the source of restore overhead, but we believe that a large portion of the overhead comes from minor page faults when the restored application starts running. For example, restoring Python with no imports incurs 1946 minor faults while restoring `boto3` incurs 7912.

5 Future Work

This work is ongoing. In the future, we aim to work towards enabling a true sub-millisecond start time for functions. We believe the following are interesting problems that can help us achieve this goal.

Multithreading Support. This requires issueing a signal to all threads to stop and trap into the Junction kernel to retrieve their register state so they can be included in the snapshot. This may require reevaluating our mechanism to trigger snapshots.

Signal-based trigger. Allow the signal to snapshot a process to come from Junction instead of requiring programs to be modified to use our special syscall.

Leverage hardware compression. This allows us to reduce the memory pressure of restoring from large snapshots, making restoring faster.

Add support for differential snapshotting. Inspired by SEUSS [2], we aim to be able to support differential snapshots. Then, by taking a tree-based view of the snapshots and the diffs, we can circumvent the provisioned concurrency problem [8] by keeping partially-specialized warm instances that require only a last level differential snapshot application to start running.

6 Conclusion

In this project, we have implemented a basic snapshot/restore feature in Junction, a kernel-bypass libOS for serverless. We have observed that restoring from a snapshot ran reduce the startup time up to 90% of the original time. In the future, we aim to add support for multithreading, differential snapshots and hardware compression accelaration, in order to achieve practical sub-millisecond startup times.

References

[1] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[2] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip redundant paths to make serverless fast. In *Proceedings of*

the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Jake Edge. A `seccomp` overview. https://lwn.net/Articles/656307/, 2015.

[4] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[5] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Paul Menage. Control groups. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html, 2004.

[7] Michael Kerrisk *et al*. `chroot(2)` man page. https://man7.org/linux/man-pages/man2/chroot.2.html, 2023.

[8] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.