Impact of Microarchitectural State Reuse on Serverless Functions

Truls Asheim
Norwegian University of Science and Technology
Norway

Tanvir Ahmed Khan
University of Michigan
USA

Baris Kasicki
University of Michigan and Google
USA

Rakesh Kumar
Norwegian University of Science and Technology
Norway

ABSTRACT

Serverless computing has seen rapid growth in the past few years due to its seamless scalability and zero resource provisioning overhead for developers. In serverless, applications are composed of a set of very short-running functions which are invoked in response to events such as HTTP requests. For better resource utilization, cloud providers interleave the execution of thousands of serverless functions on a single server.

Recent work argues that this interleaved execution and short run-times cause the serverless functions to perform poorly on modern processors. This is because interleaved execution thrashes the microarchitectural state of a function, thus forcing its subsequent execution to start from a cold state. Further, due to their short-running nature, serverless functions are unable to amortize the warm-up latency of microarchitectural structures, meaning that most the function execution happen from cold state.

In this work, we analyze a function's performance sensitivity to microarchitectural state thrashing induced by interleaved execution. Unlike prior work, our analysis reveals that not all functions experience performance degradation because of microarchitectural state thrashing. The two dominating factors that dictate the impact of thrashing on function performance are function execution time and code footprint. For example, we observe that only the functions with short execution times (< 1 ms) show performance degradation due to thrashing and that this degradation is exacerbated for functions with large code footprints.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; *Cloud computing*; Client-server architectures; • **Networks** → *Cloud computing*; • **General and reference** → **Measurement**.

KEYWORDS

Serverless, FaaS, Microarchitecture, Top-Down, Measurement

WoSC'22, November 2022, Québec City, Québec, Canada
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Truls Asheim, Tanvir Ahmed Khan, Baris Kasicki, and Rakesh Kumar. 2022. Impact of Microarchitectural State Reuse on Serverless Functions. In *Proceedings of 8th International Workshop on Serverless Computing (WoSC'22).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Serverless computing (or Function-as-a-Service (FaaS)) is emerging as a prominent cloud computing model. Applications developed for the serverless model are structured using one or more stateless *functions* that are invoked in response to specific external events such as an HTTP request or a timer trigger. The underlying design philosophy of serverless applications is descendant from microservices. As such, serverless application design heavily emphasizes modularity to enable compositionality and reusability of functions. This means that the execution time of most serverless functions is very short, often as low as a few milliseconds. However, unlike microservice applications, the resources needed for executing a serverless function are transiently allocated. This enables significant cost savings as the user only needs to pay for hosting resources when they are used.

Though function execution times are very short, the majority of functions are invoked very infrequently, often leaving a few seconds or minutes between two consecutive invocations. Thus, to improve resource utilization, cloud providers are forced to co-schedule thousands of functions on each server. However, the downside of such high degree of interleaving is the long startup delay of booting new function instances because a server can keep only a certain number of recently invoked function instances in warmed-up state [9, 14, 8]. Consequently, prior research has aimed at reducing this startup delay to improve the performance of serverless functions [13, 4]. The key idea is to quickly load an execution-ready image of the function into the main memory of the system.

Another downside of function interleaving, as reported by recent work [11, 10], is that interleaved execution thrashes the microarchitectural states of functions. This means that when a function is invoked it does not find any (or much) of its microarchitectural state from its last execution in the microarchitectural structures such as caches, branch predictors, etc. This is because the interleaved functions evict this state as they bring their own microarchitectural state in these structures. Further, prior work [11] also reports that the short execution time of serverless functions prevent them from amortizing the warm-up latency of microarchitectural structures. Consequently, the majority of function executions happen with cold microarchitectural state. As a result, serverless functions show poor performance.

Table 1: The functions used for characterization.

Name	Language	Description
autocomplete	NodeJS	Returns a list of autocomplete candidates.
sentiment_analysis	Python	Identifies the sentiment of a text.
deltablue	Python	Pure-python compute benchmark.
markdown2html	Python	Converts markdown to HTML. Relies heavily on the sha256 implementation in the OpenSSL library.
json_dumps	Python	Serializes a Python dict as JSON
img-resize	NodeJS, libraries	Produces resized versions of images. All of the heavy-lifting in this function is done by native image libraries such as libpng and libjpeg
ocr-img	NodeJS, C++	Invokes Tessaract to OCR an image
dynamichtml	NodeJS	Generates HTML from a template
fib_js_NNN	NodeJS	calculates the NNN'th fibonacci number and returns it.
fib_py_NNN	Python	Same as fib_js but implemented in Python.
footprint_NN	Python, C	Synthetic function that claims a code footprint of NN KB.

This work analyzes the factors that make performance of serverless functions sensitive to interleaved execution induced microarchitectural state thrashing. We analyze both real-world serverless functions as well as synthetic functions with a wide range of execution times (from 0.25 ms to 1.1 s) and different implementation languages. Synthetic functions give us better control over function properties such as execution time, code and data footprints etc. Our results reveal that not all functions show performance degradation due to interleaving induced state thrashing; rather it depends on function properties. Our studies further identify function execution time and code footprint to be the two dominating factors that dictate the impact of thrashing on function performance. The execution time is a particularly interesting factor because real-world deployments report high variability in the execution time of different functions. For example, a study [12] reported that 50% of functions on the Azure cloud completed in less than 1s. Another study [3] found that 50% of functions deployed on AWS Lambda in 2020 completed in 60ms or less. However, the same study noted a decreasing trend in function execution time as 50% of functions completed within 130ms in 2019.

In our study, we find that only very short running functions (median runtime < 1 ms) are adversely affected by being executed on a cold microarchitectural state and that this trend is exacerbated proportionally with increasing function instruction working sizes. For functions with longer execution times (> 50 ms), we find that the performance deterioration caused by interleaved execution is small. This suggests that the microarchitectural state warm-up latency is amortized and the most of function execution happens from warmed-up state.

2 METHODOLOGY

2.1 Experimental setup

We perform our experiments on a server with a 4-core 3.8GHz Intel Xeon E3-1275 v6 (Kaby Lake) processor. The processor has 8MB of shared LLC, 256KB private L2, and 32KB each of private L1-I and L1-D cache and has 64GB of DRAM, SSD drives and both the host system and the application containers run Fedora Linux 36. SMT and Turbo Boost were disabled during the experiments.

Each of the target functions are run inside a Podman container which is pinned to a single processor core. The functions are wrapped by a gRPC server that invokes the function on request. Since the process executing the functions runs as a daemon, as opposed to running as multiple processes, the microarchitectural state is shared across contiguous invocations of the same function. The functions are invoked by a client, implemented in Go, that issues gRPC requests. Each gRPC request only triggers a single invocation and thus, where required by the experiment, the client is configured to repeatedly issue requests for a set amount of time. The client is pinned to a different processor core than the functions. Before statistics collection starts, the functions are warmed by repeated invocations for 30 seconds. This is done to ensure that caches in the execution path are warmed up and that JIT compiled functions had time to reach a fixpoint. Then, the experiments are run for 300 seconds with data collection. The exception to this is the instruction working set estimation (Section 3.3) which is run only for 30 seconds due to the large amount of data collected. All of the invocations of a single benchmark use the same input data.

To simulate how a function is affected by interleaved execution, we invoke a *thrasher* process between subsequent invocations of a function. The container hosting the thrasher process is pinned to the same processor core as the function and is invoked by the client through a gRPC request after every invocation of the function. Upon invocation, the thrasher performs two operations. First, it fills the BTB and branch predictor with garbage using a process [3] that executes a long series of conditional and unconditional jumps. Then, to clear caches, it invokes the WBINVD x86 instruction that writes back and invalidates the entire cache hierarchy (L1 to LLC).

To avoid unintentionally collecting data from the thrasher process we configure `perf` to filter events not belonging to the `cgroup` of the container running the function.

The evaluated functions are executed in two different configurations. In both cases, function executions happen as fast as possible depending on the function. To distinguish these, we use a consistent naming scheme when presenting our results where the experiment configuration is indicated by a suffix added to the benchmark name as follows:

Back-to-back (*benchmark_name* suffix *-none*) Each benchmark is repeatedly invoked by back-to-back requests.

Thrashing (*benchmark_name* suffix *-thrashing*) The thrasher (as described above) is invoked after each invocation of the function.

2.2 Function description

To perform our experiments, we use a mix of representative and synthetic functions written in Python and NodeJS, the two most popular application runtimes for serverless applications [3]. A description of the functions and their implementation is provided in Table 1. The functions used are mainly derived from the FaaSProfiler framework [11] but we use a custom setup for invoking the functions. As mentioned, we also introduce two synthetic functions to the suite in addition to the representative functions. One, *fib_js* and *fib_py* calculates the Nth Fibonacci number and the other, *footprint*, hogs a configurable instruction working set while it is running. *Footprint* achieves this by executing a sequence of jumps to randomized locations in its instruction working set. The Fibonacci calculation function is implemented in both Python and NodeJS allowing us

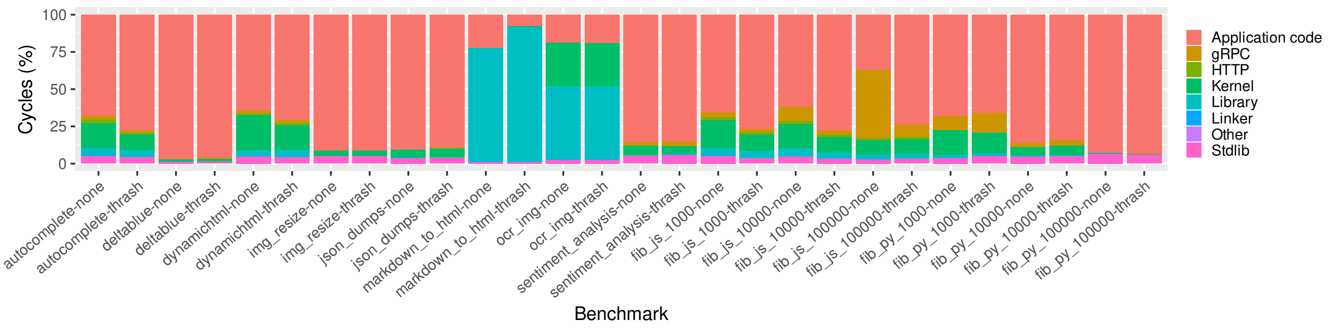

Figure 1: The breakdown of where executed cycles are spent divided into categories.

Table 2: Percentiles of the observed running times for the functions in ms.

Benchmark	50%	90%	95%	99%
autocomplete	0.26	0.29	0.31	0.38
deltablue	8.82	9.56	9.65	15.10
dynamichtml	0.47	0.51	0.54	0.74
img_resize	747.69	793.62	799.36	803.95
json_dumps	14.54	14.63	14.65	14.90
markdown_to_html	38.24	38.43	38.78	38.92
ocr_img	1164.65	1177.91	1179.57	1180.89
sentiment_analysis	2.03	2.14	2.17	2.23
fib_js_1000	0.25	0.27	0.29	0.34
fib_js_10000	0.34	0.43	0.47	0.63
fib_js_100000	0.48	0.59	0.64	0.86
fib_py_1000	0.52	0.60	0.62	0.66
fib_py_10000	1.82	2.11	2.28	4.49
fib_py_100000	97.74	98.37	98.40	99.15
footprint_long_16	10.52	10.58	10.61	10.70
footprint_long_32	20.75	20.83	20.86	21.00
footprint_long_64	52.93	53.04	53.16	53.36
footprint_long_128	121.25	121.41	121.64	121.95
footprint_long_256	260.76	262.45	262.65	262.82
footprint_short_16	0.30	0.35	0.37	0.40
footprint_short_32	0.32	0.37	0.39	0.42
footprint_short_64	0.37	0.43	0.44	0.48
footprint_short_128	0.49	0.54	0.55	0.59
footprint_short_256	0.74	0.80	0.81	0.84

to directly compare the runtime behavior of these two platforms. Since the runtime characteristics of the synthetic functions change depending on their invocation parameters, we use them to corroborate hypotheses derived from the behavior of the representative functions. The observed execution times of the functions are shown in Table 2.

3 MEASUREMENTS

This section presents the measurements we made with our functions. For each of these measurements, we compare back-to-back and interleaved function execution and discuss the consequences.

3.1 Where does time go?

The invocation machinery of a FaaS function is complex and multi-layered as it involves multiple components that are not directly related to a function's core functionality. To understand the potential of these components to impact the function execution behaviour, we analyze their contribution to a request's overall processing time, Figure 1. The sampled cycles are categorized primarily based on the Dynamic Shared Object (DSO, e.g. a shared library or executable) they originated from and, in some cases, secondarily subdivided based on the function that was executing when the cycle was sampled. This subdivision is necessary to correctly decompose NodeJS functions as they depend on platform-native libraries executed as JIT-generated code that do not appear as separate DSOs.

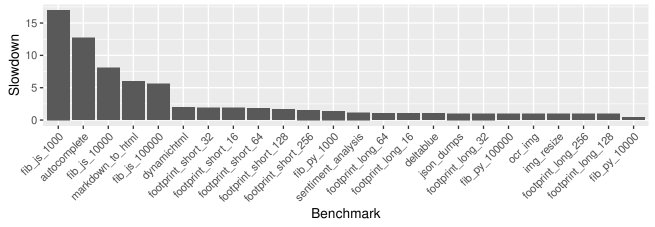

Figure 2: The wall-time slowdown encountered when running functions interleaved instead of back-to-back.

The categories shown in Figure 1 were chosen to represent the majority of the execution time. They are defined as follows:

Application Code The core part of the functionality of the function.

gRPC Cycles spent in the gRPC and Protobuf libraries.

HTTP Cycles spent processing HTTP requests.

Kernel Cycles spent in the kernel. Note that we do not trace cycles in this category back to the component that triggered their execution.

Library Cycles spent in libraries that are directly related to the core functionality of the function and only invoked from application code. For example, an image processing library is counted in this category whereas glibc is not.

Linker Cycles spent in the dynamic linker.

Stdlib Calls to C and C++ standard libraries. Like the kernel category, we do not identify who made the standard library calls.

In general, application or application-specific library code dominates the CPU cycle distribution regardless of the function and execution mode used. The principal pattern that emerges is that the functions with the shortest execution times (cf. Table 2) spend relatively more time in the function invocation machinery. This is not

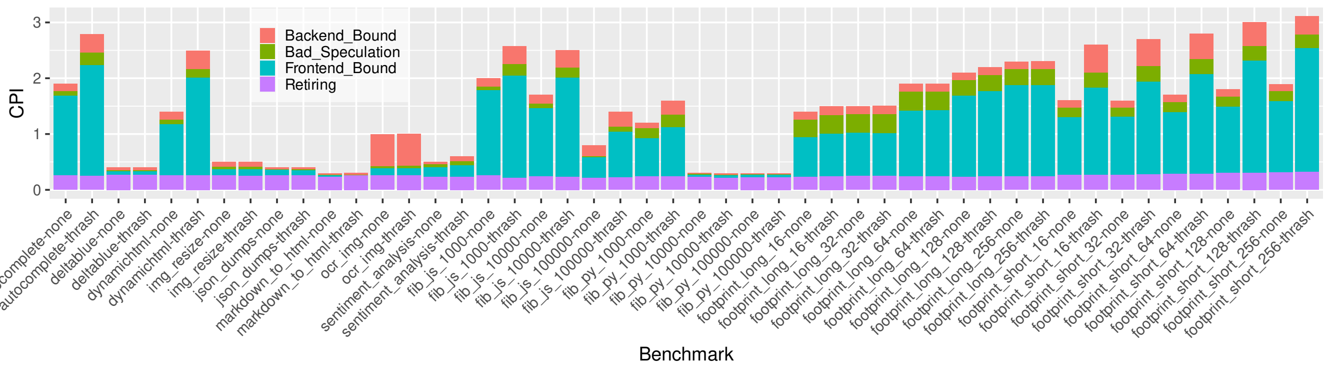

Figure 3: CPI stack for the functions broken down based on the contribution from each bottleneck category.

surprising considering that, unlike functions with longer execution times, they are unable to amortize the invocation overhead.

Next we look at how the cycle distribution changes when comparing back-to-back invocations of the functions to interleaved invocations. Examining this change in distribution informs us about how the various components involved in the function execution lifecycle are affected by the interference from the thrasher. If a component takes up relatively more cycles in the interleaved execution case, it means that the component is disproportionally negatively affected by the thrasher. Our results show that there are significant differences in the degree to which different functions are affected that largely depend on the function execution time. For example, in *autocomplete* and *dynamichtml*, which have very small execution time, we observe an increase in the fraction of cycles spent in application code with interleaved execution. In contrast, long running functions such as *img_resize* and *ocr_img* do not show much difference in cycle distribution.

Now, we demonstrate the significance of interleaved execution on functions with a short execution time from another perspective by comparing the elapsed per-invocation wall-clock time between the two invocation modes. Figure 2 shows the difference in request round trip time between back-to-back and interleaved executions as measured from the client. For the majority of the functions there is no significant difference in the execution time between the interleaved and back-to-back executions. We only see marked increases in execution time for very short functions with < 1 ms median execution time. The most extreme example is the 17× increase in execution time of the *fib_js_1000* function.

Finally, we also observe that the execution time alone does not always explain a function's sensitivity to interleaved execution. Compare, for example, *dynamichtml* to *fib_py_1000* which have very similar execution times as depicted in Table 2. For the *dynamichtml* function, we see that more cycles are spent on application code in the interleaved execution case, whereas the characteristics of *fib_py_1000* are largely unchanged. A reason for this behaviour can be that other function properties such as code and data footprints, control flow behaviour, etc. influence performance sensitivity to interleaved execution. As there can be a large number of function properties, we first analyze the ones that have the largest impact on function performance, in the next section, and then analyze the impact of thrashing on them.

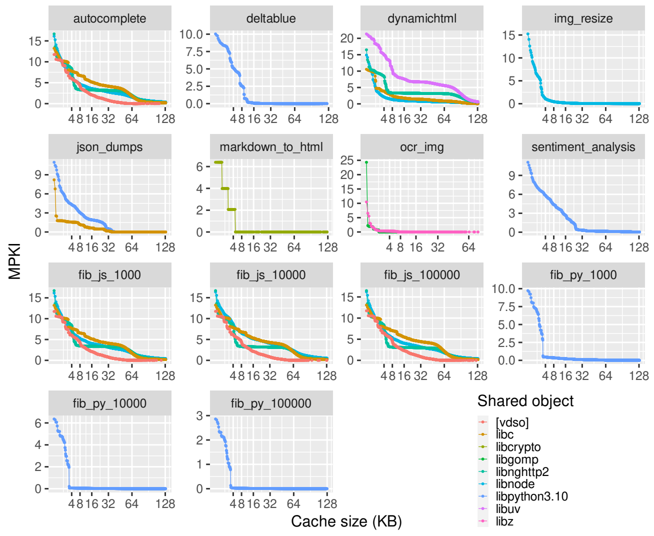

Figure 4: The estimated instruction working set sizes of the functions.

3.2 Microarchitectural analysis

To analyze the microarchitectural behaviour of our functions, we use the established Top-Down methodology [16]. The Top-Down methodology uses performance counters to estimate the fraction of pipeline slots that are stalled due to bottlenecks in specific parts of the processor. The top level of the analysis is broken down into four categories: *Retiring*, *Backend_Bound*, *Bad_Speculation*, and *Frontend_Bound*. The *Retiring* category covers slots containing retired instructions, that is, instructions that completed and committed their result. As such, this is the desirable category of Top-Down and we want to maximize the number of pipeline slots that fall in this category. *Backend_Bound* denotes slots that are stalled due to the execution units of the processor backend being unable to accept additional instructions. The *Bad_Speculation* category denotes slots that are stalled due to incorrect speculations, for example, branch mispredictions. Finally, the *Frontend_Bound* category contains slots that are stalled due to the frontend's inability to supply the backend with instructions at a sufficiently high rate. The identified

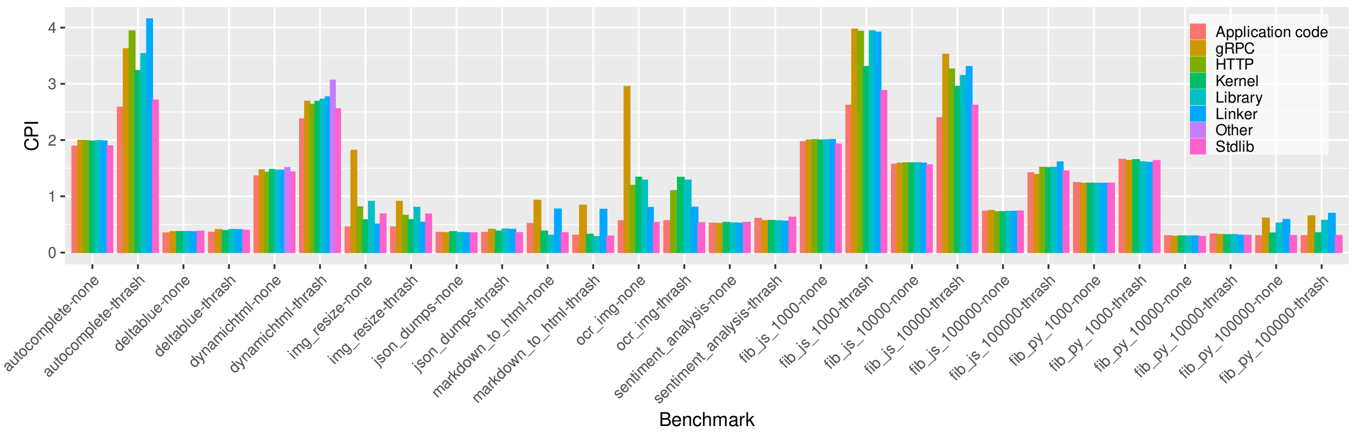

Figure 5: The CPI for the different components of the application.

bottlenecks can be broken down in a hierarchical fashion making it possible to identify a specific microarchitectural structure that is put under stress by the evaluated function.

The cycles per instruction (CPI) stacks resulting from the Top-Down analysis are shown in Figure 3. The CPI stack visualizes the contributions of the individual bottlenecks to the overall performance of the function. From the results, we see that there is a strong correlation between the execution time of a function and the instructions per cycle (IPC) rates that they achieve: shorter running functions achieve lower IPC rates (i.e., high CPI). Additionally, when comparing back-to-back and interleaved executions, the functions with low IPC also show the largest relative performance degradation in the interleaved execution mode.

The figure also implies that short running functions show low IPC because of the front-end bottleneck, i.e., they are *Frontend_Bound*. These results corroborate prior work which also found serverless functions and conventional server applications to be *Frontend_Bound* and proposed diverse mechanisms to mitigate this bottleneck [1, 2, 5, 6, 7, 10]. Further, prior work [10] also reported that instruction cache (L1-I) misses are the principal reason for this front-end bottleneck in serverless functions. This finding suggests that the instruction working set size of the functions also has the most significant impact on their performance. To show the instruction working set of a function impacts its performance sensitivity to interleaved execution, we estimate the instruction working set of our functions in the next section.

3.3 Estimating instruction working set

Motivated by the Top-Down analysis results, we now estimate the instruction working set size of our functions to understand how it impacts function performance. To perform the estimation, we use a method described in [15] adapted to work with traces gathered using Intel PT. The result is shown in Figure 4 giving the Cumulative Distribution Functions for the functions of the estimated MPKI rate of each function's shared objects depending on the instruction cache size. When the MPKI for a cache size reaches zero, it means that the entire instruction working set of the function fits the cache. Therefore, this cache size corresponds to the instruction working set of the function. For clarity, we only show the CDFs for shared objects that combined contribute 99% of the executed instructions or the single shared object that alone contribute more than 99% of executed instructions.

With these results, we can now shed further light on the data presented in Section 3.1 and Section 3.2. Comparing the *fib_py_1000* and *fib_js_10000* functions using the data from Figure 4, we see that the NodeJS version of the function has a significantly larger instruction working set than the Python version. The instruction working set of the Python version of the function is less than 4KB while the NodeJS implementation requires more than 64KB, a 16× difference. This observed correlation remains consistent across all of the measured functions.

Next, we look at how the instruction working set of a function affects its sensitivity to interleaved execution. Looking at the synthetic footprint functions (right side of Figure 3) and comparing them to *fib_py_1000* we see that, again, only functions with a short execution time are affected by interleaved execution. Additionally, for short-running functions a large instruction working set exacerbates the impact of interleaved execution. On the other hand, the performance of longer-running functions are unaffected by interleaved execution regardless of their instruction working set. The conclusion of this is that only functions with a very short execution time *and* and a large instruction working set see a performance degradation because of interleaved function execution.

Finally, our results highlight the importance of choice of programming language and runtime environment for application performance. For example, looking at *fib_js* and *fib_py* functions in Figure 3, the NodeJS implementation show significantly worse CPI. However, computing the 100,000th Fibonacci number takes 181× longer using the Python implementation than with the NodeJS implementation as shown in Table 2. This observation is far from novel but it highlights the magnitude of the gains that are possible by making purely application-level changes.

3.4 Category-wise performance

Lastly, we discuss the performance of the functions broken down by category. We divide the executed cycles into the same categories as in Section 3.1. The results are shown in Figure 5. The purpose of this experiment is to assess if particular parts of the application exhibits worse performance than the average. For the back-to-back executions, no particular component diverges significantly from the average performance of the function. For the interleaved execution, a different pattern emerges. Again, only the functions with the shortest execution times are affected but the components taking the smallest part of the total execution time are disproportionally affected. For example the Linker component contributes only a negligible fraction of the executed cycles (see Figure 1) of the *autocomplete* function and in the back-to-back execution scenario it exhibits the same performance as the application as a whole, around 2 CPI. However, in the interleaved execution scenario, its performance deteriorates more than 2×. Meanwhile, the performance of the Application Code category deteriorates slightly, from 2 to 2.5 CPI, ending up just below the overall function performance of 2.9 CPI (as seen in Figure 3). The same pattern can be observed for the *dynamichtml*, *fib_js_1000* and *fib_js_10000* functions.

4 DISCUSSION

The results of our study shows that functions with very short execution times (< 1 ms) benefit significantly from being executed on a processor with a warm microarchitectural state. However, as noted in the introduction, such short functions are quite uncommon in real-world applications. Additionally, even if the microarchitectural efficiency of these functions are improved using targeted optimizations the running time of the actual function may still only constitute a small fraction of the total round trip time of a function invocation. Meanwhile, larger functions, whose execution time contribute significantly to the total request round trip time, will not see any benefit from targeted microarchitectural optimizations.

These observations motivates research into high-level approaches for improving the execution time of the shortest functions. For example, functions that are executed in sequence as part of an application graph could be dynamically compiled together into a single component. Such an approach would preserve the function compositionality and modularity that are essential to the serverless application model while, at the same time, eliminate the overhead arising from the execution of a large number of short functions on both the microarchitectural and system level.

However, it is important to note that even if the execution overhead of serverless application graphs can be reduced by dynamically compiling functions into a single component, a large number of serverless applications consist only of a single function [3]. Furthermore, many of these requests are interactive, that is, the user that made the request expects an immediate response. The functions responding to such requests have an inherently short running time and are therefore likely to still see significant benefit from targeted microarchitectural optimizations.

5 CONCLUSION

This paper aimed to identify properties of serverless functions that predicts if a function is likely to benefit from warm microarchitectural state. To do this, we evaluated a suite of both real-world and synthetic functions to identify their per-invocation execution times and their instruction working set sizes. Subsequently we interleaved the function executions with a process that thrashes the microarchitectural state of the previously invoked function. By comparing the performance of the back-to-back and interleaved function executions across several metrics we identified key properties that makes a function likely to benefit from being executed from a warm microarchitectural state. We found that only the functions with a very short execution time (< 1 ms) and large instruction working sets are negatively affected by being interleaved with the thrasher. However, functions with longer execution times (> 50 ms) were not adversely affected by the microarchitectural state thrashing.

REFERENCES

[1] Truls Asheim, Boris Grot, and Rakesh Kumar. 2022. A specialized btb organization for servers. In *Proceedings of the 31st International Conference on Parallel Architectures and Compilation Techniques* (PACT '22). Chicago, IL, USA.

[2] Truls Asheim, Boris Grot, and Rakesh Kumar. 2021. Btb-x: a storage-effective btb organization. *IEEE Computer Architecture Letters*, 20, 2, 134–137.

[3] Datalog. 2021. The state of serverless. https://www.datadoghq.com/state-of-serverless-2021/. (2021).

[4] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '20). Association for Computing Machinery, Lausanne, Switzerland, 467–481.

[5] Tanvir Ahmed Khan et al. 2021. Twig: profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '21). Association for Computing Machinery, Virtual Event, Greece, 816–829.

[6] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the front-end bottleneck with shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '18). Association for Computing Machinery, Williamsburg, VA, USA, 30–42.

[7] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: a metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 493–504.

[8] H. Lee, K. Satyam, and G. Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. (July 2018), 442–450.

[9] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. 2018. Serverless computing: an investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. (Apr. 2018), 159–169.

[10] David Schall, Artemiy Margaritov, Ustiugov Dimitrii, Andreas Sandberg, and Boris Grot. 2022. Lukewarm serverless functions: characterization and optimization. In *Proceeding of the 49st Annual International Symposium on Computer Architecture* (ISCA '22). IEEE Press, New York, New York, USA, ??

[11] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '52). Association for Computing Machinery, Columbus, OH, USA, 1063–1075.

[12] Mohammad Shahrad et al. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, (July 2020), 205–218.

[13] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. [n. d.] Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[14] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, (July 2018), 133–146.

[15] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. 1995. The splash-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, 24–36.

[16] A. Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. (Mar. 2014), 35–44.