Mosaic: Harnessing the Micro-architectural Resources of Servers in Serverless Environments

Jovan Stojkovic, Esha Choukse†, Enrique Saurez†, Íñigo Goiri†, Josep Torrellas
University of Illinois at Urbana-Champaign †*Microsoft Azure Research - Systems*
jovans2@illinois.edu, {esha.choukse, esaurez, inigog}@microsoft.com, torrella@illinois.edu

***Abstract*—With serverless computing, users develop scalable applications using lightweight functions as building blocks, while cloud providers own most of the computing stack, allowing for better resource optimizations. In this paper, we observe that modern server-class processors are inefficiently utilized in serverless environments. Cores perform frequent context switches within function invocations and have a high degree of oversubscription. In such an environment, functions frequently lose their micro-architectural state in stateful hardware structures like caches, TLBs, and branch predictors, causing performance degradation. At the same time, modern processors are dimensioned for the needs of a broad set of applications, rendering them suboptimal for serverless workloads.**

Based on these insights, we propose *Mosaic*, an architecture optimized for serverless environments that maintains generality to efficiently support other workloads. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both serverless workloads and traditional monolithic applications, and (2) *MosaicScheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU. MosaicCPU slices micro-architectural structures into small chunks and assigns tiles of such chunks to functions. The processor retains the state of functions in their tiles across context switches, thereby improving performance. Furthermore, currently-inactive tiles are set to a low power mode, thereby reducing energy consumption. In addition, MosaicScheduler maximizes efficiency by introducing predictive right-sizing of the per-function tiles, alongside with smart scheduling based on the state of the tiles. Overall, compared to conventional server-class processors, Mosaic improves the throughput of serverless workloads by 225% while using 22% less power.

***Index Terms*—Cloud computing, Serverless computing, Hardware partitioning**

I. Introduction

Serverless computing is an emerging cloud paradigm that provides benefits to both users and providers. The basic execution unit is a lightweight function whose execution environment (*e.g.*, a container or virtual machine) is created on demand in an event-driven manner. Users develop applications using functions as building blocks while being charged in a fine-granularity manner, while cloud providers provision all resources and system services needed to run the functions. Hence, providers have the opportunity to co-locate many short-lived function containers on the same server and discard them once completed. Importantly, these lightweight functions share the infrastructure with various other workloads, including long-running monolithic applications. As a result, providers can maximize resource utilization by efficiently allocating resources across diverse workloads [89]. Today, serverless services are offered by major cloud providers [3], [26], [27], [49] and are widely used in various domains [5], [23], [41], [59], [82], [86].

Serverless workloads are a significant departure from workloads in conventional cloud environments. A typical function is *short-running* [33], [69] and its execution environment is *short-lived* [3], [69]. These properties introduce a range of challenges that undermine the efficiency of existing software and hardware. Prior work has addressed various software inefficiencies [12], [18], [24], [32], [37], [39], [43], [47], [58], [62], [69], [72], [74], [75], [78], [80]. Hardware inefficiencies have also received attention [63], [64], [68], albeit to a more limited degree.

In this work, we observe that serverless workloads use modern processors inefficiently for at least two reasons. First, large stateful hardware structures (caches, TLBs, and branch predictors) are poorly exploited, bringing marginal performance benefits while consuming substantial power. Indeed, frequent context switches in oversubscribed serverless environments [63], [64], [68], [74] cause functions to often interleave their executions on the cores, preventing the state in these micro-architectural structures from being reused.

Second, server-class processors [31] are designed to support a wide-range of compute and memory intensive workloads (*e.g.*, graph, database, and AI [30] applications). As a result, they are composed of relatively fewer, more powerful cores than would be ideal for serverless environments with many small functions. In serverless environments, it would be best to use servers with many smaller cores—which we call manycore servers. In this case, the state of many different functions can be concurrently stored in the structures of these many cores. However, having different types of servers for traditional monolithic workloads and serverless workloads would increase the cost for cloud providers.

In this paper, our goal is to enhance the performance of serverless workloads without introducing major changes to general-purpose server-class processors. To understand what processor architecture changes would benefit serverless environments, we characterize serverless functions on conventional processors. Using a production workload from *Microsoft Azure*, one of the largest serverless providers, we find that executing functions with a cold *micro-architectural* state increases their response time by 4×. In addition, typical functions are short-running, have small data and instruction footprints, and execute a small number of branch instructions.

Based on these insights, we propose *Mosaic*, an architecture optimized for serverless environments. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both serverless and traditional workloads, and (2) *MosaicScheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU. With Mosaic, serverless workloads efficiently share the same servers with traditional cloud workloads.

Mosaic is based on four main principles. First, MosaicCPU slices oversized hardware structures into chunks and assigns collections of chunks called tiles to individual functions. Second, MosaicCPU assigns resources to each function based on the needs of the function. Third, MosaicScheduler uses a performance model to predict a nearly-optimal assignment of tiles to functions after profiling a few invocations of the functions. Finally, MosaicCPU and MosaicScheduler are tightly coupled: the hardware exposes its current state to the software, which uses it for off-line performance modeling and on-line micro-architectural state-aware scheduling.

We prototype MosaicScheduler on an Intel Sapphire Rapids system [31]. Our evaluation with production-level function invocation traces shows that MosaicScheduler reduces the functions' tail latency by 28.3%. We evaluate the combination of MosaicCPU and MosaicScheduler using full-system simulations. On average, and compared to server-class processors, Mosaic reduces the functions' tail latency by 74.6%, improves their throughput by 225%, and uses 22% less power, while adding only 0.05% area overhead. Compared to an iso-area manycore, Mosaic reduces the functions' throughput by only 13%, without slowing down monolithic applications; however, the manycore processor reduces the throughput of monolithic applications by 68%. Thus, Mosaic efficiently runs various co-located workloads, reducing the cost for cloud providers.

This paper makes the following contributions:

- A characterization of the sensitivity of serverless functions to the sizes of the micro-architectural structures of general-purpose server-class processors.
- MosaicCPU, a general-purpose processor architecture that is highly optimized for serverless environments.
- MosaicScheduler, a software stack readily deployable on existing hardware that optimizes the execution of serverless workloads on MosaicCPU.
- An evaluation of Mosaic.

II. Characterizing Serverless Workloads on Current Server-Class Processors

To understand the micro-architectural inefficiencies of hosting serverless workloads, we characterize the execution of common serverless functions on conventional processors. We run experiments on an Intel Sapphire Rapids [31] server at 3.6GHz with a 2MB 16-way per-core L2 cache and a 1.875MB 15-way per-core last level cache (LLC) slice (more details in Table II). We use open-source functions [16], [36], [76], [87] and production-grade functions from *Microsoft Azure*. The evaluated functions are from different domains: image processing (*ImgProc* and *Thumbn*), video processing (*VidProc*), machine learning inference (*CnnSrv*, *RnnSrv*, *LrSrv*, grouped as *MLSrv*), data analytics (*EvStr* and *RiskQ*), document processing (*WordCnt*) and web services (*HotelB*, *SocNet* and *WebSrv*). These functions cover popular serverless use cases [11], [19], [66]. Next, we describe our main observations.

1. What is the impact of micro-architectural state on the performance of functions? Prior work observed that serverless functions often execute on polluted micro-architectural state due to frequent context switches [32], [74] and core oversubscription [1], [21], [34]. We quantify the impact of micro-architectural state loss by measuring the execution time of functions while varying the number of functions interleaved. Figure 1 shows the execution time of a function when, between two of its invocations, there are 2, 4, 8, or 16 different functions executed, normalized to the execution time of the function's isolated execution. To compare with a complete loss of state, we also run the *ClearAll* environment, which flushes all cache, TLB, and branch predictor state on context switch. The figure shows a few representative functions and the average of all our functions.

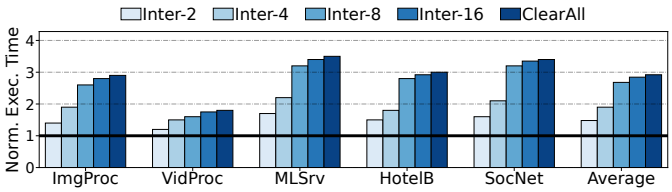


Fig. 1: Function execution time while varying the number of functions interleaved, normalized to isolated execution.

We see that all functions degrade performance when executing on a core with polluted micro-architectural state. As the number of interleaved functions increases, the execution time gradually increases. For functions that have significant state reuse across invocations (*e.g.*, *MLSrv*), or that frequently context switch within an invocation (*e.g.*, *SocNet*), completely losing the state increases the execution time by more than 3×. On average, *ClearAll* increases the execution time by 2.9×.

We also measure function interleaving on a core in a real-world deployment at *Microsoft Azure*. Given a function f, we observe that there are at least 8 and 16 other functions interleaved on a core, for 21% and 9% of consecutive invocations of f, respectively. Note that f is not evicted from memory, but it executes with polluted micro-architectural state. Thus, preserving the micro-architectural state of functions is of great importance in real-world serverless deployments.

2. How to preserve the micro-architectural state of functions? Servers are optimized for long-running applications with large data and instruction footprints. The size of stateful structures such as caches increases with every new processor generation. However, many serverless functions have substantially different needs. In this section, we contrast serverless functions with traditional monolithic cloud applications [52].

First, we measure the applications' LLC occupancy via Intel's *pqos* tool [29]. The right part of Figure 3 shows the

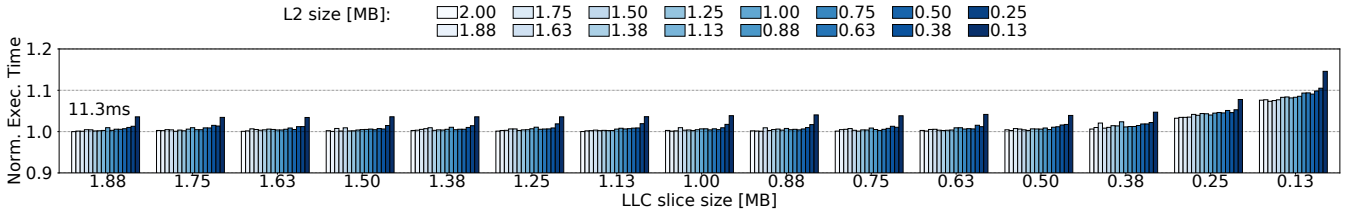


Fig. 2: Normalized execution time of the *ImgProc* serverless function with different sizes of L2 and LLC slice. The number on top of the leftmost bar is the execution time with full L2 and LLC slice sizes.

LLC occupancy of different serverless functions. Most of the functions use around 2MB. The average LLC occupancy of all our 10 functions is 2.9MB. On the other hand, the left part of Figure 3 shows the LLC occupancy of monolithic applications. We see that all applications use the maximum size of the LLC, which is 15MBs for our 8-slice experiments. Compared to serverless functions, monolithic applications are longer-running (a few minutes vs. a few milliseconds), have significantly larger memory footprints (10s of GBs vs. 10s-100s of MBs), and occupy the whole LLC and could benefit from even larger caching space (15MB vs. 2.9MB).

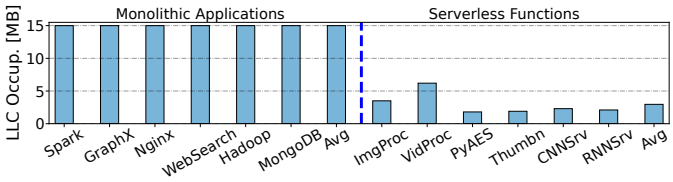


Fig. 3: Last Level Cache (LLC) occupancy for long-running monolithic applications and serverless functions.

The small LLC occupancy of serverless functions indicates that they can execute with a reduced cache capacity and still perform well. Hence, we use Intel's CAT [28] to execute functions and monolithic applications with different numbers of LLC ways per slice: 15, 10, 5 or 2 LLC ways per slice. Figure 4 shows the resulting execution time of monolithic applications (left) and functions (right). The bars are normalized to 15 ways per slice, which is the default design. We see that monolithic applications experience a severe increase in execution time if they run on smaller caches. For example, *DataSrv* increases its execution time by 50% when using 2 LLC ways rather than the default 15 ways. In contrast, the right part of Figure 4 shows that all functions barely change their execution time even when running with only 2 LLC ways. On average, using 2 instead of 15 ways per LLC slice increases the functions' execution time by only 2.8%.

We now reduce the size of both the L2 cache and the LLC. Figure 2 shows the execution time of a representative function, *ImgProc*, with different numbers of ways in the L2 and in the LLC slice. The groups of bars correspond to different LLC slice sizes and, within a group, there are bars for different L2 sizes. All bars are normalized to the case of full L2 and LLC slice size. From the figure, we see that even with a modest number of ways in the L2 and in the LLC slice,

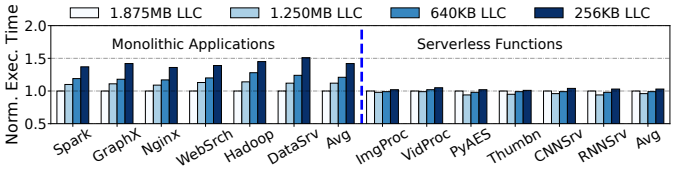


Fig. 4: Normalized execution time of monolithic applications and serverless functions with different LLC slice sizes.

the execution time of the function does not increase much. Therefore, serverless functions can still run efficiently with relatively small L2 and LLC caches.

Finally, we collect instruction traces with Pin [45] and simulate different sizes of branch predictors with the SST simulator described in Section IV. Our baseline architecture of Section IV has a 32KB TAGE-SC-L [67] branch predictor and an 8K-entry branch target buffer. Figure 5 shows the hit rate of the branch predictor and branch target buffer as we reduce the size of these structures for the *ImgProc* function. We consider structures with a fraction of the entries in the baseline structures and normalize the hit rates to that of baseline structures. We can see that 32× smaller branch predictor table and BTB reduce the hit rates by only 0.9% and 3.9%, respectively.

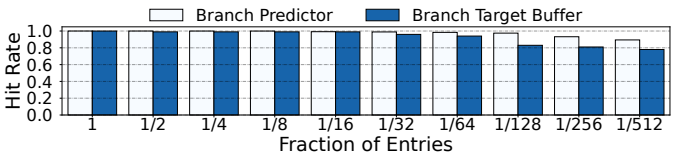


Fig. 5: Normalized hit rate of the branch predictor table and branch target buffer as we reduce the number of entries in the structures for the *ImgProc* serverless function.

In Mosaic, we exploit the small footprints of functions to partition structures and preserve the functions' state in their partition. Mosaic overcomes the inefficiencies of existing partitioning schemes (*e.g.*, Intel CAT [28]). Such schemes partition only some levels of the cache hierarchy, rather than additionally partitioning stateful structures (*e.g.*, branch target buffer). Moreover, they induce non-negligible ms-scale overheads when they change the core's allocation policy. Finally, they support only a small number of classes of service, limiting the number of concurrently stored states on the server.

3. Why maintain processor generality? Instead of creating a specialized core/accelerator for serverless workloads [71], Mosaic aims to maintain processor generality and introduce modest changes that allow a general-purpose server-class CPU to efficiently run both traditional and serverless workloads. There are three reasons for this decision: (1) handling inter-function heterogeneity, (2) reducing the provider's TCO, and (3) accommodating end-to-end cloud workflows.

First, although many functions execute acceptably in low-performance cores (*i.e.*, cores with small hardware structures, low frequency, and low issue width), some functions benefit from executing in high-performance server-class cores. We simulate the execution of our functions on a beefy core modeled after Intel's Sapphire Rapids [31] at 3.6GHz and on a small core modeled after ARM's A15 [7] at 2.5GHz. It can be shown that while WebSrv and SocNet see minimal performance degradation, MLSrv and ImgProc increase the response time by more than 47%. Some heterogeneous platforms such as ARM's big.LITTLE [8] include both high-performance beefy cores and energy-efficient small cores. However, they have a fixed number of each core type and cannot dynamically adapt to the workload.

Second, serverless workloads are only a fraction of the workloads in the cloud, and often share the same server with monolithic applications that require large cores [89]. Creating a separate cluster dedicated to serverless workloads substantially increases the TCO for providers due to the introduced fragmentation (as shown in Figure 22).

Finally, end-to-end serverless applications are often composed of both serverless functions and monolithic services. As an example, many serverless functions [25] use databases such as MongoDB [50] as their backends. For high performance, these different pieces of an application should run close to each other—ideally, on the same physical server.

4. What is the heterogeneity across functions? Different functions can have very different hardware requirements. As an example, Figure 6 shows the execution time of three functions, *RnnSrv*, *ImgProc* and *WordCnt*, when executing with different numbers of L2 ways. The execution time is normalized to the one with all 16 ways. We see that *RnnSrv* is highly sensitive to L2 size, and needs at least eight L2 ways, while *ImgProc* is modestly sensitive to L2 size and needs at least two L2 ways, and *WordCnt* is insensitive to L2 size.

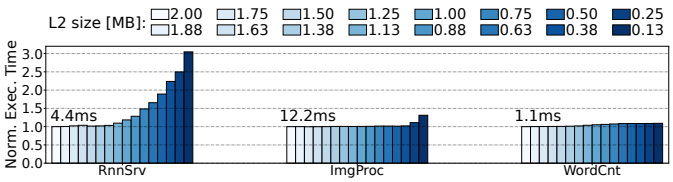


Fig. 6: Normalized execution time of functions executing on a core with a single LLC way per slice and different L2 sizes. The numbers on top of the bars are the execution times with a full L2 size.

To generalize a function's needs, we categorize functions into *Low*, *Medium*, and *High* intensity for data, instruction, and branches. These categories are determined by the function's data working set size, instruction working set size, and branch working set size (*i.e.*, the working set size of the cache lines that contain branches), respectively. Table I shows the categorization of some of the functions we use. Functions that fall in the same bucket typically need the same hardware structure size for optimal cost-performance.

TABLE I: Categorization of functions into low, medium, and high intensity for data, instructions, and branches.

Function	Data	Instructions	Branches
RiskQ	Low	Low	Low
EvStr	Low	Low	Medium
WordCnt	Low	Low	Medium
ImgProc	Medium	High	High
MLSrv	High	Medium	High
HotelB	Low	Low	Low
SocNet	Low	Low	Low
WebSrv	Medium	Low	Low

We use this categorization to classify a subset of popular production-level functions at *Microsoft Azure*. The majority of the evaluated functions have *Low* data intensity with an average data working set size of 2MB, *Medium* instruction intensity with an average instruction working set size of 12MB, and *Medium* branch intensity with an average branch working set size of 1.5MB. For comparison, monolithic applications [52] have 8GB, 0.7GB, and 0.3GB average data, instruction, and branch working set sizes, respectively.

III. Mosaic: Server Architecture Co-Design for Serverless Functions

To address the observed inefficiencies, we introduce *Mosaic*, a system optimized for serverless environments. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both monolithic applications and serverless functions, and (2) *MosaicScheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU.

Mosaic materializes the main insights of our characterization via four principles. First, MosaicCPU slices oversized hardware structures into fine-grained chunks and assigns collections of chunks called *Tiles* to individual functions. Each tile preserves the state of a function in the structure, maximizing the opportunities for state reuse across context switches and minimizing the interference between co-located functions. Second, MosaicCPU assigns a different tile size to each function based on the needs of the function. A tile spans non-contiguous entries in a structure. Third, MosaicScheduler uses a performance model to predict a nearly-optimal assignment of tiles to functions after profiling a few invocations of the functions. Profiling is performed online, while performance modeling is performed offline, outside of the functions' critical path. Finally, MosaicCPU and MosaicScheduler are tightly coupled: the hardware exposes its current state to the software, which uses it for off-line performance modeling and on-line

micro-architectural state-aware scheduling. Next, we detail each of the principles.

A. Fine-grained Per-Function Hardware Partitioning

1. Overview. In current processors, a function has access to the entirety of stateful structures of the core it is running on. When a function runs, it displaces the state of the functions that were running on the core before, preventing them from reusing the loaded micro-architectural state after they resume execution on the core. MosaicCPU overcomes this challenge by slicing large stateful hardware structures such as caches, TLBs, and branch predictor units into smaller physical partitions called *Chunks*, and dedicating a group of chunks called a *Tile* to individual functions. Figure 7 shows the high-level overview of the MosaicCPU architecture and its partitioning technique.

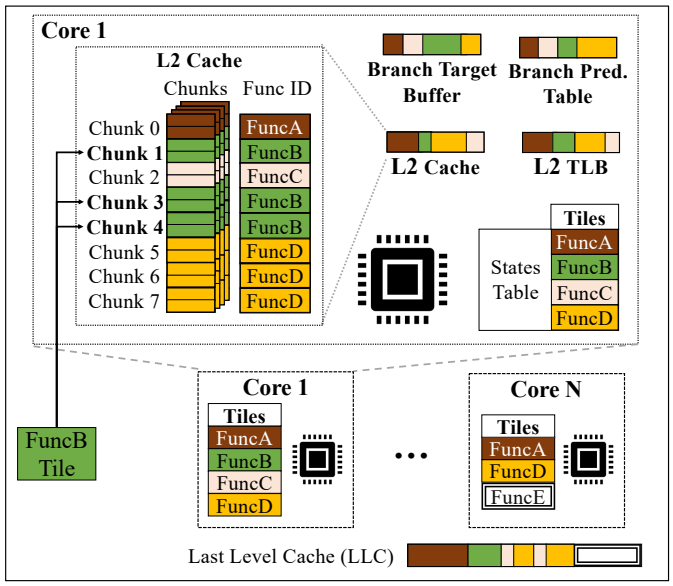


Fig. 7: High-level overview of the MosaicCPU architecture.

Hardware structures are partitioned into fixed-sized chunks of the same associativity. A chunk comprises a few physically contiguous sets in a given hardware structure. In principle, Mosaic can partition any stateful hardware structure. However, to minimize complexity and avoid potentially increased access latencies, Mosaic targets the most oversized structures whose state saving brings the most performance benefits and whose access latencies can be hidden. Specifically, it targets the following 5 structures: L2 cache, L2 TLB, LLC, branch target buffer, and branch predictor table.

A function is assigned a group of chunks called a tile in each of the partitioned structures. The chunks in a tile are not necessarily contiguous. A tile preserves the micro-architectural state of a function for future reuse. For example, assume that $FuncA$ yields a core to $FuncB$. To preserve $FuncA$'s state, MosaicCPU restricts $FuncB$ to access entries only within its tile, keeping the tile of $FuncA$ uncontaminated. During $FuncB$'s execution, MosaicCPU translates the accesses of $FuncB$ to the correct physical chunks.

Each partitioned structure has tags indicating which function owns which chunk. The chunks with state for functions that are not currently running are placed in a low power mode. The goal is to save power while still retaining the state in the chunks. On a context switch, the chunks of the pre-empted function are put in low-power mode and the chunks of the new function are activated.

A MosaicCPU has a hardware *States Table* that tracks which functions currently hold their state in each of the core's structures. As shown in Figure 8, the States Table contains as many entries as the maximum number of functions that can concurrently hold state in the core structures. A given entry contains the function ID and, for each of the partitioned structures, the number of chunks currently assigned to the function and the number of chunks that the function requires to run efficiently based on the MosaicScheduler prediction. In addition, the States Table entries have LRU bits. When a new function needs additional chunks, they are taken from the LRU function. The OS keeps one chunk in each structure.

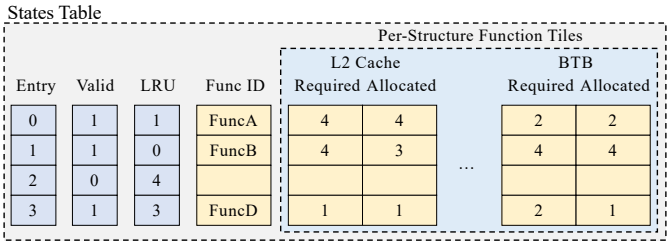


Fig. 8: The *States Table* in MosaicCPU tracks which functions currently keep their state in the core's structures.

2. Assigning tiles and chunks. MosaicCPU includes mechanisms to assign and deassign chunks to/from functions. When a core schedules a function invocation for execution, the OS first checks the States Table to see if the function has the required number of chunks in all the core's structures. If so, we call this a *hit*. In this case, for each structure, the OS sets the chunks of the previously-running function to low-power mode and activates the chunks of the new function. The LRU bits in the States Table are updated.

If, instead, the new function is not in the States Table or it is there but does not have all the required chunks in all the structures, a *miss* occurs. If the function is not in the States Table, the OS allocates an entry for the function in the table, which includes filling in the Function ID and the required chunks in each of the structures. Further, in all the miss cases, the OS computes the number of chunks that the new function is missing in each structure, then harvests such number of chunks either from unused chunks or from chunks owned by the LRU function or functions, and finally reassigns these victim chunks to the new function. This process involves updating the States Table and then, for each of the structures, reassign the victim chunks. The latter consists of activating the victim chunks, writing back and invalidating their entries, and updating the function ID tags of such chunks in the structure. Writebacks are only needed for the dirty lines in the caches

and the updated state bits in the TLB.

After this, the same operations as a hit occur: the chunks of the previously-running function are put in low-power mode, all the chunks of the new function are activated, and the LRU bits in the States Table are updated.

For the state in Figure 8, if the core attempts to execute $FuncA$, a hit will occur. However, if it tries to execute $FuncB$, $FuncD$, or a new function $FuncE$, a miss will occur.

We model the write back overheads in our evaluation and see that they are tolerable: the average overhead of writing back a chunk is 500-600ns, while the highest overhead is 1.5-2 μs. In a secure cloud environment, the whole cache hierarchy is flushed at a context switch [10], [83], causing overheads of a few ms, while offering isolation equivalent to Mosaic's.

3. Memory de-duplication. Currently, Mosaic does not allow memory de-duplication across users, following security guidance from state-of-the-practice [79] and state-of-the-art [44]. However, Mosaic can be extended to support deduplication and reduce the total memory footprint of the server, while potentially sacrificing security. In such an environment, shared pages (*e.g.*, libraries and other code) can be marked as read-only and shared in the page tables, and placed in a separate *shared-page* tile. This tile is always active and shared across all functions. When a core issues a request for a shared page (indicated by a bit in the TLB), the structures search the shared-page tile. Otherwise, they search the function's tile.

4. Fine-grained power setting. Mosaic sets the voltage of idle chunks to a lower value than the active chunks, so that the static power of idle chunks is minimized. This is accomplished by having two voltage rails, Vdd_{high} and Vdd_{low}, and selecting the rail for each chunk based on an Active bit. As a result, Mosaic does not need a voltage regulator per chunk. The simple logic used is shown in Figure 9.

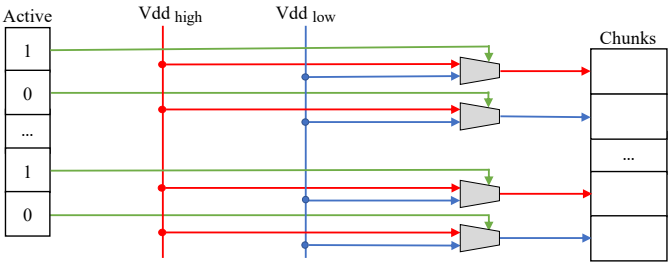


Fig. 9: Per-chunk voltage selection in Mosaic.

A similar approach has been implemented in prior work, with different voltage sources per section of the cache [85] or even per cache line [22]. Switching the voltage for a chunk has low overhead. The transition time for a drowsy cache line is 1-2 cycles [22], while the transition for the whole core is in the order of 1-2 μs [84]. In Mosaic, the chunks transition between voltage settings only on a context switch; hence the overhead is negligible. To reduce the overhead even further, we can predict when a chunk will need to change rails and perform the change in advance in the background.

5. Maintaining cache coherence. As in conventional systems, different functions communicate with each other via RPC messages and not via shared memory [73]. However, Mosaic caches still need to support data sharing in situations such as multi-threaded functions, concurrent invocations of the same function, and migration of invocations across cores. Thus, Mosaic caches are coherent. To achieve so, Mosaic ensures that snooping hardware can snoop chunks that are in low-power mode [22], [84]. Coherence messages and responses in Mosaic include the FuncID, so that the controller of the receiver cache can use the FuncID to identify the correct destination chunk (Section III.B4). Another possible approach would be to keep an up-to-date translation between core ID and running FuncID for all the cores in the snooping hardware of all the caches. In this case, the receiver cache would identify the correct FuncID based on the ID of the core that sent the coherence message.

B. Accessing a Function's Tile

1. Translation process. Since functions are heterogeneous, having a uniform tile size for all functions and in all structures is inefficient. Instead, Mosaic sizes the tile of each function in each structure differently, based on the requirements of the individual function. For instance, data-intensive functions may get larger tiles (*i.e.*, tiles with more chunks) in data caches, while branch-intensive functions may get larger tiles in branch prediction units. Allowing non-uniformly sized tiles may fragment the structures. Hence, to avoid fragmentation, function's tile is allowed to span physically non-contiguous chunks in a given structure.

Accessing the tile of a function in a hardware structure requires a level of indirection: the function's addresses need to be mapped to the correct positions in the structures. Consider the case of a cache. Recall that, in a conventional cache, the hardware splits the address into *tag*, *index*, and *offset* bits. In Mosaic, the *index* bits are separated into two parts: if chunks have S sets, then the $\log S$ least significant bits specify the set within a chunk (*Set* bits), and the rest of the bits specify the chunk (*Chunk* bits). This is shown in the upper part of Figure 10. The figure shows a 2MB L2 cache organized into 2K sets of 16 ways. Mosaic splits the cache into 16 chunks of 128 sets and 16 ways. For simplicity, we only show 4 ways per set and 4 sets per chunk.

Most of the time, a function's tile will not cover the whole structure and, instead, will have a number of chunks C lower than the maximum number of chunks C_0. In our design, S and C are powers of two. Let us call *index* the value of the index bits. Then, the hardware computes the ID of the chunk of the function that needs to access as $(index/S)\%C$. This operation is performed as simple bitwise shift operations.

Mosaic enables such operation by the *Mask* register shown in Figure 10. Mask has its $\log C$ least significant bits set to 1 and the rest to 0. Mask masks out some of the most significant bits (MSBs) of the Chunk field. In the example, assume that C=8. In this case, Mask masks out the MSB of Chunk as it generates the chunk ID.

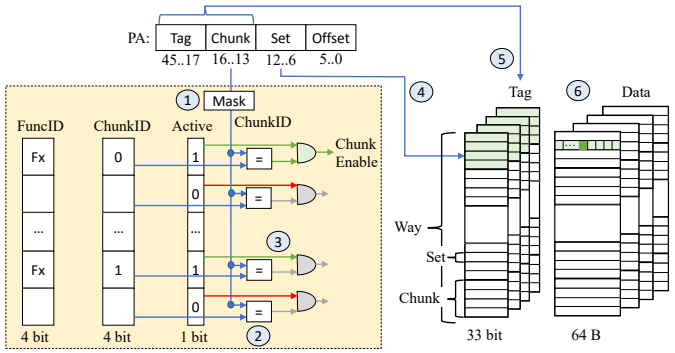


Fig. 10: Detailed micro-architecture of Mosaic's L2 cache.

To identify the desired chunk in the structure, Mosaic tags each chunk with the function ID and the ID of the chunk in the function (*i.e.*, the chunk's logical index within the tile). In addition, there is an Active bit per chunk that is set only for the chunks of the currently running function. This is shown in Figure 10, which assumes that function *Fx* is running. With this support, the output of the Mask register is compared to the ChunkID and Active bits of all the chunks to determine the target chunk.

2. Example operation. We showcase the access to two representative partitioned structures: L2 cache and branch predictor. The other structures, namely TLB, BTB, and LLC, follow the same principles.

L2 cache. Figure 10 shows how the L2 cache is accessed. We assume that a core issues a 46-bit physical address (PA). Moreover, as multiple logical chunks may map to the same physical chunk, Mosaic uses the combination of *Tag* and *Chunk* bits as cache tag.

The whole translation is as follows. Mosaic takes the *Chunk* bits and uses *Mask* to generate the chunk ID bits (1). These bits are compared to the *ChunkID* field in all 16 L2 chunks (2). As there can be multiple chunks tagged with the same ChunkID but owned by different functions, Mosaic checks the ownership of the chunk (3). There are two ways to implement this functionality: use the chunks' Active bits or compare the FuncID tag with the ID of the currently-running function. Mosaic uses the former approach for single-threaded cores (as shown in the figure), and the latter approach for SMT cores and coherence messages. After choosing the correct chunk, Mosaic uses the *set* bits from the PA to select the set in the chunk (4). Then, it compares the tags of all the ways of the set with the *tag* and *chunk* bits from the PA (5). Finally, Mosaic reads at the offset determined by the PA's *offset* bits (6), as in conventional caches.

Branch predictor. We model a state-of-the-art 32KB TAGE-SC-L branch predictor [67]. The predictor is composed of a base predictor (2-bit counter bimodal table T0) and 15 tagged tables with different history lengths and number of entries per table. The base predictor is directly indexed using the program counter (PC), while the tagged predictors are indexed using a hash of PC and a subset of history length.

Mosaic does not change the predictor's functionality. It only modifies the way the predictor tables are accessed, as shown in Figure 11. Specifically, each table (T0 to T15) is split into 16 chunks (for simplicity, the figure shows only three tables). The sizes of the chunks are different in different tables (as the total number of entries differs across tables), and a function is assigned the same number of chunks and the same chunk IDs in all the tables. Thus, the tables share the translation mechanism, Mask, and ChunkID/Active tags. After computing the hashes for all table accesses (a), Mosaic breaks the table index into *chunk* and *set* bits. It translates the chunk bits using the same principle as for the L2 cache (b). Then, it uses the translated *chunk* and the *set* bits to access the correct entry in the table (c). Beyond this, Mosaic does not change the baseline functionality of the branch predictor. Each table makes its own prediction, and the table indexed with the longest history and a tag match produces the final prediction (d). As in L2 cache, the tables are tagged with *tag* and *chunk* bits.

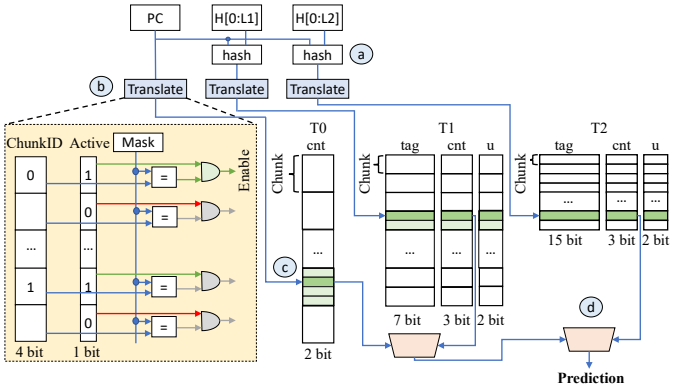


Fig. 11: Architecture of Mosaic's branch predictor based on state-of-the-art TAGE-SC-L [67].

Overall, the process of translating addresses for the partitioned structures is simple. It involves only comparing for equality the four bits of *ChunkID* with the PA's masked *chunk* bits, and an AND-gate with the *Active* bit. We estimate that this takes about one processor cycle.

3. Minimizing function tags overheads. Mosaic software assigns a 64B FuncID to each function—*e.g.*, a hash of the function name and the user ID [69]. Tagging all the chunks of this function with this FuncID in all the partitioned structures is a non-negligible storage overhead. To reduce this overhead, MosaicCPU exploits the fact that only functions that have an entry in the States Table of a core can have an allocated chunk in any of the partitioned structures of that core. Hence, MosaicCPU uses a function's *entry number* in the States Table as its unique ID. For example, in the system of Figure 8, the chunks of $FuncA$ and $FuncB$ are tagged with 0 and 1, respectively. This approach saves substantial space in the partitioned structures.

This improved design requires snooping caches to keep a small table that maps FuncIDs to States Table entry numbers. When an incoming coherence message is received, its FuncID

is translated into an entry number using this table before checking against the tags of the chunks in the cache.

C. Tile Sizing for the Functions

1. Overview. To pack the state of many functions in a core, Mosaic tries to allocate, for each function, the minimum number of chunks in each partitioned structure that still deliver acceptable performance for the function. Assume that a function execution completes in C_{full} cycles on a core that does not partition structures. Then, Mosaic attempts to find an assignment of chunks to the function that uses the minimal amount of resources and still completes execution in $C_{par} = C_{full} \times (1 + Threshold)$ cycles, where $Threshold$ is a small value like 0.1.

Determining the optimal tile sizes for a function through exhaustive search is impractical, given the large exploration space. One would need to execute the function with all possible combinations of tile sizes in all of the partitioned hardware structures, resulting in a few thousand invocations.

To address this limitation, Mosaic uses MosaicScheduler to profile a few invocations of a function online with live traffic, and then create a performance model offline to use for predictions. Moreover, to reduce profiling overheads, MosaicScheduler also uses *Transfer ML* to predict a function's optimal tile sizes based on previously-profiled similar functions.

2. MosaicScheduler. We consider the two prediction methods.

Predictions via performance modeling. A function is initially profiled with a few configurations. During the execution with each configuration, MosaicScheduler collects the number of cycles, the IPC, and the misses in each of the five partitioned structures. Then, MosaicScheduler picks the configuration with the smallest tiles that still finishes execution within the required deadline as the *temporarily optimal function size*. Finally, offline, MosaicScheduler takes this configuration and predicts if any of its tiles can be further reduced.

For this prediction, MosaicScheduler considers each partitioned structure in turn. For each one, it examines the trend of misses as the size of the tile in the structure decreases and, using the expected penalty of a miss in that structure, estimates the function execution time if the tile in the structure is reduced by one more notch. For example, as it considers the BTB, MosaicScheduler observes the trend of miss increases with smaller tile sizes in the BTB, predicts the extra number of misses that will occur if the BTB tile size is reduced one more notch and, given the cost of a miss, estimates the overall function execution time with the smaller BTB tile size. If the longer function execution time is acceptable, MosaicScheduler reduces the function's tile size in the BTB.

If there are multiple equally-desirable options, *e.g.*, the size of either the L2 cache or the BTB can be reduced, MosaicScheduler stores these options as function alternatives. Different alternatives can be used at different times depending on which other functions are running concurrently and sharing the structures.

All future invocations of a function execute with the predicted tile sizes. Mosaic does not change a function's tile sizes during an invocation of the function, but different invocations of the same function can execute with different configurations over time. MosaicScheduler monitors the execution and may recompute the function's tile sizes if the execution time is too long or too short.

This model yields high accuracy with low overhead. Typically, MosaicScheduler needs only 8-10 function invocations to accurately set the optimal sizes of a function's tiles.

Predictions via transfer ML. To minimize the profiling overheads, MosaicScheduler also uses an approach based on transfer ML. Instead of repeated profiling to establish the best tile sizes for a function, MosaicScheduler takes some high-level characteristics of the function (data and instruction footprint, number of branch instructions, and IPC) and compares them to the characteristics of some already-profiled functions. Then, with a regression model, MosaicScheduler finds the most similar functions and predicts the optimal sizes of the tiles for the new function based on these similarities. To achieve high accuracy, our prediction system needs only about 10 functions of different properties to be initially profiled. Note that the regression model is periodically augmented and retrained with new functions as they execute in the system.

Figure 12 shows the approach. A random forest classifier using a database of already profiled functions takes the characteristics of a function (*FuncX* in the figure) and generates the optimal sizes of the tiles for the function. We implement the classifier in Python's scikit-learn library [55]. As the model outputs the tile size for each of the partitioned structures, it is wrapped around a MultiOutputClassifier [54]. The model uses 100 trees, and the minimum number of samples required to split an internal node is 2. The size of a tile is classified into one of five classes: 1, 2, 4, 8, or 16 chunks. The prediction is done in software and off the critical path of function execution.

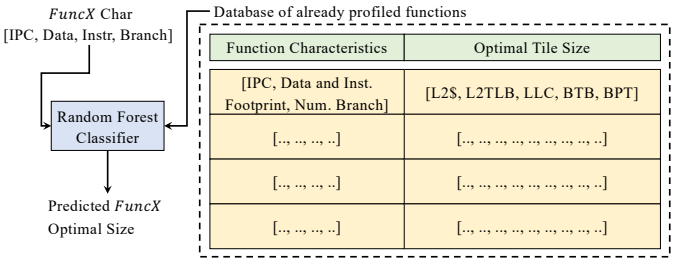


Fig. 12: Transfer machine learning architecture used to predict the optimal size of new functions.

D. Scheduling Function Invocations

1. Overview. In conventional serverless frameworks, function invocations are scheduled on a random core or on the least-loaded core [35]. This approach would diminish the benefits of Mosaic. Instead, to maximize performance, MosaicScheduler is aware of the state in the cores when scheduling invocations. It uses the interface exposed by the hardware and a set of heuristics to decide on which core to place an invocation.

2. Scheduling invocations. Figure 13 shows how MosaicScheduler schedules function invocations on cores. When a function invocation (such as the one for *FuncX* in the figure) arrives at the server, MosaicScheduler checks the predictor for the function's optimal tile sizes. Based on this information and the state of the cores, MosaicScheduler picks one core to execute the invocation. Then, it deposits the invocation augmented with the tile size information on the software request queue of that core.

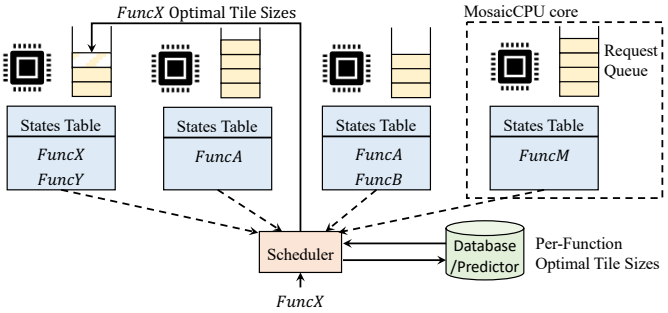


Fig. 13: Scheduling a function invocation in Mosaic.

As shown in Figure 13, MosaicScheduler reads the state of the States Tables of the cores to make its decision on where to schedule the invocation. Generally, it favors a core that already holds state for the function. Specifically, if there is one or multiple cores with function state that are idle, the scheduler randomly picks one of them. If there are no cores with function state, the scheduler picks a core to balance load across cores.

However, if all the cores with function state are busy, the scheduler predicts in which case the invocation execution will complete sooner: 1) if it waits in one of these cores until it can execute and reuse the state or 2) if is assigned to another, possibly idle core and can start executing sooner. MosaicScheduler predicts the waiting time in the queues and the execution time of the function invocation based on the profiles of the functions. MosaicScheduler then picks the core that would minimize the sum of waiting time and processing time for the invocation.

To deal with occasional mispredictions, the system also uses work-stealing. When a core becomes idle, its worker thread periodically checks if there are cores with queued up invocations. If so, the thread fetches invocations from other cores and executes them locally. In this way, the system is robust to head-of-the-line blocking.

3. Advanced scheduling policies. A scheduler that is unaware of the resource needs of functions may co-locate functions that require intense use of the same resource on the same core. Such core would then suffer high contention on one resource while the other resources would be underutilized. This problem is not present in Mosaic. MosaicScheduler balances resource utilization by spreading functions with similar resource requirements across the cores of the server. Specifically, when scheduling a function invocation that has no state in any of the cores, the scheduler checks which core has enough idle chunks to satisfy the invocation's predicted needs or which core would observe the least number of chunk evictions. This approach could be generalized for function placement across servers in a cluster. The cluster controller could classify functions as cache, TLB, or branch intensive and place only functions with distinct requirements on the same server.

4. Non-serverless workloads. Mosaic cores operate in two modes: serverless and non-serverless. These modes are activated by setting the *FaaSMode* register. In the serverless mode, all Mosaic optimizations are enabled. Conversely, in the non-serverless mode, the cores operate in a conventional manner, without structure partitioning. This enables high-performance execution for monolithic applications.

Privileged software such as the Virtual Machine Manager (VMM) sets the FaaSMode register on a cross-VM context switch. When changing the FaaSMode register, the state of all the partitionable structures is written back and invalidated. Hence, the VMM tries to schedule non-serverless workloads on cores that are already in non-serverless mode. The FaaS platform controllers [6], [38], including MosaicScheduler, run on one or more dedicated cores in non-serverless mode.

5. Harvest VMs. To reduce cost, some serverless environments use harvest VMs [89]. A harvest VM is created with a minimal number of cores, but it can dynamically grow and shrink by harvesting idle cores and releasing them when they are needed by higher-priority VMs (called primary VMs). Primary VMs run latency-sensitive workloads, while harvest VMs run workloads that can tolerate resource fluctuation, including serverless functions. Mosaic can run in such environments. When a core context switches between a primary and a harvest VM, the VMM changes the FaaSMode register of that core.

6. Mosaic beyond serverless workloads. While our focus is on serverless functions, Mosaic can offer benefits to other workloads that also exhibit frequent context switches, small working sets, and short execution times. One example of such workloads is microservices, which are used in many cloud deployments. Recent studies from Google [65] and Alibaba [46] show that microservices often require numerous RPC invocations, leading to frequent context switches that challenge traditional execution environments. For example, an individual microservice may issue hundreds of RPC calls. Mosaic can preserve the micro-architectural state across context switches, thereby enhancing microservice performance.

IV. Evaluation Setup

1. Systems modeled. Our base architecture is a server with 16 cores and 128GB of main memory. Each core is a 6-issue processor modeled after Golden Cove micro-architecture in Intel Sapphire Rapids (SPR) [31]. Table II shows the detailed micro-architectural parameters.

We use this base architecture to evaluate six server systems. (1) *Baseline* is a conventional server that does not partition hardware structures and schedules function invocations on the least-loaded cores. (2) *Baseline+Affinity* augments Baseline with simple affinity scheduling: a function invocation is sched-

TABLE II: Architectural parameters used in the evaluation.

Processor Parameters	
Multi-core chip	16 6-issue OoO cores, 512-entry ROB, 3.6GHz
L1 data cache	48KB, 8-way, 4 cycles round trip (RT), 64B line
L1 instruction cache	32KB, 8-way, 4 cycles round trip (RT), 64B line
L2 cache	2MB, 16-way, 16 cycles RT, 30 MSHRs
L3 shared cache	Slice: 1.8MB, 15-way, 60 cycles RT, 30 MSHRs
L1 data TLB	256 entries, 4-way, 2 cycles RT
L1 instruction TLB	256 entries, 4-way, 2 cycles RT
L2 TLB	2048 entries, 8-way, 12 cycles RT
Branch predictor	32KB TAGE-SC-L [67], 15 cyc. mispred. penalty
Branch target buffer	12K-entry, 4-way
Main-Memory Parameters	
Capacity;	128GB
Frequency; Rate	1GHz; DDR

uled on the core that last executed the same function if the core is idle. (3) *Baseline+MosaicScheduler* runs the software support of Mosaic on conventional hardware while partitioning L2 and L3 caches using Intel CAT [28]. This system maintains a *States Table* per core in software, and does not require any non-conventional hardware support. (4) *Mosaic* is our complete Mosaic design. (5) *Jukebox* [63] is a recent hardware proposal for serverless environments that enhances Baseline with hardware-supported instruction prefetching. (6) *Manycore* is a conventional server with 128 simple cores similar to ARM Cortex A15 [7] that has the same area as Baseline.

We evaluate the architectures with full-system simulations. We use the QEMU [70] emulator integrated with a modified SST framework [57] and DRAM-Sim2 [60] memory simulator. In this way, we simulate the operating system (Ubuntu 20.04), the serverless software stack, and our benchmark functions. To validate our simulation accuracy, we also run Baseline, Baseline+Affinity, and Baseline+MosaicScheduler on the SPR server used in Section II and calibrated the simulator. With L2 and L3 cache partitioning with Intel CAT [28], the server has 8 classes of service (COS).

Our simulation infrastructure models the two main Mosaic overheads. On a chunk eviction, we model efficient hardware that walks the tags of the chunk writing back all dirty entries and then invalidates all entries. Further, on changing the voltage rail of a chunk, we add a fixed 5-cycle latency. To model the area and power overheads, we use McPAT [42] and CACTI [9]. Recall that Mosaic increases the tag width of the partitioned structures by 4 bits. The increased tag size does not affect the access latency and only marginally increases the per-access energy. Mosaic further adds a hardware States Table per core, per-chunk ChunkID, FuncID, and Active bits in the partitioned structures, and some other small structures. Overall, this hardware adds 43.5KB of storage per core, which is 1.06% of the total core storage and 0.42% of the core area.

2. Workloads. We execute eight open-source serverless functions described in Section II: *ImgProc*, *MLSrv*, *EvStr*, *RiskQ*, *WordCnt*, *HotelB*, *SocNet*, and *WebSrv*. We invoke the functions with real-world invocation traces from *Microsoft Azure*. The traces cover peak-load and include 71,434 invocations of 64 different functions. We randomly map each production function to one of the eight evaluated functions. When multiple production functions map to the same evaluated function, we create separate instances for such functions, which do not share any data or instructions.

In some experiments, we evaluate monolithic applications from CloudSuite [52] which span multiple domains: data processing (*Spark* and *Hadoop*), graph applications (*GraphX*), web frontends (*Nginx*), web search engines (*WebSrch*), and data serving (*DataSrv*).

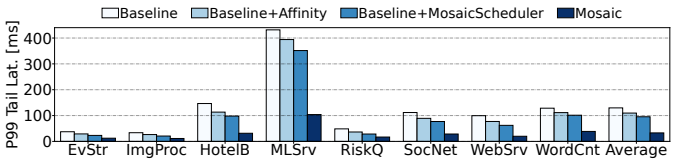
V. Evaluation Results

A. Performance Improvements

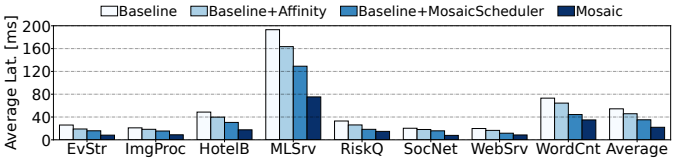
We measure the end-to-end latency and throughput of the function invocations on the evaluated systems. The end-to-end latency of a function invocation is the time from when the client sends a request until when it receives the result.

1. Tail latency. Figure 14a shows the P99 tail latency of our functions in four architectures. We see that, on average, adding affinity scheduling with Baseline+Affinity reduces the tail latency by 15.5% over Baseline. On top of Baseline+Affinity, partitioning the L2/L3 caches with Baseline+MosaicScheduler reduces the tail latency by 12.8%. Mosaic is much more effective: it reduces the tail latency over the Baseline by 64.8%–79.9%, with an average of 74.6%. The latency reductions are higher for functions with short duration (*e.g.*, *EvStr*), or frequent context switches (*e.g.*, *HotelB*).

Every partitioned hardware structure contributes to the reduction in tail latency. For example, it can be shown that not saving the function state of the L2 cache or the branch target buffer in Mosaic increases the tail latency of functions in Mosaic by 46% or 34%, respectively.



(a) Tail latency



(b) Average latency

Fig. 14: Tail and average latency of the function invocations.

2. Average latency. In addition to reducing the tail latency, Mosaic also reduces the average latency. Figure 14b shows the average latency with the four evaluated systems. Baseline+MosaicScheduler reduces the average latency over Baseline by 28.7%. As an average function invocation is more likely to execute on a warm micro-architectural state than the invocations at tail, Mosaic's benefits are slightly lower for the

average latency than for the tail latency. On average, Mosaic reduces the average latency by 59.6% and 37.4% over Baseline and Baseline+MosaicScheduler, respectively.

3. Throughput. We define the throughput as the maximum load a system can sustain without violating the SLOs of functions. The SLO of a function is defined to be 5× the execution time of the function on an unloaded system. Figure 15 shows the throughput for the evaluated functions in kilo requests per second (kRPS) with Baseline and Mosaic. Mosaic substantially improves the throughput across functions. On average, it improves throughput by 225%. The reason is that, with Mosaic, invocations can reuse their state while still multiplexing their execution on a core with many other invocations of the same or different functions.

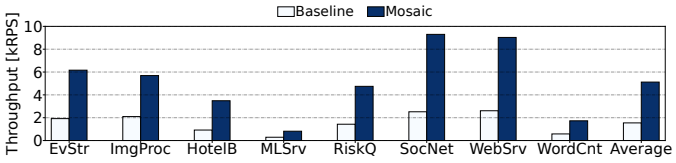


Fig. 15: Per-function throughput with Baseline and Mosaic.

4. Comparison to prefetching. Figure 16 compares the tail latency of Baseline, Jukebox [63], and Mosaic. By reducing the instruction misses, Jukebox reduces the tail latency over Baseline by 12.1% on average. However, Jukebox does not reduce misses in data caches or branch predictors, and introduces additional memory traffic for prefetching. In addition, after a running function invocation is pre-empted and then scheduled to run again, Jukebox does not reprefetch the instructons. Compared to Jukebox, MosaicCPU reduces the tail latency by 71.1% while using less on-chip area.

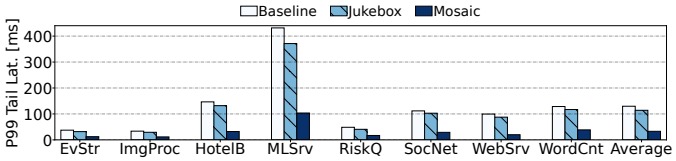


Fig. 16: Tail latency of Baseline, Jukebox, and Mosaic.

5. Comparison to a manycore. Figure 17 shows the throughput of Manycore and Mosaic for monolithic applications and serverless functions normalized to Baseline. Due to space limitations, the figure does not show all the serverless functions, but the average bars include them all. We see that both Manycore and Mosaic substantially improve the throughput of all functions. On average, Mosaic and Manycore increase the functions' throughput by 225% and 271%, respectively. However, the small cores of Manycore prevent it from running the monolithic applications efficiently. On average, Manycore reduces the throughput of monolithic applications by 68% over the Baseline. On the other hand, Mosaic delivers the same throughput as Baseline for these applications. Thus, providers can efficiently run both serverless and non-serverless workloads on Mosaic servers, improving cost-efficiency.

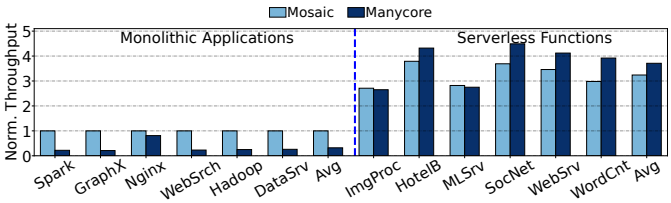
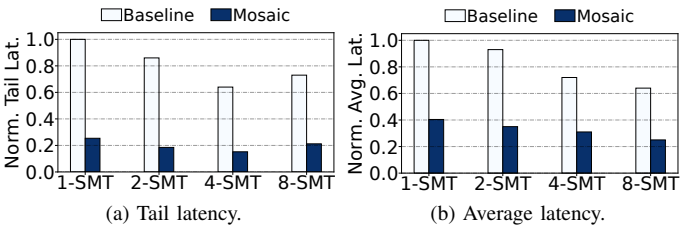


Fig. 17: Throughput of Mosaic and Manycore for monolithic applications and serverless functions normalized to Baseline.

6. Comparison to SMT. SMT cores can improve the throughput of serverless workloads. However, they (1) compromise security in public clouds due to side channel attacks, and (2) increase tail latency due to resource contention. Figure 18 shows the tail and average latency of SMT cores in Baseline and Mosaic averaged across all functions. The results are normalized to single-threaded Baseline. We see that, for both Baseline and Mosaic, 2- and 4-SMTs reduce the latencies due to shorter wait times. As we add more threads (8-way SMT), they compete for the shared resources and increase the tail latency. In all configurations, Mosaic significantly reduces the latency over Baseline.



(a) Tail latency. (b) Average latency.

Fig. 18: Latency of Baseline and Mosaic with different numbers of SMT threads normalized to single-threaded Baseline.

7. Comparison to MXFaaS. MXFaaS [74] is a software serverless platform that schedules concurrent invocations of the same function on a set of cores "owned" by the function. In addition, MXFaaS allows concurrent invocations of the same function to share the same container—which reduces the memory footprint and the cold-start latency. By binding functions to cores, however, MXFaaS may introduce load imbalance. Mosaic is orthogonal to MXFaaS and can be combined with it to further boost its performance. Figure 19 shows the tail latency of Baseline, MXFaaS, Mosaic, and MXFaaS+Mosaic. Compared to Baseline, MXFaaS and MXFaaS+Mosaic reduce the tail latency by 53.1% and 79.3%, respectively.

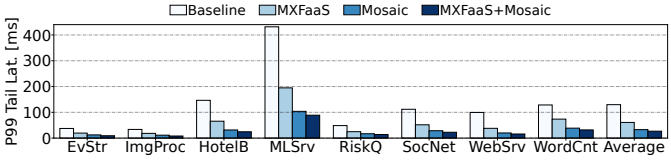


Fig. 19: Tail latency of Baseline, MXFaaS, Mosaic, and MXFaaS+Mosaic.

B. Average Power Consumption Reduction

Mosaic keeps the inactive chunks of the partitioned structures at a low voltage level, reducing power consumption. Figure 20 shows the average power consumed by each function in Mosaic relative to that in Baseline. We see that, across functions, Mosaic reduces the average power consumption over Baseline by 22%. Power reductions are higher for functions that require smaller tiles in the partitioned structures, such as *RiskQ*. Overall, Mosaic reduces both average response time and power consumption over the Baseline, resulting in an average 80% decrease in the *energy-delay product* of functions. For monolithic applications, Mosaic changes neither performance nor power consumption over Baseline.

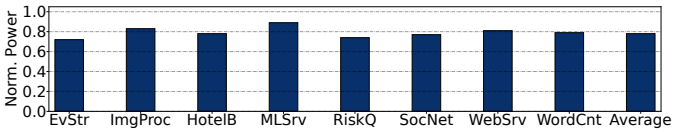


Fig. 20: Power usage of Mosaic normalized to Baseline.

C. Co-locating Serverless and Monolithic Workloads

We evaluate Mosaic when co-locating serverless functions with traditional monolithic applications [52] and allowing harvesting of cores assigned to monolithic applications. A monolithic application owns 8 cores, while serverless functions execute on the other 8 cores of the server and can steal more cores when they are idle. We run experiments with each of the 6 monolithic applications of Figure 17, and take the average across runs. Figure 21 shows the average and tail latency of the serverless functions when running with Mosaic normalized to when they run with Baseline. We see that Mosaic has lower latencies than Baseline even when functions are co-located with monolithic applications. In such an environment, Mosaic reduces the average and tail latencies by 49.8% and 67.8%, respectively.

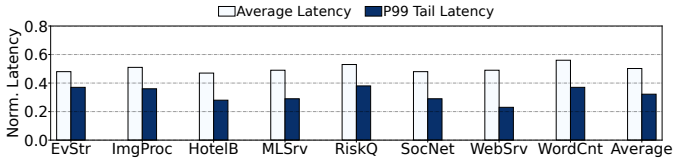


Fig. 21: Average and tail latency of functions when co-located with monolithic applications normalized to Baseline.

D. Cost Savings of Mosaic at Scale

To evaluate the cost savings of Mosaic at large scale, we model a datacenter with over a thousand servers running both serverless functions and regular (*i.e.*, non serverless) VMs. We use large open-source production traces [48] for the arrivals, departures, and resource requirements of functions [69] and VMs [17]. Based on the peak utilization, we provision servers using bin-packing. We evaluate three different datacenter designs based on the types of servers they have. *Baseline* uses only traditional servers for both workloads. *Baseline+Manycore* uses a traditional server pool for regular VMs and a separate Manycore pool for serverless workloads. *Mosaic* uses only Mosaic servers for both workloads.

Figure 22 shows the number of servers needed in the Baseline, Baseline+Manycore, and Mosaic datacenters while varying the fraction of total CPU hours used by serverless workloads. The number of servers is normalized to the number of servers in Baseline with a 0% fraction of CPU hours for serverless workloads. Note that, in the Baseline+Manycore design, the mix of traditional servers and Manycores is different at different X-axis points—*i.e.*, at different fractions of CPU hours for serverless workloads.

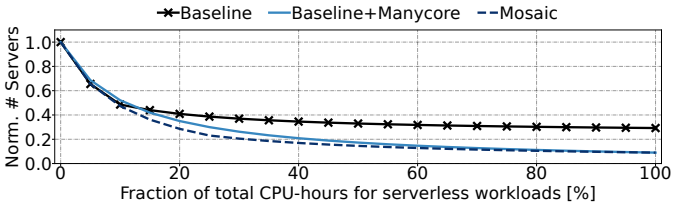


Fig. 22: Normalized number of servers in Baseline, Baseline+Manycore, and Mosaic datacenters while varying the fraction of total CPU-hours used by serverless workloads.

We see that, as the fraction of CPU hours devoted to serverless workloads increases, Baseline needs to provision more servers than the other two designs due to its lower throughput for serverless workloads. Baseline+Manycore provisions each server pool independently for each peak, which leaves some of the servers underutilized for certain periods due to fragmentation. Mosaic needs the lowest number of servers because its servers are optimized to execute both types of workloads. Overall, Mosaic reduces the number of servers needed by 10-24% over Baseline+Manycore for a range of fraction of CPU hours for serverless workloads. It never needs more servers than the other designs. Therefore, Mosaic has the lowest cost.

E. Sensitivity Studies

We conduct sensitivity studies to analyze the efficiency of Mosaic under various conditions.

1. Sensitivity to system load. We maintain the mix of functions in our workload and invoke the functions with Low, Medium, and High loads using a Poisson distribution. These loads correspond to attaining 25%, 50%, and 70% average CPU utilization as in prior work [74]. Each function is invoked with equal probability. Figure 23 shows the tail latency for each function when running on Mosaic with the three loads normalized to the tail latency when running on Baseline with the same load.

We see that, at all loads, Mosaic has a substantially lower tail latency than Baseline. At high loads, cores context switch more frequently and, therefore, as the load increases, the benefits of Mosaic over the Baseline are even higher. On average, Mosaic reduces the P99 tail latency over Baseline

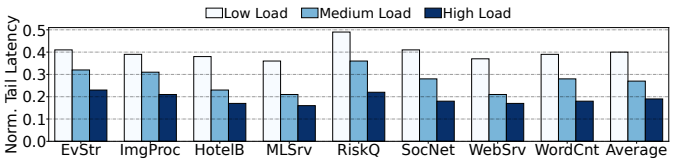


Fig. 23: Normalized tail latency of Mosaic over the Baseline.

by 59.8%, 72.7%, and 80.9% in Low, Medium and High load, respectively. We observe similar trends for average latency.

2. Sensitivity to core count. Serverless providers may want to reduce their operating cost by reducing the number of cores. We perform a sensitivity analysis to measure how the tail latency changes as the core count decreases. Figure 24 shows the tail latency of Mosaic with various core counts normalized to the tail latency of the 16-core Baseline. On average, Mosaic with 16, 12, 8, and 4 cores reduces the tail latency of the 16-core Baseline by 75%, 66%, 39%, and 4%, respectively. Thus, the provider can maintain the current response time of Baseline at 4× lower cost with Mosaic.

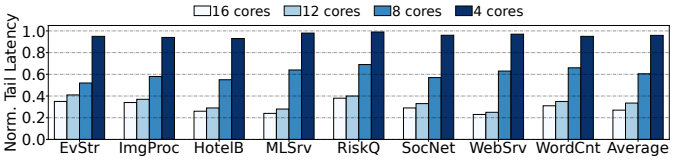
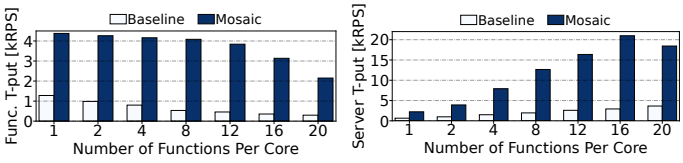


Fig. 24: Tail latency of Mosaic with different numbers of cores normalized to Baseline with 16 cores.

3. Sensitivity to core oversubscription. In this experiment, we vary the number of *different* functions that are scheduled in a round-robin manner to execute on a given core. We measure the highest sustainable load without SLO violations—*i.e.*, the throughput. With a higher number of different functions, the core oversubscription increases, and the state in the core structures in both Baseline and Mosaic gets polluted more.



(a) Per-function throughput. (b) Total server throughput.

Fig. 25: Throughput while varying core oversubscription.

Figure 25 shows the changes in throughput in *Baseline* and *Mosaic* as we increase the number of different functions scheduled on a core. Figure 25a shows the per-function throughput and Figure 25b the total server throughput. We see that, as the oversubscription increases, per-function throughput drops for both Mosaic and Baseline due to more state pollution. On the other hand, total server throughput increases as more functions are running in a server. In all cases, Mosaic delivers much higher throughput than Baseline. Even with 20 functions per core, Mosaic delivers a 68% higher per-function throughput than Baseline with a single function per core. Overall, Mosaic delivers a much higher total server throughput than Baseline.

4. Sensitivity to number of different functions. In all of our experiments before Section V-E, we run 8 different functions. Recall that we set-up the environment so that different instances of the same function do not share any instructions or data (Section IV), ensuring that there is no state reuse across them. Running the experiments with a higher number of different functions should not change the results if the rate of function invocation is the same. To prove it, we perform a new experiment with 64 different functions [4], [36], [87], [88] (as many as there are functions in the production traces) with the same total invocation rate. We observe that the results change very little. It can be shown that, with 64 different functions, Mosaic reduces the average and tail latency by 62.1% and 76.5%, respectively, over Baseline. With 8 different functions, Figure 14 showed that Mosaic reduces the average and tail latency by 59.6% and 74.6%, respectively. Hence, our methodology mimics environments with many functions, as in real settings.

F. MosaicScheduler Prediction Accuracy

We compare the tile sizes created by MosaicScheduler for the evaluated functions and the optimal tile sizes obtained via exhaustive search. Figure 26 shows the tile sizes created by MosaicScheduler in number of chunks for each function and hardware structure. We see that most of the functions use 1-2 chunks. It can be shown that these tile sizes closely follow the optimal ones. The only discrepancy occurs in HotelB and SocNet for the BTB, where MosaicScheduler creates a tile larger by one chunk. MosaicScheduler over-predicted the tile sizes in these two cases and never under-predicted them, ensuring no performance degradation.

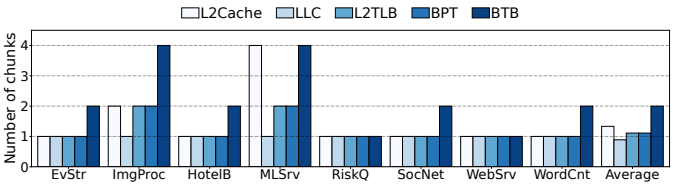


Fig. 26: Tile sizes created by MosaicScheduler.

Also, we measure the accuracy of our regression model that predicts the size of non-profiled functions. We collect metrics for 70 open-source functions [4], [15], [36], [76], [88], find their optimal configurations via exhaustive search, and create the dataset. We split the dataset into 80% train and 20% test. The resulting model produces accurate predictions. For most of the structures, such as the branch predictor table, the model predicts the optimal tile sizes with 100% accuracy. For some structures, such as the branch target buffer, the model can slightly overpredict the tile size. On average, the model achieves 92% accuracy. The compute requirements of the model are very low: a prediction takes a few hundred μs

and is done off the critical path. The model is queried only when a function is admitted to the cluster for the first time.

VI. Related Work

1. Partitioning schemes. Many researchers have explored resource partitioning of architectural resources so that applications meet quality of service (QoS) (*e.g.*, [13], [14], [40], [51], [53], [61], [90], [91]). In some cases, they consider concurrently-running SMT threads (*e.g.*, [20], [56], [77], [81]). PARTIES [14] tracks the tail latency of services and moves the resources (LLC, memory bandwidth, I/O, or cores) between services based on their deadlines. In these works, resources are managed with a feedback controller [14], Bayesian networks [61], multi-armed bandits [13] or ML techniques [40], [51], [90]. These schemes are efficient for long-running datacenter services and a relatively fixed mix of co-located services. Using them in serverless environments would not prevent a function from losing its state in a core on a context switch. With SMT proposals, one would sacrifice the security by allowing different functions to run concurrently on the same core. Mosaic targets more dynamic serverless environments with the goal of preserving the functions' micro-architectural state across context switches while keeping the security guarantees.

2. Micro-architecture prewarm. A few studies have explored the impact of serverless environments on the underlying hardware. Shahrad *et al.* [68] observed that cold-starts, containerization, and inter-function interference reduce the effectiveness of micro-architectural structures. Jukebox [63] uses on-chip metadata to prefetch instructions for functions that start with a cold micro-architectural state. In non-serverless environments, Ahn *et al.* [2] preserve a VM's context in the LLC on a cross-VM context switch. Mosaic preserves a function's micro-architectural state across context switches without increasing the on-chip area or memory bandwidth consumption.

3. Serverless optimizations. Optimizing serverless software stacks has received substantial attention (*e.g.*, [24], [43], [58], [69], [72], [74], [75], [78]). MXFaaS [74] improves performance by efficiently sharing a server's resources between concurrently executing same-function invocations. REAP [78] records a function's stable working set of guest memory pages and prefetches it from disk. SpecFaaS [75] accelerates the execution of multi-function serverless applications with software-supported speculative execution of functions. EcoFaaS [72] redesigns the serverless software stack to improve energy efficiency while maintaining high performance. Mosaic can further enhance the effectiveness of such systems. Researchers have also proposed using simple cores to host serverless workloads [71]. More advanced designs such as μManycore [73] potentially move the tipping point closer toward processor specialization for serverless environments.

VII. Conclusion

This paper presented a micro-architectural characterization of serverless environments and proposed Mosaic, an architecture optimized for serverless environments. Mosaic has two components: (1) *MosaicCPU*, a processor architecture that efficiently runs both serverless and traditional workloads, and (2) *MosaicScheduler*, a software stack for serverless systems that maximizes the benefits of MosaicCPU. MosaicCPU partitions micro-architectural structures into small chunks and assigns tiles of such chunks to functions. MosaicScheduler sizes the tiles for functions and schedules function invocations based on the state of the tiles. Compared to conventional server-class processors, Mosaic improves the throughput of serverless workloads by 225% while using 22% less power. Conversely, Mosaic achieves the performance of server-class processors with one quarter of the cores.

Acknowledgments

This work was supported in part by NSF under grants CNS 1956007, CCF 2107470 and CCF 2316233; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020.

[2] J. Ahn, C. H. Park, and J. Huh, "Micro-Sliced Virtual Processors to Hide the Effect of Discontinuous CPU Availability for Consolidated Systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '14)*, 2014.

[3] Amazon AWS, "AWS Lambda," April 2024. [Online]. Available: https://aws.amazon.com/lambda/

[4] ——, "AWS Samples: AWS Serverless Workshops," April 2024. [Online]. Available: https://github.com/aws-samples/aws-serverless-workshops/

[5] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the 2018 ACM Symposium on Cloud Computing (SoCC '18)*, 2018.

[6] Apache OpenWhisk, April 2024. [Online]. Available: https://openwhisk.apache.org/

[7] ARM, "ARM Cortex A15," April 2024. [Online]. Available: https://developer.arm.com/Processors/Cortex-A15

[8] ——, "Processing Architecture for Power Efficiency and Performance," April 2024. [Online]. Available: https://www.arm.com/technologies/big-little

[9] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO '17)*, 2017.

[10] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization," in *Proceedings of the IEEE 25th Computer Security Foundations Symposium (CSF '12)*, 2012.

[11] Brittany King, "Top use cases for serverless computing," April 2024. [Online]. Available: https://www.digitalocean.com/blog/top-use-cases-for-serverless-computing

[12] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip Redundant Paths to Make Serverless Fast," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, 2020.

[13] R. Chen, H. Shi, Y. Li, X. Liu, and G. Wang, "OLPart: Online Learning Based Resource Partitioning for Colocating Multiple Latency-Critical Jobs on Commodity Computers," in *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, 2023.

[14] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[15] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," in *Proceedings of the 22nd International Middleware Conference (Middleware '21)*, 2021.

[16] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling High Performance Serverless with RDMA and Leases," in *Proceedings of the 37th IEEE Interational Parallel and Distributed Processing Symposium (IPDPS '23)*, 2023.

[17] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[18] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "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)*, 2020.

[19] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A Review of Serverless Use Cases and their Characteristics," *CoRR*, vol. abs/2008.11110, 2020. [Online]. Available: https://arxiv.org/abs/2008.11110

[20] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-bandwidth aware thread allocation in multicore SMT processors," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*, 2013.

[21] Firecracker, "Firecracker design," 2022. [Online]. Available: https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md

[22] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*, 2002.

[23] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[24] A. Fuerst and P. Sharma, "FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[25] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[26] Google, "Google Cloud Functions," April 2024. [Online]. Available: https://cloud.google.com/functions

[27] IBM, "IBM Cloud Functions," April 2024. [Online]. Available: https://cloud.ibm.com/functions/

[28] Intel, "Intel® CAT: Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2016.

[29] ——, "Intel RDT Software Package," https://github.com/intel/intel-cmt-cat/tree/master, 2024.

[30] ——, "Intel® Xeon® Processors," April 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/processors/xeon.html

[31] ——, "Products formerly SAPPHIRE RAPIDS," April 2024. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/codename/126212/products-formerly-sapphire-rapids.html

[32] Z. Jia and E. Witchel, "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[33] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How Does It Function? Characterizing Long-Term Trends in Production Serverless Workloads," in *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*, 2023.

[34] W. Jordan, "The serverless server," 2022. [Online]. Available: https://fly.io/blog/the-serverless-server/

[35] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)*, 2022.

[36] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, 2019.

[37] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[38] Knative, April 2024. [Online]. Available: https://knative.dev/docs/

[39] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service Workflows," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[40] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[41] S. Kumar and A. Puthiyavettle, "Architecting a Highly Available Serverless, Microservices-Based Ecommerce Site," 2021. [Online]. Available: https://aws.amazon.com/blogs/architecture/architecting-a-highly-available-serverless-microservices-based-ecommerce-site/

[42] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*, 2009.

[43] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[44] J. Lindemann and M. Fischer, "A memory-deduplication side-channel attack to detect applications in co-resident virtual machines," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*, 2018.

[45] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, 2005.

[46] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[47] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows," in *Proceedings of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '22)*, 2022.

[48] Microsoft Azure, "Azure Public Dataset," April 2024. [Online]. Available: https://github.com/Azure/AzurePublicDataset

[49] ——, "Microsoft Azure Functions," April 2024. [Online]. Available: https://azure.microsoft.com/en-gb/services/functions/

[50] MongoDB, "Serverless Application Development," April 2024. [Online]. Available: https://www.mongodb.com/solutions/use-cases/serverless

[51] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.

[52] T. Palit, Y. Shen, and M. Ferdman, "Demystifying Cloud Benchmarking," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '16)*, 2016.

[53] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 2020.

[54] Python Scikit-learn, "MultiOutputClassifier," April 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.multioutput.MultiOutputClassifier.htmll

[55] ——, "RandomForestClassifier," April 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.htmll

[56] S. Raasch and S. Reinhardt, "The impact of resource partitioning on SMT processors," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, 2003.

[57] A. F. Rodrigues, J. Cook, E. Cooper-Balis, K. S. Hemmert, C. Kersey, R. Riesen, P. Rosenfeld, R. Oldfield, and M. Weston, "The Structural Simulation Toolkit," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '10)*, 2006.

[58] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*, 2021.

[59] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated Model-less Inference Serving," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*, 2021.

[60] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[61] R. B. Roy, T. Patel, and D. Tiwari, "SATORI: Efficient and Fair Resource Partitioning by Sacrificing Short-Term Benefits for Long-Term Gains," in *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[62] ——, "IceBreaker: Warming Serverless Functions Better with Heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[63] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm Serverless Functions: Characterization and Optimization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*, 2022.

[64] D. Schall, A. Sandberg, and B. Grot, "Warming Up a Cold Front-End with Ignite," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*, 2023.

[65] K. Seemakhupt, B. E. Stephens, S. Khan, S. Liu, H. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy, "A Cloud-Scale Characterization of Remote Procedure Calls," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, 2023.

[66] Serverless, "Use cases," April 2024. [Online]. Available: https://www.serverless.com/learn/use-cases/

[67] A. Seznec, "TAGE-SC-L Branch Predictors," in *JILP - Championship Branch Prediction*, Minneapolis, United States, Jun. 2014. [Online]. Available: https://inria.hal.science/hal-01086920

[68] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '19)*, 2019.

[69] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020.

[70] Software Freedom Conservancy, "QEMU: A generic and open source machine emulator and virtualizer," April 2024. [Online]. Available: https://www.qemu.org/

[71] R. Starc, T. Kuchler, M. Giardino, and A. Klimovic, "Serverless? RISC more!" in *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies (SESAME '24)*, 2024.

[72] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA '24)*, 2024.

[73] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "µManycore: A Cloud-Native CPU for Tail at Scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[74] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[75] ——, "SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*, 2023.

[76] The vHive Ecosystem, "vSwarm - Serverless Benchmarking Suite," April 2024. [Online]. Available: https://github.com/vhive-serverless/vSwarm

[77] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA '96)*, 1996.

[78] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "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)*, 2021.

[79] VMware vSphere, "Sharing Memory Across Virtual Machines," April 2024. [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-F9111E35-E197-46EC-8350-77827A5A2DEC.html#GUID-F9111E35-E197-46EC-8350-77827A5A2DEC

[80] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, 2020.

[81] H. Wang, I. Koren, and C. M. Krishna, "An adaptive resource partitioning algorithm for SMT processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, 2008.

[82] S. Wilson and D. Pickering, "A Guide to Developing Serverless Ecommerce Workflows for Commercetools with AWS Lambda," 2021. [Online]. Available: https://aws.amazon.com/blogs/industries/a-guide-to-developing-serverless-ecommerce-workflows-for-commercetools-with-aws-lambda/

[83] H. Xia, D. Zhang, W. Liu, I. Haller, B. Sherwin, and D. Chisnall, "A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy (SP '22)*, 2022.

[84] J. H. Yahya, H. Volos, D. B. Bartolini, G. Antoniou, J. S. Kim, Z. Wang, K. Kalaitzidis, T. Rollet, Z. Chen, Y. Geng, O. Mutlu, and Y. Sazeides, "AgileWatts: An Energy-Efficient CPU Core Idle-State Architecture for Latency-Sensitive Server Applications," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022.

[85] S.-H. Yang, M. Powell, B. Falsafi, and T. Vijaykumar, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *Proceedings of the Eighth International Symposium on High Performance Computer Architecture (HPCA '02)*, 2002.

[86] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: A Native Serverless System for Low-latency, High-throughput Inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[87] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*, 2023.

[88] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '20)*, 2020.

[89] Y. Zhang, I. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proceedings of the International Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[90] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.

[91] Y. Zhang, J. Chen, X. Jiang, Q. Liu, I. M. Steiner, A. J. Herdrich, K. Shu, R. Das, L. Cui, and L. Jiang, "LIBRA: Clearing the Cloud Through Dynamic Memory Bandwidth Management," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*, 2021.