Khepri: Crystallizing TAGE for Memory Efficient Prewarm in Serverless Computing

Zengshi Wang[1], Zhiyuan Zhang[1], Zhuoyuan Yang[1], Kanheng Jiang[1], Chao Fu[2]*, Jun Han[1]*
[1]State Key Laboratory of Integrated Chips and Systems, Fudan University, Shanghai, China.
[2]Shao-Chips Laboratory, Shaoxing, China.
{zswang23, zhiyuanzhang22, zhuoyuanyang22, khjiang22}@m.fudan.edu.cn {cfu19, junhan}@fudan.edu.cn

***Abstract*—As an increasingly popular cloud computing model, serverless computing suffers from performance degradation caused by microarchitectural cold start. Previous studies identify the front-end as the bottleneck and explore solutions such as instruction prefetching and restoring Branch Target Buffer. However, they fail to prewarm the Conditional Branch Predictor (CBP), because the large size of its core component, the TAgged GEometric history length predictor (TAGE), makes it impractical to be saved and restored.**

This paper observes the predictive sparsity of TAGE, where only a small subset of entries can dominate the predictor's coverage and accuracy. We introduce Khepri, a memory efficient CBP prewarming mechanism that uses a TAGE Crystallization algorithm to identify these dominant entries. Khepri records them in main memory and restores them to prewarm TAGE at the next invocation. Khepri achieves a 1.57× speedup over the baseline and outperforms the state-of-the-art technique by 14%, requiring only 1.54KB in main memory on average.

***Index Terms*—Cloud Computing, Serverless Computing, Cold Start, Branch Predictor, TAGE**

I. Introduction

Serverless computing has emerged as a novel and increasingly popular paradigm of cloud computing [1, 16, 25, 34]. Unlike traditional cloud computing models, serverless computing provides developers with containerized, ready-to-use runtime environments [2, 33]. Developers simply upload stateless function code and the cloud provider automatically provisions the necessary computing resources for each function and charges based solely on actual usage.

In serverless computing, thousands of serverless functions share hardware resources and execute in an interleaved fashion, and both software contexts and microarchitectural state are inevitably flushed between two invocations of the same function [4, 22]. Thus, each time a function is invoked, it must endure a process known as **cold start**, where its runtime state needs rebuilt, introducing substantial performance degradation.

Previous studies on mitigating microarchitectural cold start focus on the processor front-end, as it has been identified as major bottleneck [6, 7]. JukeBox prefetches instructions into the L2 cache for serverless functions [6]. Ignite additionally restores Branch Target Buffer (BTB) state [7]. However, these techniques exhibit a significant performance gap compared to back-to-back execution, which we refer to as *warm execution*. As shown in Fig. 1(b), there is a speedup gap up to 26% between state-of-the-art solution and warm execution. A fully

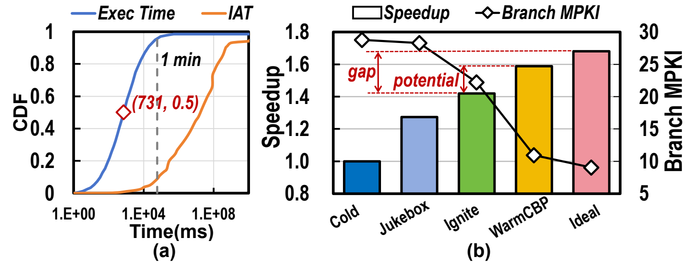

Figure 1. (a) Cumulative Distribution Function of execution time and Inter-Arrival Time of serverless functions in Microsoft Azure. (b) Performance gap of serverless functions.

prewarmed Conditional Branch Predictor (CBP) improves performance by 18%, which closes most of that gap, underscoring its critical role in mitigating microarchitectural cold start.

The main **challenge** to prewarm CBP is its large memory overhead to save and restore. As the de facto standard component in server-grade processors, the TAgged GEometric history length predictor (TAGE) serves as the core of CBP, with its size reaching up to 64KB [3, 35]. Thus, saving its entire state incurs significant memory overhead. Restoring such a large predictor state is time-consuming and creates more contention for TAGE read/write ports and memory bandwidth, which could undermine the benefit of prewarming.

To address this challenge, we profile TAGE under serverless workloads. Our analysis identifies the ***predictive sparsity*** of TAGE, where only a small subset of entries determines the overall prediction coverage and accuracy, coexisting with the remaining invalid or training entries. Therefore, these decisive entries can be *crystallized* from TAGE for prewarming, which can restore most of its predictive effectiveness.

Motivated by this observation, we propose *Khepri*, a memory efficient CBP prewarming mechanism to mitigate microarchitectural cold start for serverless functions. Khepri employs a TAGE Crystallization algorithm to identify dominant entries and stores them in main memory as Khepri metadata. Khepri metadata are restored to TAGE upon the next invocation. Khepri outperforms the state-of-the-art by 14% as it eliminates most branch mispredictions caused by cold start. Khepri metadata occupies less than 2KB in main memory per function on average, significantly smaller than the TAGE size. The area overhead of Khepri is merely 3.5KB, which is feasible to integrate into server-grade processors.

*co-corresponding authors.

To summarize our **contributions**:

- We demonstrate that prewarming CBP in serverless computing has the potential to improve performance by 20%.
- We are the first to identify the predictive sparsity of TAGE, where a small subset of entries dominates the overall prediction coverage and accuracy. We propose a TAGE Crystallization algorithm to identify these entries.
- We introduce Khepri, a CBP prewarm mechanism for serverless computing. It improves performance by 57% over the baseline and 14% over the state-of-the-art, with a minimal memory usage and reasonable area overhead.

II. Background and Motivation

A. Microarchitectural Cold Start in Serverless Computing

In the serverless programming model, applications are composed of several stateless and event-triggered functions. When a function is invoked, computing resources are allocated on demand, and users are charged based on actual CPU time and memory usage [33].

The serverless computing model is especially attractive for infrequent and lightweight applications, as it supports automatic scaling and provides cost efficiency in such scenarios [34]. However, the characteristic of these functions makes them susceptible to cold starts. Our analysis of serverless function traces from Microsoft Azure [5] reveals that 90% of serverless functions have inter-arrival times exceeding one minute, whereas their median execution time is only around 700 ms, as shown in Fig. 1(a). It results in hundreds of other function invocations executing interleaved between two invocations of the same function [16], which flushes software context and microarchitectural state, making cold starts the main source of performance degradation in serverless computing [6, 7].

Previous studies have identified the front-end as performance bottleneck during cold start, since Frontend Bound and Bad Speculation events collectively account for 58% of execution time on average according to Top-Down analysis [8]. JukeBox is an instruction prefetcher tailored for serverless computing [6]. It leverages the similarity of instruction footprints across invocations of the same function to prefetch instructions. Ignite introduces a mechanism to generate instruction prefetches and restore the state of the BTB by monitoring BTB insertions [7]. It reduces both I-Cache misses and BTB misses.

As the state-of-the-art solution, Ignite leaves an unexploited performance potential of 26% compared to warm execution, as shown in Fig. 1(b). This gap arises from the cold start of the CBP, where pipeline flushes caused by mispredicted branch directions degrade performance. A fully warmed-up CBP can close most of the gap as it reduces branch mispredictions per kilo-instructions (Branch MPKI) to that comparable to warm execution, highlighting its critical role in serverless computing.

However, it is a great challenge to prewarm CBP due to its large memory overhead. Modern high-performance processors employ increasingly complex predictors due to the rising branch misprediction penalty in deep pipelines [3, 9, 18, 19]. As the core component of CBP, the size of TAGE has reached 64KB. The substantial memory overhead and bandwidth contention to restore CBP state may negate its benefits.

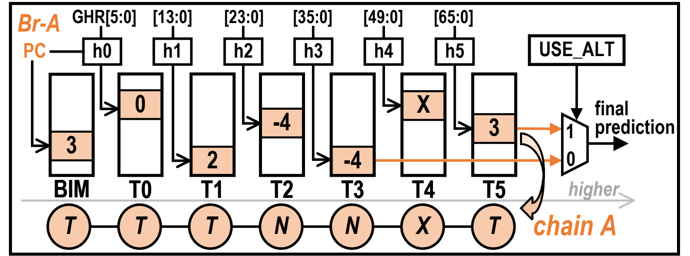

Figure 2. Structure of TAGE and prediction chain. Branch A indexes multiple prediction tables and generates a prediction chain, in which T is taken, N is not-taken and X is invalid.

B. Conditional Branch Predictor

CBP is used to predict the direction of conditional branches in modern high-performance processors [20, 21, 26]. A typical implementation consists of a Global History Register (GHR), a bimodal predictor (BIM), and a TAGE predictor.

GHR records the branch histories. BIM uses the program counter (PC) to index a counter whose most significant bit indicates prediction direction. TAGE is the core component in CBP. It comprises multiple tables indexed by the hash of branch histories of varying lengths and PC, as shown in Fig. 2. Each TAGE entry contains a *tag* to identify the branch, a *confidence* counter to indicate predicted direction and strength, and a *useful bit* for replacement. In this paper, we refer to the table indexed by the longer history as the *higher* table.

During the prediction phase, a branch will index an entry in BIM and each TAGE table, which is highlighted in orange in Fig. 2. We define the set of these entries as a *prediction chain*. The highest and second-highest valid entries in the prediction chain serve as candidates, which are named provider and alternative, respectively. The final predicted direction is selected between the two candidates by USE_ALT counter.

During the update phase, the confidence counter of the entry that selected for final prediction is updated towards the correct direction. If the provider and alternative predict the same and correct direction, the useful bit of provider will be reduced, making it more likely to be replaced in the future. When the provider mispredicts, a new entry will be allocated in a higher table. The USE_ALT counter is also updated to favor the candidate that has historically been more accurate.

C. Predictive Sparsity of TAGE

We are the first to identify the *predictive sparsity* of TAGE, where a small subset of entries dominate the predictor's overall coverage and accuracy. The sparsity arises from two factors. First, TAGE contains a mix of invalid, training, and well-trained entries at any given moment, but only the well-trained ones contribute reliable predictions. Second, some of the well-trained entries may be redundant because they can be functionally replaced by others along the prediction chain. We support these findings with both theoretical analysis and empirical evidence.

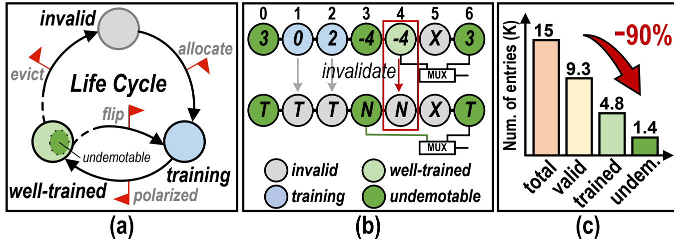

Figure 3. Predictive Sparsity of TAGE. (a) Lifecycle of TAGE entry. (b) Case Study of redundancy along prediction chain. (c) TAGE state breakdown.

*1) **Lifecycle Breakdown**:* We divide the lifecycle of each TAGE entry into three stages: *invalid*, *training*, and *well-trained*, as illustrated in Fig. 3(a). At the beginning of its lifecycle, a TAGE entry is allocated for a branch and transitions from *invalid* state to *training* state. If it continues to make correct predictions, the confidence counter is updated until saturation, at which point the entry is considered *polarized*. The TAGE entry enters the *well-trained* state when it is polarized, as it can consistently provide reliable prediction results. Confidence counter of *well-trained* entry drops if it produces incorrect predictions. It transitions to the *invalid* state when it is eventually evicted, or returns to the *training* state if its predicted direction flips. Although polarized entries account for only 32% of the total TAGE capacity, they contribute 90% of the prediction coverage for conditional branches with an accuracy of up to 99.2% under serverless applications.

*2) **Redundancy along Prediction Chain**:* Even among *well-trained* entries, some are still redundant along the prediction chain. We define a polarized entry as *demotable* if its removal do not change the final predicted direction. An entry is demotable if its predicted direction matches that of the adjacent lower valid entry on the prediction chain. When this entry is disabled, the neighbor can replace it to predict the same direction, thus preserving the final predicted direction of the chain. For example, the final predicted direction of the prediction chain in Fig. 3(b) is selected between Node4 and Node6. Node4 is demotable because its adjacent valid entry, Node3, also predicts not-taken. Invalidating Node4 will shift the prediction candidates from (Node4, Node6) to (Node3, Node6), resulting in the same predicted direction of the chain. Therefore, Node4 is redundant on this chain during prediction.

We take a snapshot of the TAGE during the execution of serverless functions and analyze the distribution of entries across different states, as shown in Fig. 3(c). We use the same architecture configuration and serverless functions as in Section IV. Among the 15K TAGE entries, 62% are valid as they are allocated for branch instructions in the current function, but only 32% are *well-trained*. Furthermore, after removing redundant entries, only 1.4K entries are found to be undemotable for prediction, comprising just 10% of the total size. Our experiments verify the predictive sparsity of TAGE under serverless workloads, revealing huge potential for CBP state space compression.

III. Design

We propose Khepri, a memory efficient mechanism for pre-warming CBP in serverless computing. As illustrated in Fig. 4, Khepri consists of a *Crystallizer*, a *Recorder*, a *Replayer* and *Khepri Registers*, and operates in three stages: *Mark*, *Record*, and *Replay*.

During the Mark stage, Crystallizer applies a TAGE Crystallization algorithm to identify prediction-dominant entries and set their extended mark bit. During the Record stage, Recorder scans all TAGE predictor tables in parallel, collects and encodes marked entries into Khepri entries and stores them in main memory as Khepri metadata. During the Replay stage, Replayer loads Khepri metadata from memory, decodes and restores them into TAGE.

Within the system, Khepri is directly controlled by the Khepri Registers. It interfaces with the operating system (OS), which assists in managing memory and coordinating workflow by configuring the Khepri Registers.

A. Mark

When a serverless function is invoked, Crystallizer is enabled and Khepri enters the Mark stage. As a branch instruction accesses the multiple branch predictor tables, it generates a prediction chain. Crystallizer analyzes every prediction chain and marks entries to be recorded and replayed ❶ , as illustrated in Fig. 4. When a marked entry is evicted or its predicted direction is reversed, its mark bit will be cleared.

In order to identify the smallest subset of TAGE entries with trivial loss to coverage and accuracy, we propose the TAGE Crystallization algorithm as shown in Algorithm 1. In the algorithm, we evaluate the priority of each TAGE entry to be marked from both intra-chain and inter-chain perspectives.

*1) **intra-chain priority**:* Undemotable entries has higher intra-chain priority. According to Section II-C, undemotable entries dominate prediction within each prediction chain and thus should be prioritized for marking. TAGE Crystallization algorithm first filters polarized entries and records their predicted directions, as *lines 1–8* in Algorithm 1. It then collects undemotable entries by detecting direction flips on the chain (*lines 9–14*). Finally, the top two undemotable entries are selected to mark, as only the highest two entries in a chain are considered as candidates for final prediciton, which is discussed in Section II-B (*lines 15–18*).

*2) **inter-chain priority**:* Entries that are selected more frequently have higher inter-chain priority. *Mark conflicts* may occur due to the node sharing among prediction chains caused by instruction's repeat execution and hash conflicts. For instance, if a node was previously marked but current chain selects another node with identical predicted direction, we need to compare their priorities and retain only one of them. Our principle is to prioritize retaining the node selected most frequently. A direct implementation is to extend each TAGE entry with a counter that tracks marking occurrences, but this incurs significant hardware overhead. Instead, we employ a probabilistic replacement policy. When mark conflicts occur, the existing mark bit is cleared with a defined probability

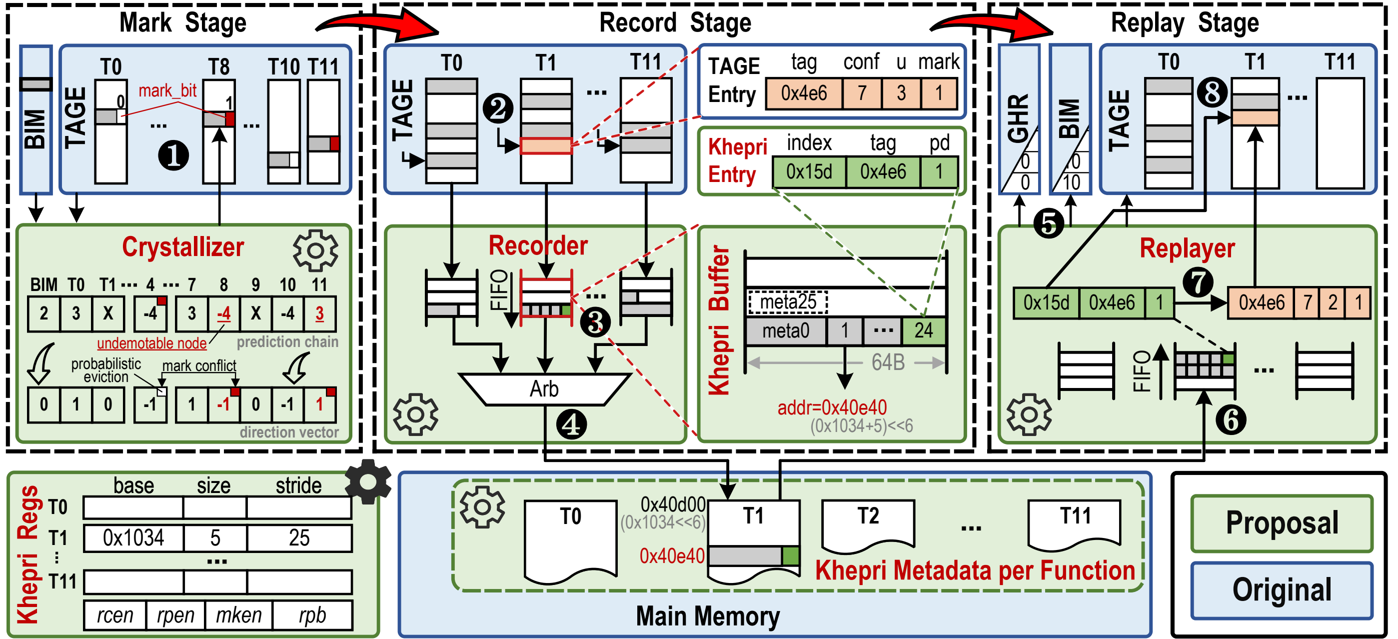

Figure 4. Overview of Khepri. Modules introduced by Khepri are highlighted with green shading.

(*lines 20-22*). This approach approximately prioritizes more frequently selected entries to hold the mark flag.

As exemplified in Fig. 4, when a branch instruction accesses CBP, Crystallizer collects the prediction chain and generates direction vector for polarized entries. Node8 and Node11 are identified as the top two undemotable entries and marked with red font and underline in the figure. With a probability defined by the *rpb* field in the khepri register, the mark bit of Node4 is cleared and the mark bit of Node8 is set.

B. Record

At the end of invocation, Khepri enters the Record stage, in which the Recorder module collects the marked entries, encodes and stores them in main memory.

Recorder concurrently scans all TAGE tables to collect the marked TAGE entries and encodes them into Khepri entries ❷. Each Khepri entry contains the branch *tag* from the TAGE entry, 1-bit wide predicted direction (*pd*), and *index* of the TAGE entry. Since different TAGE tables vary in size and tag width, the corresponding Khepri entries have different widths, which are tracked in the Khepri Registers as *stride*. After being scanned, the mark bits of entries are cleared to prevent interference cross different serverless functions.

To reduce the number of memory accesses, Khepri entries are first gathered in a 64B-wide *Khepri Buffer* ❸. Once the accumulate size reaches 64 bytes, a memory write request is issued with the target address calculated by the *base* address and metadata *size* from Khepri Registers. Since the size of Khepri entry may not perfectly fit the 64B buffer width, unaligned leftover space is discarded to avoid hardware complexity to split and pack entries across buffer boundaries. Recorder arbitrates concurrent memory requests from multiple buffers, giving higher priority to lower tables as they are more frequently accessed ❹.

Algorithm 1: TAGE Crystallization

Input: `chain`: prediction chain.
ϵ: eviction probability.
Output: Mark selected entries in `chain`.

```
   // Identify polarized nodes. Node0 is BIM.
1  directions[0] ← (chain[0].conf ≥ 2);
2  for i ← 1 to (|chain| − 1) do
3      if chain[i].conf = −4 then
4          directions[i] ← −1;
5      else if chain[i].conf = 3 then
6          directions[i] ← 1;
7      else
8          directions[i] ← 0;
   // Collect undemotable node indexes.
9  undemNodes ← empty;
10 lastDirection ← 1;
11 for i ← 1 to (|chain| − 1) do
12     if directions[i] × lastDirection = −1 then
13         undemNodes.push(i);
14         lastDirection ← directions[i];
   // Mark the highest two undem. nodes.
15 for i ← 0 to 1 do
16     if undemNodes.isEmpty() then
17         break;
18     Mark (undemNodes.pop());
19 Function Mark (idx)
20     if ¬chain.hasConflict(idx) or Rand(0, 1) < ϵ then
21         chain.clearConflict(idx);
22         chain.mark(idx);
```

C. Replay

At the beginning of invocation, Khepri enters the Replay stage, in which the Replayer module restores Khepri metadata of the function from main memory into TAGE.

Replayer first reset BIM to weakly taken and flush GHR

in replay stage ❺. Then Khepri metadata is load from main memory and cached in Khepri Buffers ❻. Replayer segments the metadata into Khepri entries based on the *stride* and decodes them into TAGE entries ❼. During restoration, the confidence of each TAGE entry is set to saturated, the useful bit is initialized as 2'b10, and the mark bit is set ❽.

D. System Support

Khepri is controlled by the Khepri Registers and the OS. Khepri Registers are a set of dedicated registers that hold enable signals for each module and metadata address information. The *base*, *size*, and *stride* fields in Khepri Registers define the memory region for each prediction table's Khepri metadata, corresponding to the start address, total size, and the size per Khepri entry. Additional configuration parameters, such as replacement probability (*rpb*), are also stored.

Khepri integrates with the OS to manage memory and control its workflow by configuring the Khepri Registers. The OS allocates memory for the Khepri metadata and binds addresses of these memory regions to the serverless function instance by recording them in the *task_struct* in Linux. At the end of an invocation, the OS activates the Recorder to store metadata in main memory by setting the *rcen* bit in Khepri Registers. The size of the saved metadata is tracked by Khepri Registers and finally updated to the *task_struct*. When a serverless function is assigned to a core, the OS loads the metadata addresses from *task_struct* to initialize Khepri Registers and enables the Replayer by setting the *rpen* bit. Simultaneously, the Crystallizer is activated via the *mken* bit.

IV. Methodology

A. Serverless Workload

We evaluate Khepri using serverless functions in vSwarm benchmark suite [14, 15], as listed in Table I. These functions are drawn from real-world serverless applications such as hotel reservation and online shopping. Several of the functions are available in multiple programming languages, including Python, Nodejs, and Go. The experiments are conducted on Ubuntu 20.04 with Docker 20.10 as the container runtime. During testing, each function is invoked with randomized inputs and is warmed up with 1000 invocations before measurement.

B. Simulator Infrastructure

We use gem5 v23.0, a widely used full-cycle simulator, to model a serverless computing system [10, 11]. A two-node system is constructed, where an atomic cpu acts as the client and an IceLake–like CPU acts as the server [13], with the two node connected via built-in Ethernet model. The configuration details of the server CPU are summarized in Table II. Serverless function are deployed in containers on the server, and are invoked through gRPC [31, 32] by the client. To simulate the microarchitectural cold start caused by interleaved serverless workloads, we flush caches and randomize CPU state of the server between consecutive invocations of the same function.

We evaluate the following related techniques:

TABLE I: Serverless Functions and Abbreviations

Function	Abbr.	Function	Abbr.
Hotel Reservation [23]		**Online Shopping [27]**	
Geo	Geo	Currency	Curr
Profile	Profile	EmailService	Email
Rate	Rate	Reservation	Resv
Recommend	Recd	ProductCatalog	Prod
User	User	Shipping	Ship
Others [28–30]			
Authentication	Auth-P/N/G	Fibonacci	Fib-P/G
AES	AES-P/N/G		

TABLE II: Configurations of Simulated System

Architecture	IceLake-like, x86-64, 2.6 GHz
Front-end	Fetch Bandwidth: 16 B/cycle L-TAGE: 12 Table, 11K entries Bimodal: 4K entries BTB: 12K entries, 6-way
Back-end	ROB: 352 entries LSQ: 128 load + 72 store RF: 280 Int + 224 FP
Memory Hierarchy	L1-I: 32KB, 8-way, 1 cycle L1-D: 48KB, 12-way, 4 cycles L2: 1280KB, 20-way, 13 cycles LLC: 8MB, 16-way, 50 cycles Main Memory: DDR4, 2400MHz

1) Baseline: We use gem5's decoupled front-end CPU model. A fetch-directed instruction prefetcher and a stride data prefetcher are enabled [12].

2) Jukebox: Use a 16-entry Code Region Buffer with a 1KB region size, and instructions are prefetched to L2 [6].

3) Ignite: BTB entries are compressed by encoding target addresses as 7-bit wide offsets [7].

4) Khepri: Crystallizer uses a 0.3 eviction probability. Each TAGE table is configured with a 64B-wide, 2-entries deep Khepri Buffer. The metadata size per table is limited to 4KB.

5) Ideal: Functions are executed in a back-to-back invocation mode, which avoids interleaved functions flush cache and other microarchitecture state.

V. Evaluation

A. Performance Analysis

We evaluate the performance of Baseline, Jukebox, Ignite, and Khepri under serverless functions. Fig. 5(a) shows the normalized speedup over Baseline. Jukebox mitigates instruction fetch stalls by prefetching. It improves performance by 28.3% on average. Ignite prefetches instructions and restores BTB. It achieves an average speedup of 1.43×. Building on Ignite, Khepri further restores the state of TAGE, reducing pipeline flushes caused by branch mispredictions. Khepri delivers a 20%-100% improvement in execution speed with an average of 57.2%, and outperforms Ignite by 14%.

The performance improvement of Khepri benefits from the reduction in Branch MPKI, which has two sources: wrong

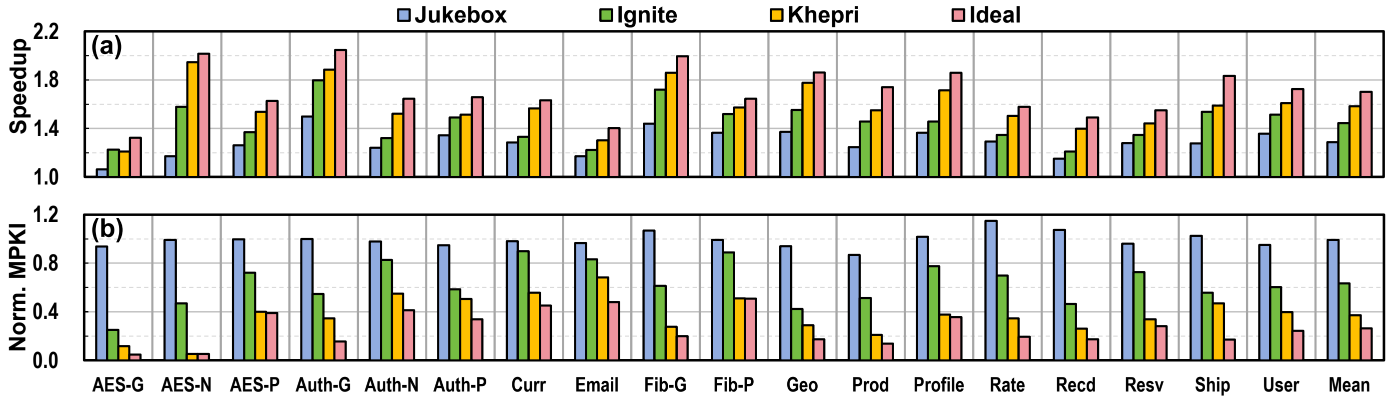

Figure 5. (a) Speedup over Baseline. (b) Normalized Branch MPKI (Baseline = 1.0).

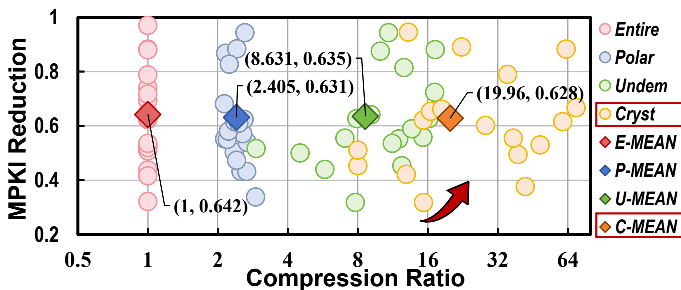

Figure 6. Ablation study of TAGE Crystallization.

branch target and mispredicted branch direction. As shown in Fig. 5(b), Ignite reduces branch MPKI by about 40%, since the restored BTB accurately predicts the target addresses for the most branch instructions. Khepri restores TAGE predictor and thus makes accurate branch direction prediction, reducing branch mispredictions by approximately 63% and closing most of the performance gap between the Baseline and Ideal.

B. Memory Usage

We conduct an ablation study for the TAGE Crystallization algorithm. Built upon the record-replay mechanism of Khepri, we employ four approaches to mark TAGE entries: Khepri-Entire marks all TAGE entries. Khepri-Polarized marks all polarized entries allocated for the current serverless function, and Khepri-Undemotable marks only undemotable entries, as discussed in Section II-C. Khepri-Crystallization use the TAGE Crystallization algorithm proposed in Section III-A.

We evaluate these approaches in terms of Branch MPKI reduction and compression ratio with the same experimental setup as in Section IV. Each scatter point in Fig. 6 represents the result of a serverless workload using one of the evaluated approaches. The compression ratio is calculated as the TAGE size (i.e. 30.69KB) divided by the memory usage of each approach. As illustrated in Fig. 6, Khepri-Entire achieves the highest performance, as it fully restores the complete TAGE state. Khepri-Polarized achieves an average compression ratio of 2.41 and reduces Branch MPKI by 63.1%. Compared to Khepri-Entire, Khepri-Polarized yields smaller performance gains, because it omits some useful entries for prediction and disrupts the training process. By eliminating redundant entries in the prediction chain, Khepri-Undemotable improves the compression ratio to 8.63. Khepri-Crystallization detects mark conflicts and employs a probabilistic eviction policy, which further raises the compression ratio to 19.96, corresponding to an average per-function memory usage of 1.54KB, while introducing negligible performance loss.

TABLE III: Hardware Overhead of Khepri

Component	Num.	Size	Sum.
Mark bit	15360	1 bits	1920 B
Khepri Register	12	64 bits	96 B
Khepri Buffer	12×2	64 B	1536 B
Total	-	-	**3552 B**

C. Hardware Overhead

The hardware overhead of Khepri is summarized in Table III. Khepri introduces an additional area of 3.5 KB, which accounts for approximately 10% of the total CBP.

The overhead primarily comes from the mark bits in TAGE, the Khepri Buffer, and the Khepri Register. Each TAGE entry is extended by a 1-bit mark flag, which collectively incurs an additional 1920B of SRAM storage. Each TAGE prediction table is configured with a 2-depth 64B-width Khepri Buffer, and a Khepri Register that stores its metadata's start address (48 b), total size (12 b), entry size (4 b), as described in Section III-D.

VI. Conclusion

Microarchitectural cold start has caused severe performance degradation in serverless computing, while prewarming branch predictors remains challenging due to its significant memory overhead. We reveal the predictive sparsity of TAGE, where a small subset of entries can dominate the predictor's overall coverage and accuracy. Leveraging this property, we propose Khepri as a memory efficient CBP prewarming mechanism. A detailed evaluation shows that Khepri outperforms state-of-the-art prewarming techniques, with a minimal memory usage and acceptable hardware overhead.

REFERENCES

[1] A. Byrne, Y. Pang, A. Zou, S. Nadgowda, and A. K. Coskun, "MicroFaaS: Energy-efficient Serverless on Bare-metal Single-board Computers," in 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Mar. 2022, pp. 754–759.

[2] M. Bacis, R. Brondolin, and M. D. Santambrogio, "BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing," in 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Mar. 2020, pp. 852–857.

[3] André Seznec. 2007. A 256 kbits l-tage branch predictor. Journal of Instruction Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2) 9 (2007), 1–6.

[4] Alibaba Cluster Trace Program. [Online]. Available: https://github.com/alibaba/clusterdata.

[5] Microsoft. Azure Functions. [Online]. Available: https://azure.microsoft.com/en-us/services/functions

[6] 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, New York New York: ACM, Jun. 2022, pp. 757–770.

[7] D. Schall, A. Sandberg, and B. Grot, "Warming Up a Cold Front-End with Ignite," in 56th Annual IEEE/ACM International Symposium on Microarchitecture, Toronto ON Canada: ACM, Oct. 2023, pp. 254–267.

[8] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), CA, USA: IEEE, Mar. 2014, pp. 35–44.

[9] D. Schall, A. Sandberg, and B. Grot, 'The Last-Level Branch Predictor', in 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2024, pp. 464–479.

[10] J. Lowe-Power et al., "The gem5 Simulator: Version 20.0+," Oct. 01, 2020, arXiv: arXiv:2007.03152.

[11] N. Binkert et al., "The gem5 simulator," SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1–7, May 2011.

[12] G. Reinman, B. Calder and T. Austin, "Fetch directed instruction prefetching," MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, Haifa, Israel, 1999, pp. 16-27.

[13] I. E. Papazian, "New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP)," 2020 IEEE Hot Chips 32 Symposium (HCS), Palo Alto, CA, USA, 2020, pp. 1-22.

[14] EASE Lab. 2022. vSwarm: A suite of representative serverless cloud-agnostic (dockerized) benchmarks. [Online]. Available: https://github.com/ease-lab/vSwarm.

[15] EASE Lab. 2022. vSwarm-u: Microarchitecture for serverless workloads. [Online]. Available: https://github.com/ease-lab/vSwarm-u.

[16] M. Shahrad et al., "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," the USENIX Annual Technical Conference (USENIX ATC), 2020.

[17] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A Metadata-Free Architecture for Control Flow Delivery," in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb. 2017, pp. 493–504.

[18] Chit-Kwan Lin and Stephen J. Tarsa, "Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions," 2019 IEEE International Symposium on Workload Characterization (IISWC), Orlando, FL, USA, 2019, pp. 228-238.

[19] S. S. Brown, J. Asher, and W. H. Mangione-Smith, "Offline program re-mapping to improve branch prediction efficiency in embedded systems," in Proceedings 2000. Design Automation Conference., Jan. 2000, pp. 111–116.

[20] A. Faravelon, N. Fournel, and F. Pétrot, "Fast and accurate branch predictor simulation," in 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), Mar. 2015, pp. 317–320.

[21] M. Tan, X. Liu, Z. Xie, D. Tong, and X. Cheng, "Energy-efficient branch prediction with Compiler-guided History Stack," in 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE), Mar. 2012, pp. 449–454.

[22] Shutian Luo et al. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '21). Association for Computing Machinery, New York, NY, USA, 412–426.

[23] Y. Gan et al., "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems," in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Providence RI USA: ACM, Apr. 2019, pp. 3–18.

[24] G. Ayers et al., "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers," in Proceedings of the 46th International Symposium on Computer Architecture, Phoenix Arizona: ACM, Jun. 2019, pp. 462–473.

[25] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless Computing: State-of-the-Art, Challenges and Opportunities," IEEE Transactions on Services Computing, vol. 16, no. 2, pp. 1522–1539, Mar. 2023.

[26] Y. Wang, H. Fan, S. Li, T. Liang, and W. Zhang, "A Modular Branch Predictor Performance Analysis Framework for Fast Design Space Exploration," in 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), Mar. 2024, pp. 1–6.

[27] GoogleCloudPlatform. 2022. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[28] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 2019, pp. 502-504.

[29] J. Kim and K. Lee, "Practical Cloud Workloads for Serverless FaaS," in Proceedings of the ACM Symposium on Cloud Computing, Santa Cruz CA USA: ACM, Nov. 2019, pp. 477–477.

[30] Amazon Web Services. 2022. Use API Gateway Lambda Authorizers. [Online]. Available: https://docs.aws.amazon.com/apigateway/

[31] 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, Virtual USA: ACM, Apr. 2021, pp. 559–572.

[32] RPC Authors. 2022. gRPC: A High-Performance, Open Source Universal RPC Framework. [Online]. Available: https://grpc.io

[33] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side web applications," in Proceedings of the 48th International Symposium on Microarchitecture, Waikiki Hawaii: ACM, Dec. 2015, pp. 762–774.

[34] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing Tail Latency in Serverless Clouds with STeLLAR," in 2021 IEEE International Symposium on Workload Characterization (IISWC), Nov. 2021, pp. 51–62.

[35] S. Lee, I. Kim, and L. Choi, "Branch predictor design and performance estimation for a high performance embedded microprocessor," in Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference, 2003., Jan. 2003, pp. 519–522.