RESEARCH-ARTICLE

Leveraging control-flow similarity to reduce branch predictor cold effects in microservices

HARIS VOLOS, University of Cyprus, Nicosia, Cyprus

STYLIANOS VASSILIOU, University of Cyprus, Nicosia, Cyprus

GEORGIA ANTONIOU, University of Cyprus, Nicosia, Cyprus

DAVIDE BASILIO BARTOLINI, Huawei Technologies Co., Ltd., Shenzhen, Guangdong, China

YIANNAKIS THRASOU SAZEIDES, University of Cyprus, Nicosia, Cyprus

Open Access Support provided by:

University of Cyprus

Huawei Technologies Co., Ltd.

Published: 21 June 2025

Citation in BibTeX format

ISCA '25: Proceedings of the 52nd Annual International Symposium on Computer Architecture
June 21 - 25, 2025
Tokyo, Japan

Conference Sponsors:
SIGARCH

https://doi.org/10.1145/3695053.3731059
ISBN: 9798400712616

Leveraging control-flow similarity to reduce branch predictor cold effects in microservices

Haris Volos
University of Cyprus
Nicosia, Cyprus
hvolos01@ucy.ac.cy

Stylianos Vassiliou
University of Cyprus
Nicosia, Cyprus
svassi04@ucy.ac.cy

Georgia Antoniou
University of Cyprus
Nicosia, Cyprus
ganton12@ucy.ac.cy

Davide Basilio Bartolini
Huawei Zurich Research Center
Zurich, Switzerland
davide.basilio.bartolini@huawei.com

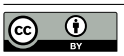
Yiannakis Sazeides
University of Cyprus
Nicosia, Cyprus
yanos@ucy.ac.cy

Abstract

Modern datacenter applications commonly adopt a microservice software architecture, where an application is decomposed into smaller interconnected microservices communicating via the network. These microservices often operate under strict latency requirements, rendering them particularly vulnerable to microarchitectural cold effects that may arise from the interleaved execution of services on cores or power-gating cores between invocations.

Previous analyses of microservices find branch mispredictions due to cold predictor resources to be a significant contributor to performance degradation, indicating that the dynamic control flow must be very similar in the set and order of executed instructions across different requests. Our analysis of control-flow similarity across requests, using static and dynamic control flow information to determine dynamic control flow reconvergence, confirms that, indeed, a large portion of requests follow similar paths.

Motivated by the above findings, we propose Similarity-based Branch Prediction (SBP), a hybrid predictor architecture that enhances conventional predictors with a similarity component. SBP leverages the control-flow similarity across microservice requests to predict control flow (branch direction and target) by utilizing the control flow of past executions encoded in a reference execution trace. We realize a specific instantiation of SBP, called CHESS, which combines a conventional history-based fetch predictor, a static-hint predictor, and a similarity predictor. CHESS judiciously applies similarity prediction for branches identified as hard-to-predict through conventional prediction techniques, effectively mitigating branch predictor cold-start effects while keeping the length of the reference trace practical. Evaluation through a suite of microservices shows that CHESS reduces branch MPKI by 94% over a cold fetch predictor and 78% over a state-of-the-art predictor, while requiring a modest 18.1KB of additional storage space. This enables CHESS to deliver performance that is, on average, within 95% of a warm baseline system.



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731059

CCS Concepts

• **Computer systems organization → Superscalar architectures**.

Keywords

Microservices, branch prediction, cold start

ACM Reference Format:
Haris Volos, Stylianos Vassiliou, Georgia Antoniou, Davide Basilio Bartolini, and Yiannakis Sazeides. 2025. Leveraging control-flow similarity to reduce branch predictor cold effects in microservices. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731059

1 Introduction

Modern applications deployed in datacenters often follow a microservice software architecture, where a monolithic application is decomposed into smaller interconnected services that explicitly communicate with each other via the network through well-defined interfaces. These applications exhibit irregular query streams, and their constituent microservices have often very tight (i.e., few tens to few hundreds of microseconds) latency requirements [28].

Microservices running on modern servers may suffer significant performance overhead due to microarchitectural cold effects. As argued in prior work, microarchitectural cold-effects can occur due to the interleaved execution of different microservices on the same core to improve resource utilization [53] or due to power-gating a core between microservice invocations to save power when idle [25, 62], with both resulting in a loss of microarchitectural state and forcing the core to relearn information across invocations. Cold-effects are particularly harmful to microservices, as the short duration of microservice invocations [28] makes it more difficult to warm up the microarchitectural structures [27].

Previous studies have examined the performance implications of microarchitectural cold effects in the context of microservices [15, 53, 54], identifying that the prediction resources at the front-end of modern cores contribute significantly to performance degradation. The existence of cold effects suggests that there is reuse of microarchitectural state across invocations of a given microservice. Prior work has recognized this opportunity and attempted to minimize such cold effects for microservices [53, 54, 56]. Lukewarm [53] records and replays instruction accesses to warm up a

cold instruction cache and minimize its impact on instruction cache accesses. Ignite [54] records and replays branch-target buffer (BTB) insertions to warm up a cold BTB together with a bimodal predictor and minimize their impact on branch predictions.

The large sensitivity of cold effects to branch mispredictions indicates that the dynamic control flow between queries must be very similar in the set and order of executed instructions. We quantify this by analyzing the control-flow similarity (CFS) between requests in a suite of microservices. Following methods similar to previous work [11, 37], we identify similarity by constructing the control-flow-graph (CFG) of a microservice and use a combination of static and dynamic control flow information to determine control flow reconvergence and the degree to which different executions follow identical or different control flow paths through the CFG.

The study reveals that requests often follow similar control flow paths, with 48% to 99% of dynamic branch instructions, typically more than 90%, appearing on the same control flow path. Moreover, almost all (99%) these dynamic branches have the same outcome between executions. Closer inspection of the workloads reveals that CFS is an outcome of both the microservice architecture and the application logic. Each microservice focuses on a small set of functionalities, limiting the number of control flow paths a request can take within a service and increasing in this way control flow reusability among requests.

Our results confirm prior studies that conduct a comprehensive analysis of CFS in web server [11] and microservice workloads [37], as well as previous work that observes CFS in serverless [53, 60] workloads. Unlike previous work, which exploited similarity to batch requests with similar control flow for efficient instruction fetch and decoding [11, 37] or to efficiently simulate terabyte-scale memories [44], we leverage similarity to mitigate cold-start effects in branch prediction within microservices.

Driven by the above findings, we introduce Similarity-based Branch prediction (SBP), a hybrid branch predictor architecture that enhances a conventional branch predictor found at the fetch stage of a modern core with a similarity branch predictor component that leverages the CFS across microservice requests to overcome predictor cold-start effects in microservice workloads. SBP derives dynamic and static control flow information from past executions into a reference execution trace. SBP utilizes this information at runtime to make control flow predictions. We realize a specific instantiation of SBP, called CHESS, which combines a conventional history-based predictor, a static-hint predictor, and a similarity predictor. CHESS judiciously applies similarity prediction for predicting instances of conditional branches and indirect jumps and calls for which a conventional history-based or static-hint predictor is found ineffective, effectively overcoming branch predictor cold-start effects while keeping the length of the reference trace practical. Our evaluation through a suite of microservices shows that CHESS reduces the branch MPKI by 94% over a cold fetch predictor and 78% over a state-of-the-art predictor, while using a modest 18.1KB of additional storage space. This enables CHESS to deliver performance that is, on average, within 95% of a warm baseline system. In particular, we make the following contributions:

- We confirm previous work with a comprehensive study that quantifies and analyzes CFS in microservices based on static and dynamic control flow relations, demonstrating a large degree of CFS among microservice requests.
- We introduce SBP, a new prediction approach that leverages CFS among requests in microservices to mitigate branch prediction cold effects.
- We develop CHESS, a specific instantiation of a SBP hybrid predictor that combines a history-based conventional predictor, a static-hint predictor, and a similarity predictor for hard-to-predict branches.
- We demonstrate that CHESS overcomes branch prediction cold effects for microservices and achieves a low branch misprediction rate at a reasonable cost.

2 Cold Effects and Branch Mispredictions

We study the impact of microarchitectural cold effects on microservices, showing that branch mispredictions due to cold effects can significantly degrade performance. Cold effects are especially harmful due to microservices' short duration, which makes the time required to warm up microarchitectural structures more pronounced.

2.1 Microservice workloads

For our study, we use open-source applications and workloads representative of realistic deployments, including Memcached [6] and seven microservices from the MicroSuite benchmark suite [59]. Memcached [6] is a lightweight key-value store that is widely deployed as a distributed caching service to accelerate applications [45, 63]. We drive Memcached using an extended version of the Mutilate load generator [39] configured to recreate the ETC workload from Facebook [18]. MicroSuite [59] is a microservice-based benchmark suite with four information-retrieval services, each composed of front-end, mid-tier, and bucket (leaf) microservice tiers: (i) HDSearch is an image similarity search service, (ii) Router is a Memcached protocol router service, (iii) SetAlgebra performs posting list set intersection, and (iv) Recommend is a recommendation service. We drive each service using the supplied workload generator.

2.2 Impact of microarchitectural cold effects

We quantify the implications of microarchitectural cold effects on microservice execution on a Skylake-based processor. We measure this by allowing cores to enter a deep sleep state (C6) between requests, effectively powering off and resetting microarchitectural structures. We compare to a warm baseline where cores stay active and retain their microarchitectural state between requests. We use Top-Down analysis [64] to examine the components of the core that contribute to the cold-start latency.

Figure 1 presents a Top-Down analysis for all the microservices. The analysis reveals that cold effects can increase the CPI of each tier of an application significantly, ranging from 25% to 126%. Noteably, the category with the highest overhead is Frontend Bound, which is further divided into Frontend Bound - Branch Resteers and Frontend Bound - Others (*e.g.*, ITLB, L1I). Branch Resteers represents the number of slots lost between resolving a mispredicted branch instruction until renaming an instruction from the correct path. Branch Resteers and Bad Speculation represent the penalty caused by branch mispredicts. Consequently, branch mispredicts

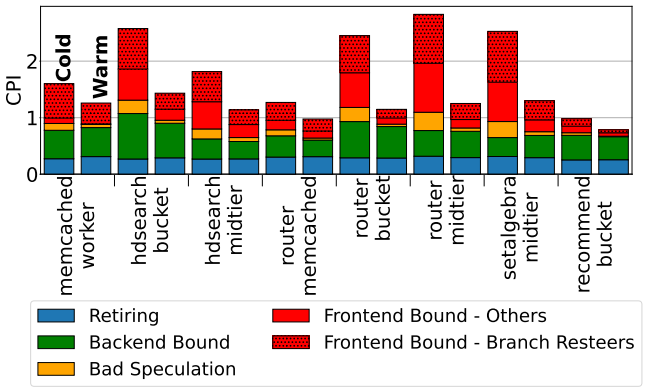


Figure 1: Top-Down Analysis for all benchmarks.

account for between 7% and 26% of the CPI of the warm core, while mispredicts caused by a cold branch predictor increase the overall CPI by 14% to 67% compared to a warm predictor.

Overall, we find that the prediction resources at the front-end of a modern core are responsible for a large fraction of the performance degradation of microservices due to cold effects. These findings align with previous studies on microarchitectural cold-effects in serverless functions [53, 54]. While our analysis is conducted on a Skylake-based processor, we expect branch mispredictions to have similar impact on microservices across other modern processors. For example, AWS Graviton4 and Graviton2 processors, which support microservices running in AWS EC2 and Lambda [2, 9], are built on ARM Neoverse V2 and N1 cores, respectively [1, 17]. These cores feature high-performance branch predictors [21, 49], suggesting that branch predictor performance can be an important consideration in production systems running microservice workloads.

2.3 Causes of microarchitectural cold effects

Cold-start in real systems can occur due to various factors, including co-residency and power gating. First, microservices often co-reside with other microservices or batch jobs [43]. When deployed with serverless computing frameworks [10, 20, 42], microservices could experience an even higher degree of co-residency [10]. Co-residency allows the system scheduler to interleave the execution of microservices on the same physical server. For an online invoice application based on microservices [16], the interarrival time between invocations of the same microservice can reach hundreds of milliseconds, allowing enough time for interleaving [53]. Interleaving can improve resource utilization but may degrade performance, as a microservice can evict the microarchitectural state of another microservice, leading to cold-start effects when the other resumes. Second, cold-start can occur due to power-gating a core between microservice invocations to save power when idle [25, 62]. Both cases result in a loss of microarchitectural state, forcing the core to relearn information between invocations.

3 Quantifying Control-Flow Similarity

Motivated by the large performance overhead of branch mispredictions on microservices due to microarchitectural cold-effects [15, 53, 54], we investigate the control-flow similarity (CFS) across requests. We find that by leveraging static and dynamic control flow information across requests for the same microservice, some branches can become quite predictable.

In the same vein as in prior work [11, 37], our study aims to gauge and understand the CFS within microservice workloads with similarity identified through dynamic reconvergence based on static and dynamic control-flow information within and across requests of a microservice. The static information is derived from a control-flow-graph of a microservice and the dynamic from execution traces of the microservice.

3.1 Methodology

To analyze CFS in our workloads, we first produce execution traces of instructions for each request. We use the execution traces to generate dynamically constructed control flow graphs [50] that capture the control flow of all requests in each microservice. We then analyze the constructed graph to identify the reconvergence points in the traces [52]. Finally, we analyze the degree of similarity by conducting pairwise comparisons between execution traces.

Execution Trace: An execution trace is the dynamic sequence of instructions executed by a program when run with a given input. An execution trace records for each instruction instance its PC, call-stack depth (CSD) and reconvergence address. Figure 2.c shows an example trace for the code in Figure 2.a. The trace consists of 28 instructions. The row labeled PC lists the executed program counters (PCs), with each entry representing a dynamic instance of an instruction and highlighted entries indicating conditional branches. The row labeled CSD shows the call-stack depth for each dynamic instruction in the example trace. The CSD is incremented by one whenever we have a call, for instance by instruction with address 16 that executes 8th in the dynamic instruction order, and decremented by one when we have a return, for instance by instruction with address 20 that two of its instances execute 22nd and 28th in the trace. We further explain how traces are collected in Section 6.

Control Flow Graph: A control flow graph (CFG) is a static representation of a program's control flow relationships [12, 14]. It is a directed graph where each node represents a basic block and each edge represents the flow of control between basic blocks. A basic block is a contiguous sequence of program instructions in which the flow of control enters at the beginning and leaves at the end without branching except possibly at the end. A post-dominator of a node (basic block) in a CFG is a node that appears on all paths from that node to the exit of the graph. The immediate post-dominator is the first post-dominator encountered on all such paths from the node to the exit.

We leverage a dynamically constructed control-flow graph algorithm to incrementally reconstruct the CFG from multiple execution traces [50]. The algorithm starts with an empty CFG and iteratively refines it as it traverses each trace and explores new paths through the CFG. Alternatively, we could rely on static binary analysis to construct the CFG.

Figure 2.b shows the CFG for the code in Figure 2.a. The basic block that starts at address 17 is the immediate post-dominator of basic blocks 3, 5, 7, 9, 11, 13 and 15, because it is the first node that appears in any path originating from those basic blocks to the exit

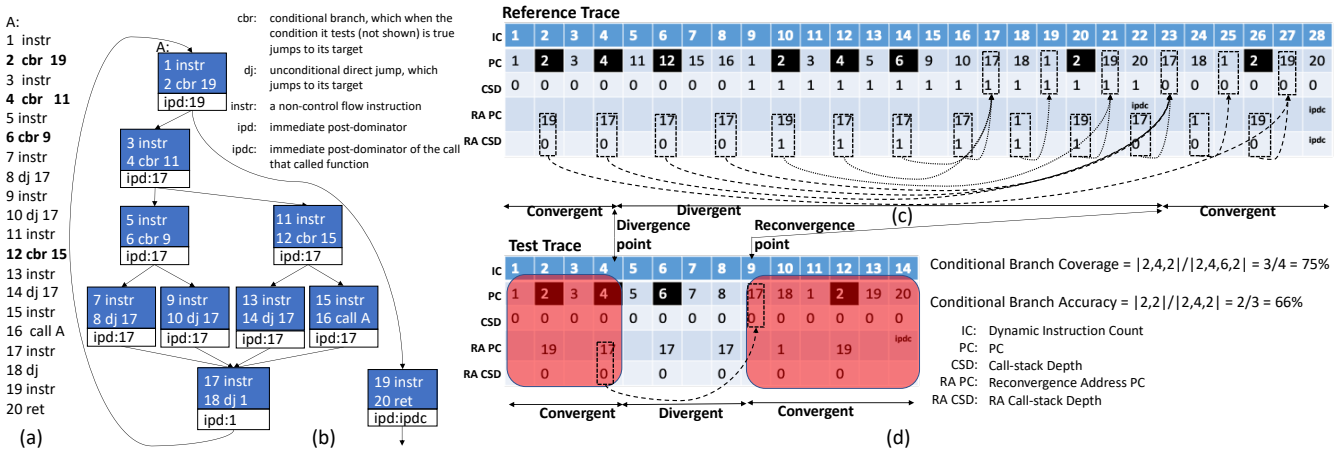


Figure 2: Figure shows: (a) listing of the code of a function, (b) its Control Flow Graph, (c) a Reference Trace and (d) a Test Trace and its CFS in terms of Coverage and Accuracy with respect to the Reference Trace

of the graph, including the target and fall-through paths of basic blocks 3, 5 and 11 that end with a conditional branch.

Reconvergence Points: We analyze the constructed graph to determine immediate post-dominators, which we then use to augment the execution traces with reconvergence points. Within an execution trace, the control flow reconvergence point (or simply reconvergence point) of a branch is the earliest subsequent dynamic instruction in the trace whose PC matches the reconvergence address of the branch and shares the same dynamic call-stack depth (CSD) as the branch [13]. The reconvergence address (RA) of a branch is the PC of the leader (first instruction) in the immediate post-dominator block of the branch in the CFG, guaranteed to be the first instruction that appears in all the control-flow paths emanating from the branch. Pairing a reconvergence address with a CSD is necessary because the reconvergence point is a dynamic instruction while the reconvergence address identifies a static instruction. The CSD helps distinguish between different occurrences of the same static instruction in the trace. This is useful in scenarios like recursive functions, where a static instruction may be visited multiple times at varying CSDs before reconvergence.

Figure 2.c shows the initial execution trace augmented with reconvergence information. The rows labeled *RA PC* and *RA CSD* list for each PC in the trace that ends a basic block in the CFG, the address of the instruction and the depth at its reconvergence point. We use arrows to point from each PC in the trace its reconvergence point. Note the RA PC and RA CSD of a PC match with the PC and the CSD where the arrows point. The reconvergence CSD usually has the same value with each trace PC's CSD and is shown for reading ease. The RA CSD for a return is one less than the CSD of the return.

We like to point out that the reconvergence point for PC with address 2 that executes second in trace order and has CSD 0, is the PC with address 19 that executes 27th in program order. It is not the instruction that executes 21st in trace order, because even-though the PCs match, the CSD is 1 and does not match the CSD of the second PC in the trace which is 0. We also mention, that the RA PC of a return is equal to the PC of the reconvergence point of the call that invoked the function instance that executes the specific return (denoted by ipdc in Figure 2.c). For clarity, note that the reconvergnece point for the return that executes 22nd in the trace, is equal to the reconvergence point of the call that executes 8th in trace order (with RA PC 17 and RA CSD 0).

Similarity Analysis: We analyze the degree of similarity between requests of the same type by conducting pairwise comparisons between execution traces as follows. We use two read pointers *test-ptr* and *ref-ptr* to sequence through the traces. They each point to a position in their respective trace and are initialized to point to the first instruction in the test trace and reference execution trace respectively. We define a test execution trace to exhibit CFS with respect to a reference execution trace when the test trace follows the same control-flow path as the reference trace. In essence, the test trace executes the same set of instructions in the same order as the reference trace. This occurs when the instructions pointed by each pointer in each trace have the **same PC and CSD**, in which case the two pointers advance to point to the next instruction in the trace. However, typically, a test trace may exhibit partial similarity with respect to a reference trace due to control-flow divergence. Control-flow divergence occurs when the two execution traces (test and reference traces) diverge at a branch instruction (divergence point), specifically when the outcome (or direction) of the same branch pointed in the two traces differs between them. In that case, each of the two read pointers is advanced to the reconvergence-point in each trace, effectively skipping over a region of dynamic instructions in each trace where control flow is divergent. The divergent region length and content between the two traces can be arbitrarily different. Following divergence, the test and reference execution traces will subsequently re-converge (barring events such as program exit or exceptions), thus alternating between phases of convergent and divergent execution.

We measure CFS between a test and reference execution traces by considering the convergent branches encountered in the test trace. Convergent branches are branches encountered in the test

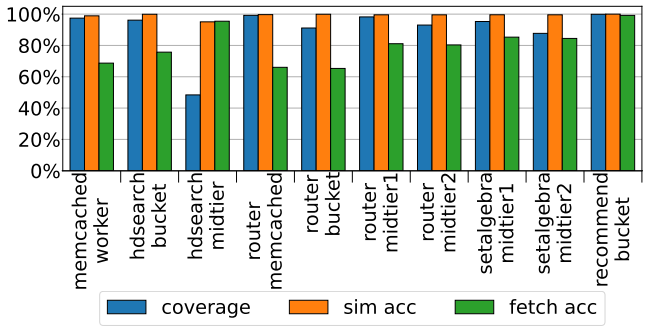


Figure 3: CFS Coverage and Accuracy Vs Fetch Accuracy

execution while the test and reference execution are on the same control-flow path, i.e., convergent.

We define two metrics to measure CFS: (i) *Coverage* is the ratio of convergent branches encountered in the test execution, relative to the total number of branches in the test execution. (ii) *Accuracy* is the ratio of convergent branches encountered in the test execution that have the same outcome as the reference execution, relative to the total number of convergent branches in the test execution.

Figure 2.d shows another trace, that we treat as test trace, for the code listing in Figure 2.a. We also show its coverage and accuracy with respect to the trace in Figure 2.c that we treat as the reference trace. The coverage and accuracy is shown only for the conditional branches in the trace. The test and a reference execution diverge at the conditional branch with PC 4 and CSD 0. This is the 4th instruction in both test and reference traces with the next PC being 11 in the reference trace and 5 in the test trace. The two traces reconverge at the basic block that starts with PC 17 and CSD 0. The reconvergence point corresponds to the 9th test trace instruction and the 23rd reference trace instruction.

Hence, the first two conditional branch instances with PC 2 and 4 are convergent as the two traces are on the same path up to and including branch 4. Branch with PC 6 in the test trace is divergent as it belongs to a divergent region. The second instance of branch 2 is convergent as the two executions reconverge at instruction 17 and depth 0 before the second instance of 2 occurs. Therefore, the test trace's conditional branch coverage from the reference trace is three out of four (75% coverage) and of the three covered branches two have same outcome as in the reference trace (66% accuracy).

3.2 Findings

Figure 3 shows the CFS coverage and accuracy per microservice. For the router midtier and setalgebra midtier microservices, two instances are shown for each, as processing spans two threads with sufficient activity to warrant separate analysis. The results are averages over all the requests for each benchmark. For each studied workload, the analysis uses the trace of one of the workload's requests as a reference trace. The results also show the prediction accuracy - for the same branches covered by similarity - of a simulated state-of-the-art branch predictor that starts predicting cold each request (we refer to this predictor as Fetch). The details for reference trace selection and the fetch predictor are given in Section 6.

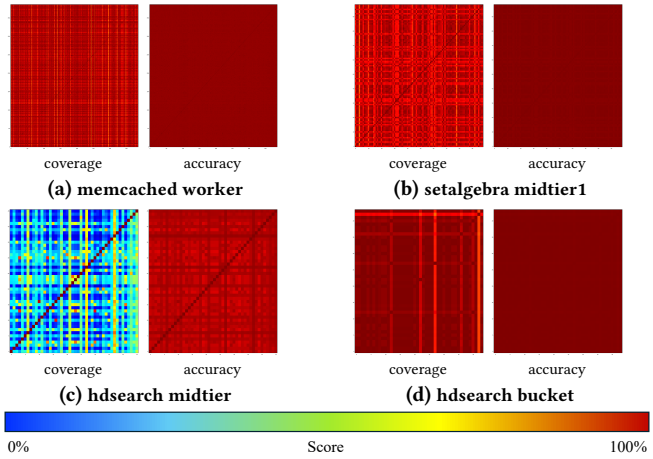


Figure 4: CFS coverage and accuracy heatmaps.

There are three key findings from Figure 3: i) the accuracy of the CFS is consistently higher, with the difference often ranging between 15-34%, except for the *hds midtier*, which has 0.4% lower accuracy than the fetch accuracy, ii) the accuracy of the CFS is often close to 99% or better for eight out of the ten workloads, and iii) the coverage of similarity is high, more than 95%, for half of the workloads but in one case, for *hds midtier*, is 48%.

Figure 4 presents pairwise comparisons of CFG coverage and accuracy among all execution traces, for a representative subset of microservices. For all services, the nearly uniform heat coloring suggests that the selection of traces for identifying a reference trace has minimal impact. *hds midtier* exhibits low coverage regardless of reference trace choice, primarily because its control flow is highly data-dependent on the query input. In particular, it employs locality-based hashing, where queries hash to linked-list buckets. An uneven distribution of items per bucket results in varying list lengths, leading to unpredictable control flow.

The overall findings confirm that the dynamic control flow between requests of the same microservice is often very similar [37]. Findings i) and ii) motivate the design of a branch predictor that leverages CFS across different requests to overcome the cold-start effects from branch mispredicts in microservice workloads. Finding iii) suggests that the new predictor needs to be hybrid since the similarity coverage is less than 100%, this predictor will employ a CFS-based component together with other component(s) that provide the predictions not covered by similarity. Section 4 discusses a similarity-based prediction (SBP): a hybrid branch predictor architecture that incorporates a component to predict CFS. In Section 5, we present a specific instantiation of a SBP-based hybrid predictor, referred to as CHESS. CHESS aims to reduce the trace size required for similarity predictions without losing coverage and accuracy.

4 Similarity-based Branch Prediction (SBP)

We introduce *similarity-based branch prediction (SBP)*, a hybrid branch predictor that can include any conventional branch predictor found at the fetch stage of a modern core and a similarity branch predictor component that leverages the CFS across different requests to overcome predictor cold-start effects in microservice workloads. SBP derives dynamic and static control flow relations

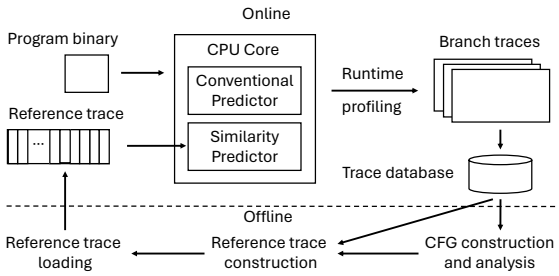


Figure 5: Similarity-based branch prediction.

from past executions and uses this information at runtime to make branch predictions and detect reconvergence points.

Figure 5 gives a high-level overview of the approach. We present the main required functionalities and structures of SBP without going into implementation details. SBP collects execution profiles of different requests of a workload and analyzes these profiles offline to construct a reference trace. A reference trace encodes the dynamic control flow, i.e., the branch outcomes, of a representative reference execution together with reconvergence information (point in the trace to reconverge in case of divergence). At runtime, the reference trace is fed into the similarity branch predictor for making predictions. As long as the current execution fetch order adheres to the reference control flow, SBP utilizes the branch outcome of the reference execution as the predicted outcome for the currently fetched branch. When the current execution diverges, SBP reverts to the conventional fetch predictor for control-flow predictions until the current execution reconverges with the reference trace.

Augmenting a conventional fetch predictor with a similarity helper predictor offers an effective design. As also argued in prior work [40], a conventional fetch predictor, such as TAGE-SC-L, performs exceptionally well on most branches, so it is preferable to keep it in place and use any additional resources to augment the predictor with methods that directly address specific challenges. Moreover, augmenting the predictor with an offline-trained helper predictor to handle branches that often mispredict is well-suited for data centers, where high performance is critical and training costs can be amortized at scale.

While alternative approaches to exploiting control flow similarity are possible, each has limitations. For instance, pinning services to cores does not eliminate cold-start effects from power-gating and most importantly restricts CPU scheduling. Achieving high CPU efficiency by scheduling and reallocating cores across applications remains important for latency-critical datacenter workloads [46]. Alternatively, prior work has explored virtualizing branch predictor components. Maintaining multiple on-chip contexts for key predictor components and switching between contexts, according to the application that currently executes, mitigates branch predictor side channels [61]. This could potentially help avoid cold-start effects from co-residency and interleaved execution of microservices, but may impact predictor timing on the critical path and does not address cold-start effects from power gating. Virtualizing the branch target buffer (BTB) by storing metadata in the cache hierarchy can increase BTB capacity, but does not address direction predictor [22]. Ignite effectively virtualizes both BTB and bimodal component of a conditional predictor but does not address full predictor state [54]. Finally, as we show in the evaluation, increasing the capacity of the predictor state does not necessarily improves accuracy.

4.1 Trace collection and analysis

SBP collects execution traces of branch instructions for each microservice. The trace records the address of each branch instruction encountered during execution along with its branch target address. This trace format fully captures the control flow of the execution. The target of a branch marks the beginning of a contiguous block of instructions. The block ends with a branch instruction, identified by the next record in the trace. SBP collects traces over a configurable tracing period using a lightweight hardware tracing facility, such as Intel Processor Trace [8]. Datacenter workloads change gradually over several weeks [24], providing flexibility for operators to trace every few weeks using a week-long tracing period. During a week-long tracing period, trace collection can use periodic sampling, gathering several hundred profiles following production practices [24]. SBP stores each execution trace in a trace database for later offline analysis.

After the end of the tracing period, SBP performs offline analysis of all the collected traces of a microservice. This analysis reconstructs a control-flow graph (CFG) for every function encountered in the traces that effectively captures the set and order of executed instructions across all the profiled requests. The CFG is used to determine various static relations, such as determining post-dominators when later constructing the reference trace.

4.2 Reference trace construction

SBP constructs a reference trace that represents the dynamic control flow of a representative reference execution, which is used for similarity prediction. Since the effectiveness of the similarity-based branch predictor depends on the similarity between the test and reference execution, SBP picks the reference execution that maximizes coverage and accuracy across all the collected traces (Section 6).

SBP augments the original branch trace, which encodes the execution as a sequence of branch instructions and corresponding targets, with static and dynamic relations to predict convergence. For each branch instruction, the trace includes a record that comprises of four fields: (i) branch address, (ii) branch CSD, (iii) branch outcome (target address), and (iv) branch reconvergence address. The target of a branch enables the SBP to predict the next branch instruction in the fetch stream when the current execution is convergent. The reconvergence address of a branch, along with the branch CSD, enables the SBP to determine the control reconvergence point when divergence occurs. This point is the earliest dynamic instruction where two divergent executions that split at the given branch may reconverge and the similarity-based branch predictor may resume offering predictions.

SBP determines off-line the static reconvergent address of a given branch using post-dominator analysis on the control-flow graph. An alternative to leveraging static analysis for determining reconvergence points [52] is to use online identification of these points using heuristics in special hardware [26]. We leave exploring this direction to future work.

The immediate post-dominator of the basic block enclosing a given branch can serve as a reconvergence point as it is the first block that appears on any path from the given branch to the exit of the control-flow graph. However, because the immediate post-dominator may not always end with a branch, SBP traverses the sequence of basic blocks starting from the immediate post-dominator till it reaches a basic block that ends with a branch instruction. This is needed so that a RA PC can match a branch PC stored in the trace so that a reconvergence point can be identified. SBP uses the static address of that branch instruction as the reconvergence address.

Figure 6.a shows an example branch reference trace. This example corresponds to the instruction trace in Figure 2.c. A key difference in Figure 6.a is that it only includes branches and, consequently, is shorter: 14 vs 28 entries. It also includes an extra field with the Target of each branch instance (i.e, its outcome, the address of the next PC). The dynamic instruction count in the figure corresponds to the original position of each instruction in the trace in Figure 2.c and is shown to ease readability. We also show the RA CSD, although it can be derived from the other fields, again for readability ease. Finally, note that RA PCs between the two figures are different because we walk until we find a branch and include its PC as the RA of another branch. In our example listing, all the basic blocks contain two instructions and start with a non-control-flow instructions and end with a branch, that is why all the RA PCs have advanced by one as compared to Figure 2.c.

4.3 Branch prediction

At runtime, SBP utilizes a branch target encoded in the reference trace to predict the next instruction in the stream. SBP uses the past outcomes recorded in the trace for as long as the current execution adheres to the reference control flow, i.e., remains convergent. If the actual branch target disagrees with the one provided by SBP, i.e., a branch misprediction occurs, SBP marks the current execution as divergent at the incorrectly predicted branch. The processor then falls back to the conventional branch predictor for prediction until the current execution reconverges with the reference control flow. When the execution reconverges to the reference control flow, the processor resumes predictions using the predictions from the reference trace.

SBP determines reconvergence to the reference control flow with the aid of the branch reconvergence address and the CSD encoded for the mispedicted branch in the reference trace. Determining reconvergence entails two steps.

First, SBP must identify the reference trace entry that corresponds to the dynamic reconvergence point. When SBP detects divergence at a given branch, it notes down the branch reconvergence address and CSD of the divergent causing instruction in the current execution. Then, it scans and skips through the reference trace until it finds a match: an entry with a branch address that matches the reconvergence address and CSD of the branch that caused divergence.

Second, in addition to identifying the reconvergence point in the reference trace, SBP monitors the execution to determine when the execution reaches the reconvergence point. SBP maintains the CSD of the execution to determine when it reaches the reconvergence point, with a call instruction increasing the dynamic CSD, while a return instruction decreasing it. The execution reaches the dynamic reconvergence point when i) it fetches a PC equal to the reconvergence address and ii) the dynamic CSD is equal to the RA CSD.

Returning back to our running example, assuming Figure 6.a is the reference trace and Figure 2.d represents an actual execution (not a trace). The hybrid SBP predictor will start in a convergent state with the similarity predictor component providing predictions using the targets recorded in the reference trace. When the actual execution determines that the prediction provided by SBP, for the instance of conditional branch with PC 4 found at entry with IC=4 in Figure 2.d, is incorrect, SBP will mark the execution as divergent. The SBP is wrong since its prediction for the next PC is 11 but the actual execution's next PC is 5. SBP, then switches to the conventional predictor. For instance, branch with PC 6 (corresponding to the entry with IC=6 in Figure 2.d) will be predicted by the conventional fetch predictor. SBP will resume similarity predictions only after the current execution fetches the instruction that its PC and CSD are equal with the reconvergence address and CSD of the divergent branch instance of 4. This corresponds to the branch with PC 18, and with depth 0, the 12th entry (TC=12) in the branch reference trace.

5 An SBP Instantiation: CHESS

Capturing all branches in the reference trace would provide a complete and accurate representation of the reference execution, ensuring no information is lost. However, it may also result in a long trace. One concern with long traces is large storage overhead. Other critical concerns are with the size of the on-chip buffers that will hold the reference trace and provide predictions, the bandwidth contention to load the trace from memory, and the timeliness to have the trace loaded in time to provide predictions when execution needs them. Consequently, reducing the SBP reference trace seems essential as long as this reduction does not impact SBP's accuracy and coverage.

In this section, we discuss a specific instantiation of SBP, referred to as CHESS, that uses a number of optimizations to reduce the reference trace length while maintaining high CFS coverage and accuracy. CHESS is designed to use the similarity predictor component as an overriding predictor in a post-decode stage of the pipeline [36]. This way, CHESS focuses on CFS of branch types whose outcome is unknown after decode: conditional, indirect jumps and indirect calls. We are not considering using CFS for return addresses because existing return-address predictors have very high accuracy.

5.1 Reducing the Length of a Reference Trace

We exploit two insights/observations to make trace length practical by removing direct branches and returns and by removing predictable conditional and indirect jumps and calls.

Direct Branches and Returns: Our first insight is that several branches, including direct calls and direct jumps are unidirectional following a single control flow path. Given CHESS is used post-decode and aimed at indirect jumps, indirect calls and conditional branches, these control-flow-instructions can simply be removed without any information loss. When removing such branches from

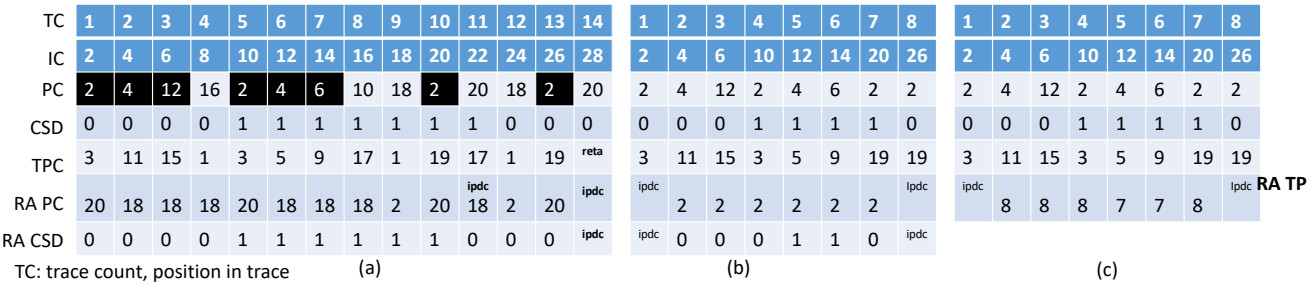


Figure 6: (a) SBP reference trace as a sequence of branch instructions, CHESS branch reference trace: (b) without direct branches and returns, (c) using pointer to reconvergence point in the trace (row RATP)

the trace, we check whether any removed branch serves as a reconvergence point of other branches in the trace. If so, we determine the nearest postdominator of the removed branch in the trace, and update the reconvergence addresses of those other branches to the address of the branch that is the nearest postdominator. We repeat this process, if that branch is removed as well.

The call depth recorded for the remaining branches in the trace stays the same, except when their reconvergence point changes and it is found at a different call depth than before. This can happen when removing a direct call instruction or a return instruction. Unfortunately, this means that the CSD of a divergent instruction at fetch time cannot be used to determine the depth where reconvergence will occur. To avoid adding an extra field in each trace entry to indicate the depth where reconvergence occurs, we replace the reconvergence address of a branch in a trace with a pointer to the entry that holds its reconvergence point in the trace. Consequently, when the similarity predictor is convergent and mispredicts, it follows the pointer and sets the reconvergence point to the PC address and depth of the pointed position. Besides being a cost optimization, this change speeds up the process of detecting the reconvergence point in the reference trace when a divergence occurs.

Figure 6.b shows the new version of the reference trace of our running example after removing the unconditional direct jumps and returns and updating the reconvergence point of each branch (i.e., RA PC and RA CSD). As compared to Figure 6.a, we notice a trace length reduction from 14 down to 8 since only the conditional branches remain in the trace. We also observe that the reconvergence points have changed. For instance, consider the conditional branch with PC 2, CSD 1, Target PC 19, RA PC 20 and RA CSD 1, which is found in position with TC=10 in Figure 6.a; in the new trace in Figure 6.b it is found in position with TC=7 with RA PC 2 and RA CSD 0. This occurs because in Figure 6.a the branch's post-dominator with PC 20 and CSD 1 is a return, which is removed, and its post-dominator with PC 18 and CSD 0 is an unconditional direct jump, which is also removed. We eventually end-up with the direct jump's post-dominator, which is a conditional branch with PC 2 and CSD 0, which is in the trace and used as the new reconvergence point in Figure 6.b. We also show in Figure 6.c a version of the trace from Figure 6.b where we replace a branch's reconvergence point (i.e., its RA PC and RA CSD) with a pointer at the position in the trace that holds the dynamic branch at its reconvergence point. These pointers are shown in the row labeled as *RA trace position (RATP)*. For example, all branches in Figure 6.b with reconvergence point RA PC 2 and RA CSD 0 are replaced in Figure 6.c with position TC=8 since it is the closest dynamic instruction that holds the PC=2 and CSD=0.

Conditional and Indirects : The second insight is that several static branches that are found in almost all traces can have virtually all their dynamic instances predicted with high accuracy either statically with hint bits or dynamically through the fetch predictor even when it starts predicting from cold. The statically predictable conditional branches - heavily biased towards taken or not-taken direction - can be identified during the offline analysis of the collected traces. The dynamically predictable branches by a cold fetch predictor can be determined using Last Branch Records and hardware counter guided optimization analysis [3] or analogous technologies.

We can omit storing these branches in the trace and rely on static hints injected into the program binary during the offline analysis that indicate the prediction scheme to be used by each branch on the fly. More specifically, to remove a branch from the reference trace, we add two hint bits to branches (sometimes available in an ISA's instruction encoding [58]) to indicate whether to: (i) Use a static prediction scheme for conditional branches that virtually always follow the same direction across traces as revealed by the off-line trace analysis. The static hint also indicates the direction of prediction, annotated with 11 for taken and 10 for fall-through, or (ii) Use the fetch predictor for branches that are highly predictable by the fetch predictor when it starts serving a request cold, annotated with 00, or else (iii) Use the similarity predictor when execution is convergent; else use the fetch predictor when execution is divergent, annotated with 01.

This scheme **c**ombines a **h**istory based conventional predictor, a **s**tatic predictor and a **s**imilarity predictor: We refer to it as CHESS.

For the rest of the paper, we refer to the static branches that are statically predictable or predictable by the fetch predictor (that starts serving a request cold) as easy-to-predict (EP), and the remaining branches delegated to the similarity predictor when execution is convergent as hard-to-predict (HP). A branch reference trace produced with the above optimizations retains only the dynamic instances of HP branches.

5.2 Trading off Coverage and Trace Length

While removing EP conditional branches and indirect call and jumps can effectively reduce the trace length, it also results in missing

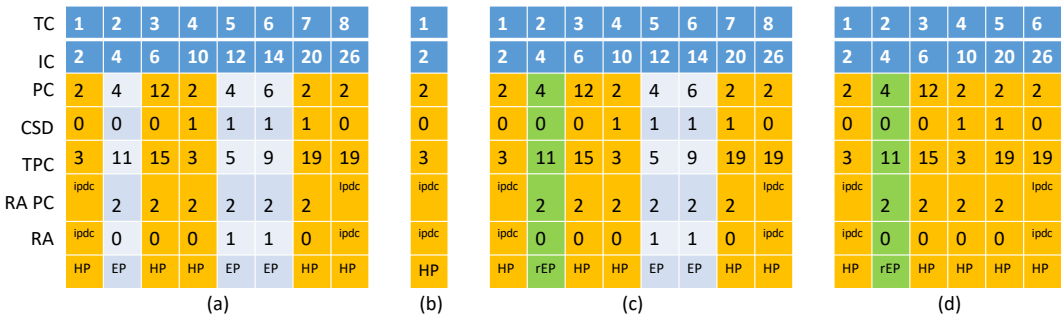


Figure 7: CHESS branch reference trace: (a) with branches in position 2, 5 and 6 labelled EP and in position 1,3,4,7 and 8 HP, (b) the resulting trace when we remove EP in positions 2,5 and 6, (c) with branches in position 2 labelled as rEP, (d) the resulting trace when we retain EP at position 2 and remove EP in positions 5 and 6.

control flow paths and reconvergence points. We can use the same approach used for direct jumps to maintain valid reconvergence address for the remaining branches in the trace, i.e., when a removed branch serves as the reconvergence point for another branch, then the reconvergence point of the removed branch becomes the new reconvergence point of the other branch. However, this also requires making the similarity predictor more pessimistic and switch to divergent mode when the next trace instruction after a HP instruction is EP and, therefore, removed from the trace (irrespective of whether the HP entry offers the correct prediction). This is needed, because failing to switch to divergent mode breaks the main similarity predictor invariance: to know when execution diverges and where to reconverge. This is the case because after an EP is removed it is impossible to know the path followed by its execution. More specifically, whether the actual EP branch target agrees or disagrees with what was in the trace. For example, the EP can be correctly predicted but disagree with its prediction that was in the trace or incorrectly predicted and agree with what was in the trace. Although the removed EP is likely to follow the path that was predicted in the trace, if it turns out to be different, the similarity predictor is unable to detect that execution diverges and will end-up using an incorrect reconvergence point when it detects divergence for HP branches that remain in the trace.

Unfortunately, by forcing divergence on a correctly predicted HP branch whose next trace entry is EP, may end up making the HP branches that come after the EP branch in the trace unreachable. This, in turn, reduces the coverage and usefulness of the similarity predictor. We illustrate the problem with removing EP branches from a trace in Figure 7.a. The figure labels as EP each branch instance with PC 4 and 6 and as HP instances with PC 2 and 12. The first trace instruction is HP and the second is EP. Because we remove the second instruction from the trace we are forced to assume that the HP instruction is divergent. Since the first trace instruction's reconvergence point is the immediate post-dominator of the function's caller instruction it means we loose the opportunity to predict any other HP branches in the trace, it is as if the reference trace includes a single instruction Figure 7.b.

To avoid this problem, we suggest a compromise: retain an EP branch (refer to as retained-EP or rEP) in the trace when there is an HP between the EP branch and the EP's post-dominator in the trace. This way, we can guarantee the same HP coverage as when using the

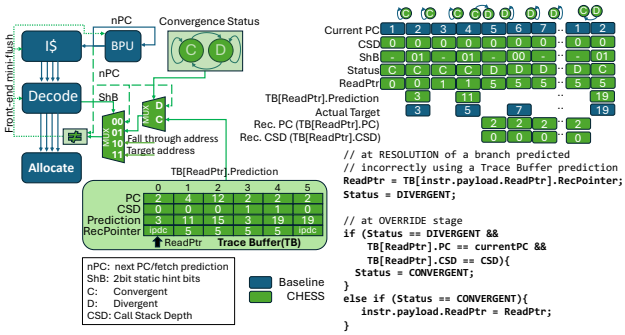


Figure 8: CHESS Implementation

full trace, but with a length larger than the number of HP branches. We analyze this cost-benefit in the evaluation. We illustrate this transformation in Figure 7.c. When we construct the reference trace, we retain the EP with PC 4 in position 2 in the trace because between this branch instance and its immediate post-dominator (PC 2, CSD 0, position 8) there exist three HPs. When we consider the EP in instruction 5, we do not need to retain it in the trace because between the instruction and its post-dominator (PC 2, CSD 1, position 7) there are no HPs. Same is true for the EP in position 6 in the trace. Therefore, the execution when it reaches branch with PC=2 and CSD=1 in position 4 in the trace, is guaranteed it will reach PC=2 with CSD=1 at position 7, irrespective of the removed EPs outcome. So, we can remove the EPs in positions 5 and 6 in the trace in Figure 7.c and end-up with the trace in Figure 7.d that contains only 6 instructions.

At the end of this process, we relabel each HP and retained EP such that their reconvergence point is their nearest post-dominator that is a HP or a retained EP. Then we remove all the EPs. Future work can consider heuristics that trade-off HP coverage to reduce the reference trace length.

5.3 Implementation

Figure 8 shows a CHESS implementation integrated in the front-end of a modern core. CHESS relies on static-hint bits encoded in a branch and a one-bit FSM, that tracks the convergence status, to help determine which prediction to use. Branch predictions from a Trace Buffer (TB) indexed by a read-pointer (ReadPtr), that store the

optimized reference trace, are consumed when i) the convergence-status is convergent and ii) a branch's static hint-bits indicate the use of the similarity component. On each read, CHESS advances the ReadPtr to the next entry. CHESS causes a front-end mini-flush when it disagrees with the fetch prediction.

Figure 8 also shows the contents of the TB when loaded with the reference trace in Figure 7.d and how the CHESS relevant state is updated during an execution that corresponds to the trace in Figure 2.d. The example execution includes a transition from convergent (C) to divergent (D) status and the opposite. An excerpt of the algorithm used to implement CHESS transition from DIVERGENT to CONVERGENT is provided as a code listing.

Determining reconvergence: Each branch predicted by similarity maintains a copy of the ReadPtr to the TB entry that provided its prediction. When branch resolution determines this branch as mispredicted, it updates the TB's ReadPtr to point to the TB entry pointed by the reconvergence pointer (RecPointer) stored in the TB entry of the mispredicted branch. Execution reaches the dynamic reconvergence point when the PC matches the pointed address, and the dynamic CSD matches the pointed depth.

Reducing the Size of an Entry: Several storage optimizations can be performed to reduce the size of trace entries. One method is to store pointers to a separate table with the unique branch PCs. The same optimization can be performed for the target PCs. Another way to reduce storage is by storing deltas instead of PCs. We evaluate the CHESS storage requirement in Section 7.

Storing Reference Trace: For each active microservice, we periodically collect execution traces and construct a reference trace. Because the reference trace contains virtual addresses, the trace is only valid for the specific microservice process. Therefore, the operating system (OS) must associate each trace with a process. For example, in Linux we can allocate a trace buffer for each request type that is contiguous in physical memory and store the physical addresses of the buffers in the process descriptor (`task_struct`) [53].

Loading Reference Trace: At runtime, a microservice identifies the type of an incoming request based on the entry point that serves the request and instructs the OS to activate the corresponding reference trace. The OS uses a privileged hardware control interface to bulk load the reference trace from memory into a trace buffer in the core, set SBP status initially to convergent, and start execution.

6 Methodology

Trace Collection: For generating execution profiles, we use a c220g5 server node from CloudLab [4, 29], equipped with two Intel Xeon Silver 4114 [35] Skylake-based processors running at a nominal frequency of 2.2 GHz. We run all the benchmarks on a single server node, using Intel Processor Trace (Intel PT) [8] to capture execution traces, similar to recent work [38, 58]. Intel PT, widely deployed in data centers, records complete control flows with low overhead [65]. Using Intel PT, we capture a full execution trace for each microservice and post-process it to extract individual request traces.

We determine the beginning and end of each microservice request within the captured trace using the RPC entry and exit points for each request type (or analogous entry function for non-RPC requests, such as in Memcached). We use these points to split each captured trace into smaller request traces, each corresponding to an individual request. Each request trace effectively corresponds to a unique user query with distinct input, as the workload generator of each benchmark randomly selects inputs from a large population, ensuring near-zero repetition across requests. We further split the request traces into two sets: (i) a training set, comprising 80% of all request traces randomly selected, used to construct the reference trace and static hints, and (ii) a testing set, comprising the remaining 20% of request traces, used to evaluate prediction accuracy.

CPU Clock Rate	4GHz
Back-end	ROB/LQ/SQ/Sched: 352/128/72/128 entries
Pipeline Widths	Fetch/Decode/Disp/Retire: 6/6/6/5
Branch Predictor	BTB: 1024 sets, 8 ways Indirect: 4096 sets, 2 ways, path-based Conditional: 64KB TAGE-SC-L RAS: 64 entries
Caches	IL1: 32 KB, 8 ways, 4 cycles, 8 MSHRs DL1: 48 KB, 12 ways, 5 cycles, 16 MSHRs L2: 512 KB, 8 ways, 10 cycles, 32 MSHRs LLC: 2048 KB, 16 ways, 20 cycles, 64 MSHRs
Prefetchers	IL1/DL1/L2/LLC: NL/NL/IP Stride/NL
TLBs	L1 ITLB: 16 sets, 4 ways, 1 cycle, 8 MSHRs L1 DTLB: 16 sets, 4 ways, 1 cycle, 8 MSHRs STLB: 128 sets, 12 ways, 8 cycles, 16 MSHRs
Memory	DRAM Data rate: 3200 MT/s

Table 1: Hardware model parameters (NL: Next Line).

Reference Trace and Static Hints: To construct a reference trace, we select the trace from the training set that maximizes coverage with all other traces of the same request type within the training set. For simulating static hints, we maintain an additional mapping that maps each static branch to its hint bits, and annotate branches as follows: static-predicted those branches that occur in nearly all traces and predominantly take one direction 95% of the time, fetch-predicted those branches where a cold fetch predictor achieves over 95% accuracy, and similarity-predicted the remaining branches.

Trace-based Simulation: For evaluating branch predictor accuracy, we develop a simulator fed by instruction traces collected via Intel PT. For all predictors, we measure accuracy using the testing set, comprising the remaining 20% of collected traces. To simulate a *cold* predictor, before running each trace, we initialize all predictor entry fields to zero except prediction-hysteresis counters that, unless indicated otherwise, are set to weakly-taken. To simulate a *warm* predictor, we use the training set to warmup the predictor. Because we focus on branch types whose outcome is unknown after decode, we measure prediction statistics by considering only conditional branches (cb), indirect jumps (ij) and indirect calls (ic).

Configurations: We evaluate the following predictors:

Fetch: uses a TAGE-SC-L [55] predictor configured with a 64KB TAGE, a statistical corrector and a 4k set x 2-way associative indirect jump predictor [23].

Fetch-Unbound: uses an essentially unbounded TAGE-SC-L that has as many tables as *Fetch* but much larger (2MB each).

Fetch-Static: a CHESS predictor without the similarity component, combines the fetch predictor with static hint bits that indicate the bias of a branch determined offline.

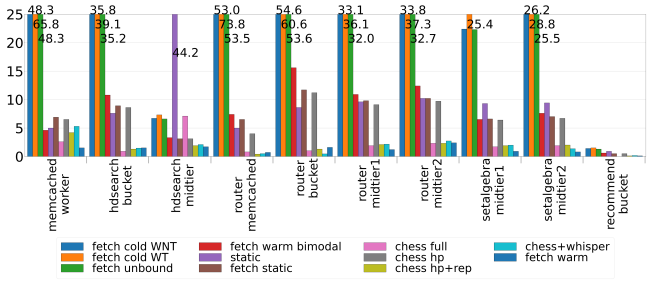


Figure 9: MPKI of all evaluated predictors.

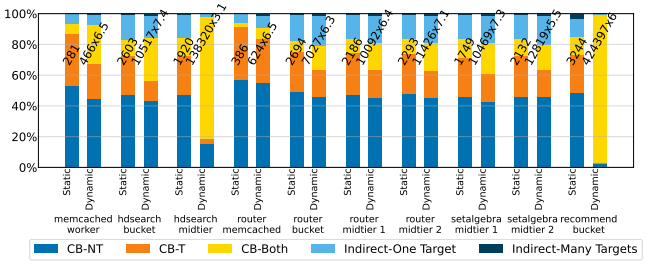
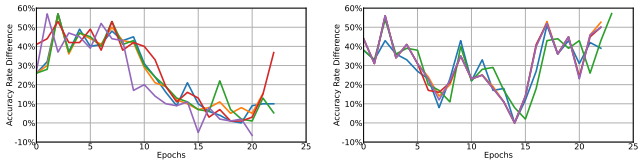


Figure 10: Conditional branches (CB) and indirect branches. Labels above static bars indicate number of static branch instructions. Labels above dynamic bars indicate product of dynamic branch instructions and instructions per branch.



(a) Router Midtier1 **(b) HDSearch Bucket**

Figure 11: Miss rate difference (fetch - similarity) per epoch for representative queries. Each line corresponds to a different query. Epoch size equal to 100 instructions.

Static: a predictor with static hint bits only, without the similarity and fetch predictors.
Warm-Bimodal: models a warm bimodal predictor between requests for conditional branches [54].
CHESS-Full: keeps all branches in the reference trace, minus the direct calls and jumps, and returns. When convergent, it uses the similarity predictor. When divergent, it reverts to the static or fetch predictor, depending on the annotation.
CHESS-HP: only keeps the hard-to-predict (hp) branches.
CHESS-HP+rEP: In addition to the hard-to-predict branches, it retains some easy-to-predict (rEP) branches.
CHESS-Whisper: Like CHESS-HP, but uses Whisper boolean formulas [38] for predicting HP conditional branches.
Performance Modeling: For measuring performance, we used the Champsim simulator [33] to model the execution of an IceLake [48] like core, with the parameters summarized in Table 1. We evaluate four configurations that vary in terms of the initial state of the conditional direction and indirect branch predictors: Fetch-Cold, Warm Bimodal, CHESS-HP+rEP and Fetch-Warm. For all configurations, core resources that are updated by the instruction stream, BTB, I$, ITLB, are assumed to be warmed by a previous query as proposed in prior work [54]. The data cache and DTLB start completely cold and the L2 and LLC are warmed only by instructions blocks of the previous query. This helps isolate the performance benefits of CHESS over prior art [54]. The simulator implements CHESS as a predictor that overrides the fetch conditional direction (TAGE-SC-L) and indirect (path-based) predictors after two cycles. This delay is meant to capture the time required to use pre-decode information from the instruction cache to drive the CHESS prediction.

7 Evaluation

Prediction Accuracy: We evaluate the accuracy of various CHESS configurations and compare it to various prediction techniques, quantifying accuracy using branch mispredictions per kilo instructions (MPKI). Figure 9 shows the results. On average, CHESS HP+rEP reduces MPKI by 94%, 78% and 75% over the fetch cold (when initialized with weakly not taken), warm bimodal, and fetch-static predictors, respectively, while delivering similar accuracy to Whisper boolean formulas. The fetch predictor suffers from high MPKI due to cold-start effects, averaging about 30.7 and 37.6 MPKI when initialized with weakly not taken (WNT) and weakly taken (WT), respectively. With the exception of the HDSearch-midtier and Recommend-bucket services, the duration of microservice requests is relatively short, consequently the fetch predictor lacks sufficient time to warm up for accurate predictions, resulting in a high MPKI. Using an essentially unbounded fetch predictor does not provide any noticeable benefit, which further reveals that the problem stems solely from cold effects, rather than a combination of cold effects and aliasing. Combining the fetch predictor with static hints, helps address cold-start mispredictions, resulting in an average MPKI of 7.1.

Figure 10 provides a breakdown of static and dynamic conditional and indirect branches in terms of how many unique targets they take. Many static branches with significant dynamic weight go in one direction, which explains why static hints are effective. The majority of one-way conditionals are not taken, which explains why initializing the fetch predictor to weakly not–taken behaves better than to weakly–taken. By leveraging CFS, CHESS HP+rEP effectively addresses the remaining cold-start mispredictions. This results in a significant boost in accuracy, reducing MPKI to an average of 1.8, approaching the accuracy of a warm predictor with an average MPKI of 1.2. For the longer running HDSearch-midtier and Recommend-bucket services CHESS HP+rEP offers no benefit over fetch-static.

Regarding fetch-corrects that similarity overrides incorrectly, we have found this to be a rare occasion occurring on average with less than 1% frequency.
Sensitivity of Similarity Benefits to Time: We analyze how the accuracy of fetch and similarity predictors evolves during microservice execution, focusing on branches where the similarity predictor made a prediction. Figure 11 shows the difference in miss rate (fetch - similarity) per epoch for two representative workloads. In Figure 11.a, similarity offers higher benefits early in a query, gradually decreasing but remaining positive. In Figure 11.b, benefits fluctuate across phases.

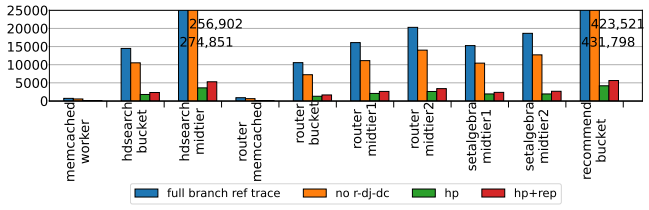


Figure 12: Reference trace length of CHESS predictors.

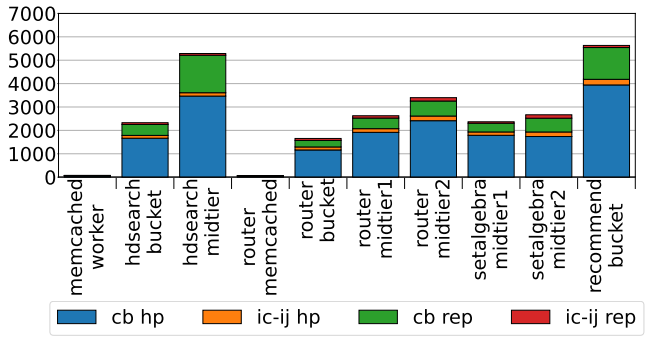


Figure 13: Reference trace length breakdown into hard-to-predict and easy-to-predict branches for CHESS.

Sensitivity to Reference Trace Length: Next, we evaluate how well CHESS meets its primary objective of reducing the size of the reference trace without experiencing a significant loss in accuracy. For this, we analyze the reference trace length for the various CHESS configurations and show the results in Figure 12 and contrast them with their implications on accuracy in Figure 9. As shown, eliminating returns, direct calls, and direct jumps in the *full branch reference trace* via CHESS-Full notably reduces the trace size. This reduction is inherently lossless, as it retains all essential information required to capture the reference control flow, thus fundamentally ensuring no loss of accuracy.

Additionally removing all the EP branches while retaining all the HP branches through CHESS-HP further reduces the trace length by an order of magnitude. However, as shown in Figure 9, this reduction comes with a 225% higher branch MPKI on average, compared to CHESS-Full. Removing all the EP branches leaves CHESS with only HP branches that can serve as reconvergence points. This, drastically reduces coverage and limits the opportunity to reconverge and leverage CFS for prediction in case of divergence. As a result, the branch prediction primarily relies on the fetch-static predictor.

By retaining the rEP branches, CHESS HP+rEP recoups the loss in accuracy, bringing accuracy on par with CHESS-Full, with a modest 35% increase in trace size.

An interesting note is that for HDSearch-midtier, the accuracy of the fetch-static predictor is higher than that of the CHESS-Full. This is because CHESS-Full always uses the similarity predictor while convergent. However, as indicated in Figure 3, the fetch-predictor component has a higher prediction accuracy than the similarity, while similarity has low coverage.

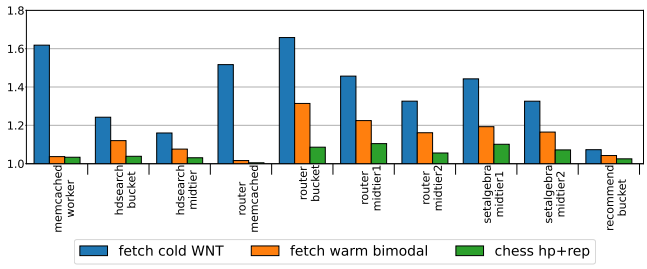


Figure 14: Performance of CHESS and other predictors over a warm predictor and core.

Finally, in Figure 13, we study the reference trace with only HP and the rEP instructions. Notably, we observe that the number of rEP instructions amounts to one-third of the hard-to-predict instructions, across all services.

Storage Requirements: We determine the storage cost of a CHESS HP+rEP assuming: a 3350 entry buffer, that holds per entry five fields: i) a 9-bit pointer that points to a separate table that holds the unique branch PCs in the trace, ii) a 2-bit code of whether the entry is an indirect jump or a conditional and if conditional its predicted direction, iii) a 5-bit CSD, iv) a 12-bit pointer (log2 size of the trace) to the reconvergence point and v) a 7-bit pointer (log2 size of the unique targets table). The 512 entry PC table and 128 entry Target table hold in each entry a 48 bit address. The choice for specific sizes stems from analysis of the HP+rEP reference traces for the eight microservices that CHESS improved accuracy. We do not observe more than 3350 length, 32 CSD, 512 unique PCs and 128 unique targets for indirect branches. In total the size comes to 3350x(9+2+5+12+7)+(512+128)x48 = 18.1KB

Performance Analysis: Figure 14 compares the performance of CHESS HP-rEP against other schemes, showing that the prediction accuracy gains of CHESS translate to substantial performance improvements that are very close to those of a warm predictor.

Reference Trace Loading Time: We estimate the time required to bulk load a reference trace, assuming the loading process is fully serialized at the start of the service's execution. Loading the trace causes minimal performance overhead, ranging from 0.4% to 1.1%.

Profiling and Analysis Cost: Online profiling via Intel PT has low runtime overhead, less than 2% when tracing, which only occurs for a few seconds per machine, per day [19]. Each collected profile is under a megabyte after compression. Generating the reference trace through offline analysis requires an hour of processing time, which is amortized over performance benefits offered over multiple days of the same workload.

CFS Sources: Finally, we deep dive into our workloads programmatic structure to further understand the source and benefit of CFS. Since CHESS main benefit comes from predicting HP branches that exhibit CFS, we conduct a breakdown of HP branches across program modules divided into four main categories: (i) application main modules, which implement application logic, (ii) application libraries, which contain third-party application-specific libraries (*e.g.*, mlpack [7] and BLAS libraries in the Recommend-bucket service), (iii) system libraries, which contain basic execution modules of the system (*e.g.*, C standard library), and (iv) RPC, which contains

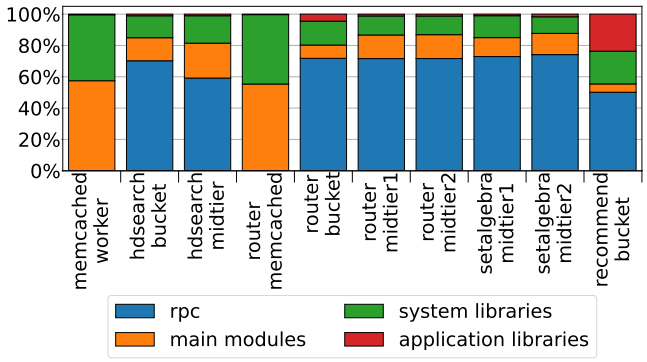


Figure 15: HP branches distributed across program modules.

libraries to support remote procedure calls between microservices (*e.g.*, Google RPC library [5]).

The breakdown in Figure 15 reveals that CFS is a result of both the microservices architecture and the application logic, with HP branches distributed across all four categories. Decomposing an application following the microservices paradigm naturally results in small services that can be independently developed, deployed, and scaled. Each service focuses on a small set of functionality, limiting the number of control flow paths a request can take within a service and increasing in this way CFS among requests. Moreover, request messages, such as RPCs, undergo the same network processing logic that is largely agnostic to message payload.

8 Related Work

Profile-Guided Optimization: Profile-guided optimization (PGO) is a compiler optimization technique where a program is optimized using feedback from runtime execution profiles. Due to its effectiveness in reducing front-end stalls PGO-optimized binaries are widely deployed in the datacenter [24, 47, 57]. Closest to our work, Thermometer [58] and Whisper [38] are PGO techniques that collect branch execution profiles using Intel PT to identify and optimize branch instructions causing frequent mispredictions. Thermometer [58] uses execution profiles to annotate static branches with two-bit static hints to guide BTB replacement, while Whisper [38] injects additional instructions to encode branch histories using boolean formulas to predict branch direction. Our comparison of CHESS+HP+rEP and CHESS-Whisper revealed the two schemes to provide similar accuracy for HP conditional branches. However, as the two schemes entail different overheads it is of interest for future work to compare their performance.

The fetch-static component of CHESS took inspiration from these efforts, but since the large majority of branches in our workloads follow one direction, injecting additional instructions is unnecessary. Instead, we use two-bit static hints to encode for statically predictable branches their predicted direction and for non-statically predictable which dynamic predictor to use (fetch or similarity).

Record and Replay: Our work draws inspiration from prior approaches that record and replay microarchitectural events for prediction. TIFS [30] records and replays recurring instruction cache block sequences, called temporal instruction streams, to predict instruction cache misses. Lukewarm [53] records and replays instruction accesses to warm up a cold instruction cache and minimize its impact on instruction cache accesses. Ignite [54] records and replays branch-target buffer (BTB) insertions to warm up a cold BTB together with a bimodal predictor and minimize their impact on branch predictions. Unlike these approaches, our proposal does not entirely replay a prior workload; instead, it replays only the convergent part of the control flow of the recorded workload, addressing challenges related to control-flow alignment requirements and the need for early branch predictions in the pipeline.

Control-Flow Integrity: Control flow integrity (CFI) thwarts control-hijacking attacks by ensuring that the control flow remains within the CFG intended by the program. Prior work leverages Intel PT for efficient online CFI enforcement [32, 34, 41]. They use Intel PT to collect runtime control flow traces and compare them with a statically derived CFG or control flow policy to detect control-flow violations at runtime. We similarly use Intel PT to collect control flow traces, but our objective is different. We utilize these traces not to enforce control flow, but to predict control flow and reconvergence.

Trace Caching: Trace-scheduling [31] and trace-caching [51] are seminal compiler and hardware efforts respectively aimed to leverage repetition in program control-flow paths.

9 Conclusion

Branch mispredictions due to cold resources in conventional history-based predictors undermine the performance of microservices with strict latency requirements. To address this, we propose Similarity-Based Branch Prediction (SBP), a hybrid predictor architecture that enhances conventional front-end predictors with a similarity component. SBP leverages the CFS across microservice requests by utilizing information from past execution traces encoded in a reference trace to predict branches. We develop CHESS, an instantiation of SBP combining conventional history-based, static-hint, and similarity predictors. CHESS judiciously applies similarity prediction to branches that are hard to predict through conventional prediction techniques, effectively mitigating cold-start effects while maintaining practical reference trace length. Our evaluation shows that CHESS improves branch misprediction ratio with a small additional on-chip storage space per core, delivering performance that is very close to a warm baseline system.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments on earlier versions of this manuscript. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101029391. The second and last authors are partially supported by an Intel academic research grant.

References

[1] Accessed April 2025. Accelerating cloud innovation with AWS Graviton4 processors, powered by Arm Neoverse. Online, https://newsroom.arm.com/blog/arm-aws-reinvent-2024.
[2] Accessed April 2025. AWS Graviton Processors. Online, https://aws.amazon.com/ec2/graviton/.
[3] Accessed April 2025. Boost Performance with Hardware Counter Assisted Profile Guided Optimization (HWPGO). Online, https://www.intel.com/content/www/us/en/developer/articles/technical/hwpgo.html.

[4] Accessed April 2025. CloudLab hardware infrastructure. Online, https://docs.cloudlab.us/hardware.html.

[5] Accessed April 2025. gRPC: A high performance, open source universal RPC framework. Online, https://grpc.io/.

[6] Accessed April 2025. Memcached: A Distributed Memory Object Caching System. Online https://memcached.org/.

[7] Accessed April 2025. mlpack: fast, header-only C++ machine learning library. Online, https://mlpack.org/.

[8] Accessed April 2025. perf-intel-pt - Support for Intel Processor Trace within perf tools. Online https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html.

[9] Accessed April 2025. Selecting and configuring an instruction set architecture for your Lambda function. Online, https://docs.aws.amazon.com/lambda/latest/dg/foundation-arch.html.

[10] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[11] Varun Agrawal, Mina Abbasi Dinani, Yuxuan Shui, Michael Ferdman, and Nima Honarmand. 2019. Massively Parallel Server Processors. *IEEE Computer Architecture Letters* 18, 1 (2019), 75–78. https://doi.org/10.1109/LCA.2019.2911287

[12] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[13] Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, and Haitham H. Akkary. 2007. Transparent control independence (TCI). In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. Association for Computing Machinery, New York, NY, USA, 448–459. https://doi.org/10.1145/1250662.1250717

[14] Frances E. Allen. 1970. Control flow analysis. *SIGPLAN Notices* 5, 7 (jul 1970), 1–19. https://doi.org/10.1145/390013.808479

[15] Georgia Antoniou, Davide Bartolini, Haris Volos, Marios Kleanthous, Zhe Wang, Kleovoulos Kalaitzidis, Tom Rollet, Ziwei Li, Onur Mutlu, Yiannakis Sazeides, and Jawad Haj Yahya. 2024. Agile C-states: A Core C-state Architecture for Latency Critical Applications Optimizing both Transition and Cold-Start Latency. *ACM Trans. Archit. Code Optim.* (July 2024). https://doi.org/10.1145/3674734 Just Accepted.

[16] Harold Aragon, Samuel Braganza, Edwin Boza, Jonathan Parrales, and Cristina Abad. 2019. Workload Characterization of a Software-as-a-Service Web Application Implemented with a Microservices Architecture. In *Companion Proceedings of The 2019 World Wide Web Conference* (San Francisco, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 746–750. https://doi.org/10.1145/3308560.3316466

[17] ARM. Accessed April 2025. Neoverse N1 makes debut in new AWS cloud instances. Online, https://newsroom.arm.com/news/neoverse-n1-makes-debut-in-new-aws-cloud-instances.

[18] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[19] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 462–473.

[20] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 645–650. https://www.usenix.org/conference/atc18/presentation/boucher

[21] Magnus Bruce. 2023. Arm Neoverse V2 platform: Leadership Performance and Power Efficiency for Next-Generation Cloud Computing, ML and HPC Workloads. In *2023 IEEE Hot Chips 35 Symposium (HCS)*. 1–25. https://doi.org/10.1109/HCS59251.2023.10254718

[22] Ioana Burcea and Andreas Moshovos. 2009. Phantom-BTB: a virtualized branch target buffer design. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/1508244.1508281

[23] Po-Yung Chang, Eric Hao, and Yale N. Patt. 1997. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 274–283. https://doi.org/10.1145/264107.264209

[24] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[25] Chih-Hsun Chou, Laxmi N. Bhuyan, and Daniel Wong. 2019. μDPM: Dynamic power management for the microsecond era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 120–132. https://doi.org/10.1109/HPCA.2019.00032

[26] J.D. Collins, D.M. Tullsen, and Hong Wang. 2004. Control Flow Optimization Via Dynamic Reconvergence Prediction. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. 129–140. https://doi.org/10.1109/MICRO.2004.13

[27] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. 2005. Performance implications of single thread migration on a chip multi-core. *SIGARCH Computer Architecture News* 33, 4 (nov 2005), 80–91. https://doi.org/10.1145/1105734.1105745

[28] Namiot Dmitry and Sneps-Sneppe Manfred. 2014. On Micro-Services Architecture. *INJOIT* 2, 9 (2014).

[29] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra". 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, USA, 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[30] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/MICRO.2008.4771774

[31] Joseph Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* C-30, 7 (1981), 478–490. https://doi.org/10.1109/TC.1981.1675827

[32] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 585–598. https://doi.org/10.1145/3037697.3037716

[33] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jiménez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR]

[34] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (Scottsdale, Arizona, USA) *(CODASPY '17)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/3029806.3029830

[35] Intel. Accessed April 2025. Intel Xeon Silver 4114 Processor. Online, https://intel.ly/3x7rx7N.

[36] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. 2000. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*, Andrew Wolfe and Michael S. Schlansker (Eds.). ACM/IEEE Computer Society, 67–76. https://doi.org/10.1109/MICRO.2000.898059

[37] Mahmoud Khairy, Ahmad Alawneh, Aaron Barnes, and Timothy G. Rogers. 2022. SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 441–463. https://doi.org/10.1109/MICRO56248.2022.00040

[38] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A. Jiménez, and Baris Kasikci. 2022. Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 19–34. https://doi.org/10.1109/MICRO56248.2022.00017

[39] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. https://doi.org/10.1145/2592798.2592821

[40] Chit-Kwan Lin and Stephen J. Tarsa. 2019. Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 228–238. https://doi.org/10.1109/IISWC47752.2019.9042108

[41] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–540. https://doi.org/10.1109/HPCA.2017.18

[42] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 159–169. https://doi.org/10.1109/IC2E.2018.00039

[43] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. https://doi.org/10.1145/3472883.3487003

[44] Mark Mansi and Michael M. Swift. 2020. ∅sim: Preparing System Software for a World with Terabyte-scale Memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 267–282. https://doi.org/10.1145/3373376.3378451

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[46] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[47] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 2–14.

[48] Irma Esmer Papazian. 2020. New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP). In *2020 IEEE Hot Chips 32 Symposium (HCS)*. 1–22. https://doi.org/10.1109/HCS49909.2020.9220434

[49] Andrea Pellegrini and Chris Abernathy. 2019. Arm Neoverse N1 Cloud-to-Edge Infrastructure SoCs. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–21. https://doi.org/10.1109/HOTCHIPS.2019.8875640

[50] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. 2021. Practical dynamic reconstruction of control flow graphs. *Software: Practice and Experience* 51, 2 (2021), 353–384. https://doi.org/10.1002/spe.2907 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2907

[51] E. Rotenberg, S. Bennett, and J.E. Smith. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 24–34. https://doi.org/10.1109/MICRO.1996.566447

[52] E. Rotenberg, Q. Jacobson, and J. Smith. 1999. A study of control independence in superscalar processors. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. 115–124. https://doi.org/10.1109/HPCA.1999.744346

[53] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm serverless functions: characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 757–770. https://doi.org/10.1145/3470496.3527390

[54] David Schall, Andreas Sandberg, and Boris Grot. 2023. Warming Up a Cold Front-End with Ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 254–267. https://doi.org/10.1145/3613424.3614258

[55] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. https://api.semanticscholar.org/CorpusID:113795057

[56] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

[57] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 617–631. https://doi.org/10.1145/3575693.3575727

[58] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-guided BTB replacement for data center applications. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 742–756. https://doi.org/10.1145/3470496.3527430

[59] Akshitha Sriraman and Thomas F. Wenisch. 2018. μ-Suite: A Benchmark Suite for Microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12. https://doi.org/10.1109/IISWC.2018.8573515

[60] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. https://doi.org/10.1145/3579371.3589069

[61] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2019. BRB: Mitigating Branch Predictor Side-Channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 466–477. https://doi.org/10.1109/HPCA.2019.00058

[62] Jawad Haj Yahya, Haris Volos, Davide B. Bartolini, Georgia Antoniou, Jeremie S. Kim, Zhe Wang, Kleovoulos Kalaitzidis, Tom Rollet, Zhirui Chen, Ye Geng, Onur Mutlu, and Yiannakis Sazeides. 2022. AgileWatts: An Energy-Efficient CPU Core Idle-State Architecture for Latency-Sensitive Server Applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 835–850. https://doi.org/10.1109/MICRO56248.2022.00063

[63] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

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

[65] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1155–1170. https://doi.org/10.1145/3453483.3454101