EXIST: Enabling Extremely Efficient Intra-Service Tracing Observability in Datacenters

Xinkai Wang
Shanghai Jiao Tong University
Shanghai, China
unbreakablewxk@sjtu.edu.cn

Xiaofeng Hou
Shanghai Jiao Tong University
Shanghai, China
hou-xf@cs.sjtu.edu.cn

Chao Li
Shanghai Jiao Tong University
Shanghai, China
lichao@cs.sjtu.edu.cn

Yuancheng Li
Shanghai Jiao Tong University
Shanghai, China
lyc1535180405@sjtu.edu.cn

Du Liu
Shanghai Jiao Tong University
Shanghai, China
sjtu_ld@sjtu.edu.cn

Guoyao Xu
Alibaba Group
Hangzhou, China
yao.xgy@alibaba-inc.com

Guodong Yang
Alibaba Group
Hangzhou, China
luren.ygd@alibaba-inc.com

Liping Zhang
Alibaba Group
Hangzhou, China
liping.z@alibaba-inc.com

Yuemin Wu
Alibaba Cloud
Hangzhou, China
yuemin.wym@alibaba-inc.com

Xiaopeng Yuan
Alibaba Cloud
Hangzhou, China
yxp377828@alibaba-inc.com

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

The complexity of online applications is rapidly increasing, bringing more sophisticated performance anomalies in today's cloud datacenter. To fully understand application behaviors, we should obtain both inter-service communication data via RPC-level tracing and intra-service execution traces via application-level tracing to precisely reason about event causality. However, the average time overhead of existing intra-service tracing schemes on the traced applications is generally about 5-10%, possibly reaching 18% in the worst case. To realize practical intra-service tracing in shared and stressed datacenters, one must achieve extreme tracing efficiency with an overhead at the per-mille level.

In this work, we present EXIST, an extremely efficient intra-service tracing system based on off-the-shelf hardware tracing capabilities. EXIST consists of three cooperative modules to pursue optimal trade-offs towards extremely low overhead. Firstly, it identifies and eliminates costly tracing control operations to guarantee the performance of the observed applications. Secondly, it allocates limited trace buffer space dynamically based on application status. Thirdly, it optimizes the trace coverage with cluster-level orchestration. We implement and evaluate EXIST on benchmark and real-world applications thoroughly. EXIST achieves 2-10× efficiency improvements compared to existing techniques and over 90% accuracy compared to exhaustive tracing reference. With extremely efficient intra-service tracing observability, we can achieve more explainable datacenter management.

CCS Concepts: • **Computer systems organization** → *Cloud computing*.

Keywords: Intra-Service Tracing; Observability; Facility Efficiency; Processor Trace; Shared Datacenter

ACM Reference Format:
Xinkai Wang, Xiaofeng Hou, Chao Li, Yuancheng Li, Du Liu, Guoyao Xu, Guodong Yang, Liping Zhang, Yuemin Wu, Xiaopeng Yuan, Quan Chen, and Minyi Guo. 2025. EXIST: Enabling Extremely Efficient Intra-Service Tracing Observability in Datacenters. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3676641.3716283

ASPLOS '25, Rotterdam, Netherlands.

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716283

1 Introduction

In modern datacenters, emerging intelligent software, like AI-powered recommendation [3] and conversation [57] applications, involve massive inner complexity and outer interactions that need to be understood for performance anomalies [58, 67]. It is challenging to maintain stable performance in production systems since various unscheduled

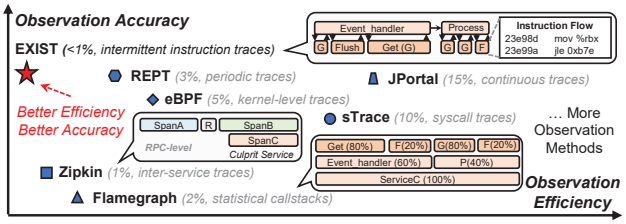


Figure 1. Efficiency and accuracy comparison of popular observation methods.

events can occur, such as hardware faults (e.g., fail-slow components [32]) and system traffic overload (e.g., metastable failures [34]). It forces cloud providers to devote significant engineering resources to explain the causality that led to, and recover from, the anomalies [69].

Understanding performance anomalies occurring on complex applications is challenging, owing to insufficient chronological runtime information. As shown in Figure 1, statistical observation methods like Flamegraph [31] mainly collect statistical metrics and sample periodically, termed *statistical observability* [26, 33]. They efficiently depict the statistics around the anomaly, which gets the average characteristics (e.g., proportions of functions *GFP*) but fails to infer causality. In contrast, *chronological observability* via tracing (i.e., tracing observability) can uncover the cause-effect of system events, which is crucial for explainable performance debugging and optimization [68]. Currently, efficient inter-service tracing like Zipkin [100] that captures RPC-level communication (e.g., execution time of services *ABC*) has been widely deployed to locate culprit services [26, 42, 79].

Faced with more complex applications, intra-service tracing (IST) is expected to dig deeper into application-level causality. IST collects chronological function and instruction flows within a single service. Figure 1 depicts a culprit example, where function *Get* after *Flush* takes abnormally long and blocks the program. With chronological intra-service traces, we can pinpoint the anomalous execution sequence instead of just observing the abnormal execution time of the culprit service or the identical function distributions. Current IST methods mainly rely on kernel and user instrumentation, which embed runtime tracepoints to produce chronological traces [1, 49, 66, 68, 88]. Admittedly, they are widely used by software developers for performance debugging, but they intrude software execution to obtain per-application traces with varied accuracy and efficiency [68].

To date, achieving such intra-service tracing observability in datacenters faces two efficiency challenges. Firstly, shared execution environments degrade intra-service tracing efficiency. Tracing applications co-located on the same hardware [62, 81] often induces higher overhead than tracing applications running alone because it requires additional costly kernel operations. Secondly, stressed execution environments further worsen the negative effect of intra-service tracing overhead. During periods of resource saturation when most performance anomalies occur [96, 99], previously tolerable *single-digit-range* (i.e., 1-10%) overhead of popular tracing tools [1, 49] can significantly slowdown the traced applications. Therefore, we argue that intra-service tracing requires a transition to *per-mille level* (i.e., $\leq 1\%$) overhead for large-scale deployment.

In addition to the pursuit of per-mille level efficiency, it is crucial for IST system to balance space overhead and data coverage. Emerging processors provide low-overhead and high-accuracy hardware tracing capabilities [6, 37, 55], but there is an inherent trade-off between time efficiency, space overhead, and data coverage in hardware tracing abstractions. Prior works that leverage hardware tracing for various downstream tasks, such as REPT [19, 28] for reverse debugging and JPortal [102] for intra-service tracing, all fall short of extreme time efficiency as shown in Figure 1. To build a successful intra-service tracing system, we not only should pursue per-mille level time efficiency but also better utilization of the limited memory space for trace buffers and better tracing data coverage.

In this work, we propose **EXIST**, an **E**xtremely **E**fficient **I**ntra-**S**ervice **T**racing system that exploits off-the-shelf hardware tracing and targets shared and stressed clusters. At the node level, EXIST is designed for extreme intra-service tracing time efficiency while also optimizing space overhead and data coverage. At the cluster level, EXIST is flexibly orchestrated in a cloud-native manner and easily accessed through a configuration interface. The collected instruction traces are automatically synthesized into human-readable application behaviors for on-call engineers and developers, enabling the deployment in large-scale clusters.

More specifically, EXIST comprises three cooperative components for intra-service tracing. Firstly, the *tracing controller* extends the system kernel to eliminate the critical control operations of hardware tracers that cause performance slowdown. It captures the just-in-time scheduled processors to minimize costly register operations in conventional designs, reducing them from the number of context switches to the number of processing cores during the tracing period. Secondly, the *memory allocator* fully utilizes the limited memory space to store more useful traces by adjusting the traced core-set and the per-core buffer based on application usage information. Thirdly, the *coverage optimizer* extends the trace data coverage by selecting proper tracing periods and repetitions in the cluster to maximize tracing cost-efficiency.

To evaluate EXIST thoroughly, we implement and deploy it in Alibaba clusters, tracing both standard benchmarks and real-world cloud applications across diverse scenarios. Our results demonstrate significant improvements in tracing efficiency by ten times compared to current tracing methods.

Also, EXIST induces negligible degradation in the performance of the monitored applications. Furthermore, EXIST achieves over 90.2% tracing accuracy across various applications. EXIST greatly enhances our insights into clusters, and we provide a realistic case study based on EXIST for understanding real-world applications.

In summary, this paper makes the following contributions:

- **Analysis**: We analyze the importance of intra-service tracing observability and the extreme efficiency challenges in shared and stressed datacenters towards permille level intra-service tracing system.
- **Design**: We introduce a novel intra-service tracing system, EXIST, that exploits existing hardware capabilities to support better observability, optimizing the ternary design considerations in realistic clusters.
- **Evaluation**: We implement and deploy EXIST for evaluations using both benchmarks and real-world applications. We prove the efficiency and performance improvements over existing methods.

In the rest of the paper, Section 2 introduces the motivations. Section 3 presents EXIST's design in detail. Section 4 introduces our implementation. Section 5 evaluates EXIST thoroughly. Section 6 discusses the limitations and future works. Section 7 summarizes related work. Section 8 concludes the paper.

2 Background and Motivation

This section first illustrates the urgent need for intra-service tracing observability and further demystifies the challenges in a shared and stressed cluster.

2.1 The Need for Intra-Service Tracing Observability

Explainable management of emerging software needs chronological diagnostic information. The increase in software complexity and aggressive co-location management make it challenging to understand both the inter-service RPC-level (Remote Procedure Call) communication [26, 67, 85] and the intra-service application-level executions [58, 62]. Making it even more challenging is the fact that the scheduled execution can easily be disturbed by unscheduled events (like metastable traffic peaks [34] and fail-slow hardware faults [32]) and resource interference (like CPU pipeline contention [91] and power throttling [84]) as shown in Figure 2. Such anomalies cause frequent performance degradation in datacenters, requiring on-call engineers to explain them and recover from them [33].

To understand and mitigate performance anomalies, cloud providers should have better *observability* (a measure of how well components' internal states can be inferred from their external outputs [36, 43]) of the served applications. Figure 2 shows two typical tasks in realistic clusters: latency-critical task *A* performs AI-powered recommendation, and best-effort task *B* performs distributed caching. We co-locate

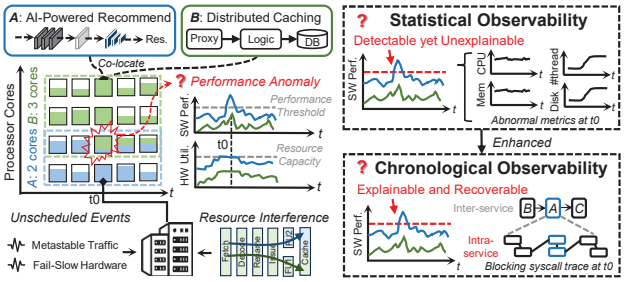


Figure 2. Intra-service tracing observability can better explain metric anomalies for performance debugging.

A and *B* to share four cores based on historical usage. At t_0, we detect metric anomalies (in red) via both an abnormal performance indicator (e.g., response time) [45] and an architectural indicator (e.g., utilization) [65]. However, it is difficult to explain the anomaly based on the abnormal multi-dimensional metrics via *statistical observability* in Figure 2.

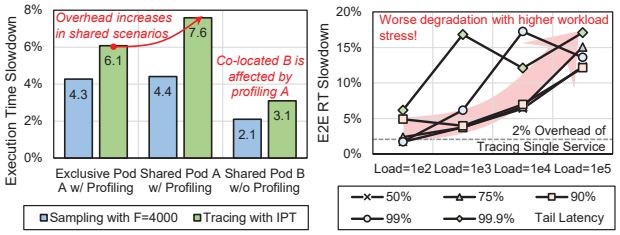
Accordingly, a tracing facility that provides chronological information is required for explainable performance debugging. Conventional statistical observability mainly focuses on the average characteristics and is blind to microsecond-level execution causality [68, 88]. In contrast, tracing observability enhances observation accuracy with chronological execution details. Figure 2 exemplifies the trace of culprit task *A* in the anomaly period. We can get the inter-service *B-A-C* causality via distributed tracing [42, 79]. To diagnose the culprit service *A*, we can get intra-service insights into the software and hardware [26, 75], which helps us find that a blocking syscall at t_0 causes the anomaly. Tracing observability significantly enhances site-reliability engineering [8] and profile-guided optimization [12, 51].

***Summary**: Future explainable datacenter management needs intra-service tracing observability to achieve a full understanding of complex performance anomalies.*

2.2 The Challenges of Extreme Efficiency

Efficiency is the primary challenge of intra-service tracing. One must minimize the negative performance impact on the traced applications. Unlike prior studies [70, 77, 86], we argue that a single-digit-range tracing overhead should be optimized in shared and stressed datacenters.

Firstly, current datacenters co-locate different tasks on the same hardware to improve resource utilization [62, 81], increasing the need for intra-service tracing efficiency. We observe that intra-service tracing incurs a larger overhead in a shared execution environment. As shown in Figure 3a, we select two applications in the SPECCPU 2017 benchmark [11] to run concurrently on the same cores. *A* (620.omnetpp) is profiled with *Perf*, and *B* (657.xz) is not profiled while running. We have two key findings. 1) By comparing the first



(a) Tracing in shared scenarios **(b)** Tracing in stressed scenarios

Figure 3. Single-digit-range intra-service tracing overhead affects performance greatly in shared and stressed clusters.

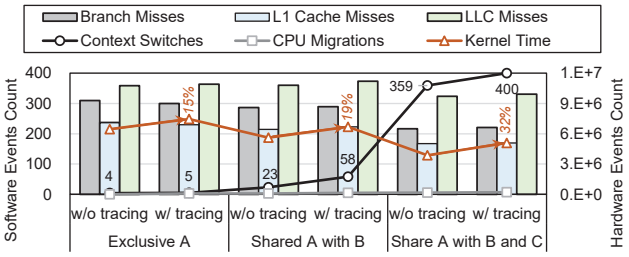


Figure 4. Software and hardware analysis of increasing tracing overheads in multi-application scenarios.

two groups of bars, we can see that the intra-service profiling overhead increases under shared scenarios, especially when tracing is on (green bars). (2) If we look at the last two groups of bars, it shows that the tracing overhead affects the co-located innocent applications, inducing cascaded performance degradation. The above overhead is much worse in real-world clusters with more aggressive co-location [81].

To further analyze tracing overheads in multi-application scenarios, we investigate the tracing process from software and hardware perspectives in Figure 4. We monitor key software events (context switches, CPU migrations, and kernel time) and hardware events (branch misses, L1 cache misses, and last-level cache (LLC) misses) via Perf [59]. Besides co-locating 620.omnetpp (*A*) with 657.xz (*B*), we further co-locate them with Mysql [22] driven by Sysbench [52] (*C*) to analyze the impact of co-location density. We find that the number of context switches increases greatly in multi-application scenarios, and tracing control operations at every context switch contributes to the increase in overhead. Moreover, conventional tracing causes an increase in kernel time and the phenomenon is worse in multi-application scenarios due to complex tracing operations. As for hardware events, co-location greatly affects the three metrics due to resource interference, but the tracing facility only causes a slight increase (1.3%) in LLC misses.

In addition, today's datacenter is a stressed execution environment that could further increase intra-service tracing overhead. Resource saturation is not unusual, and servers

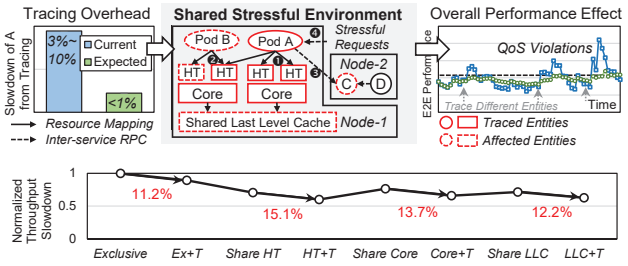


Figure 5. Intra-service tracing overheads in shared and stressed scenarios produce overall performance degradation. *X+T* means performance with tracing under current setting *X*. The experimental settings on the left include the shared resources on the right.

are often running close to their resource/performance limits (e.g., CPU utilization ≥ 90%). Such scenarios result from various causes like resource overcommitment [65], frequent full-link stress tests [99], and extreme traffic peaks [96]. There are more quality-of-service (QoS) violations than daily low-utilization situations since increased performance jitters can cause severe cascading effects. Therefore, stressed clusters are more vulnerable to intra-service tracing disturbances. As shown in Figure 3b, we mimicked a stressed execution environment with open-sourced benchmark *DeathStarBench* [25] and use *Perf* [59] to trace the *ComposePost* service. Under high load, even a single-digit-range intra-service profiling overhead (~2%) on a single service could induce more than 10% end-to-end response time degradation.

Consequently, the seemingly tolerable intra-service tracing overhead would result in significant performance degradation. However, the observed degradation would be negligible if the tracing overhead is reduced to per-mille level. Figure 5 illustrates four main factors in realistic clusters. ① The traced pod (the smallest unit of application) [54] *A* on logical cores are affected first. ② The interference on shared resources like physical cores affects the co-located pod *B*. ③ The successor pod *C* needs to wait longer for a single RPC from the affected pod *A*, and the number of RPCs between pod *A* and *C* in a single request can be tens of times to finish functionality [67], exacerbating the end-to-end performance impact. ④ The periodic outer traffic stress aggravates the shared clusters into chaotic environments [9], resulting in irregular QoS violation.

Further, we experimentally isolated the multiplexed resources in shared environments. We analyzed the impact of three key hardware resources (HT, core, and LLC) based on resource partitioning studies [14]. The throughput performance of Mysql with and without tracing was measured. We found that no specific hardware resource contributed significantly to the increased tracing overhead in multi-application scenarios. Shared LLC, core, and HT contribute to 1%, 1.5%, and 1.4% throughput slowdown, respectively.

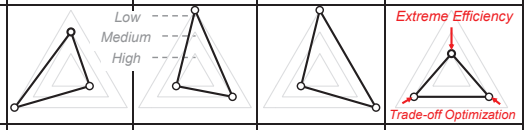
Objectives	Debugging REPT [25]	Security Griffin [23]	Tracing Jportal [94]	Tracing Our Work
Design Trade-offs				
Time Efficiency	5.35% (Avg) 9.68% (Worst)	4.8% (Avg) 18% (Worst)	11.3% (Avg) 16.5% (Worst)	< 0.5% (Avg) < 2% (Worst)
Space Overhead	1e-2 MB	1e2 MB	1e4 MB	1e3 MB
Data Coverage	Microseconds ~ Millisecond	Constant	Hours	Milliseconds ~ Seconds

Figure 6. Design considerations for intra-service tracing abstractions. Underlined are the best ones.

***Summary**: Intra-service tracing observability for shared and stressed clusters requires an overhead at the per-mille level to eliminate negative effect on the monitored applications.*

2.3 Inherent Trade-offs of Extreme Efficiency

Given the extreme efficiency challenges, the emerging hardware tracing capabilities of recent CPUs present a new opportunity. It enables instruction-level tracing without intrusion into the normal execution of workloads [6, 37, 55]. The detailed workflow of typical Intel mechanisms is specified in Section 6.1. To utilize hardware tracing, researchers have built abstractions for various downstream tasks like reverse debugging [19, 28], security enhancement [27, 60], and program tracing [102]. However, current system abstractions of hardware tracing capabilities are far from the per-mille level efficiency required in realistic clusters since it faces a conflicting three-dimensional design trade-off.

1. *Better Time Efficiency* means little negative effect on the performance of the observed applications.
2. *Better Space Overhead* means smaller physical memory occupation to store trace packets.
3. *Better Data Coverage* means larger intra-service trace spans on an individual worker.

Due to hardware limitations, all three dimensions cannot achieve the optima concurrently, hence some need to be sacrificed. As shown in Figure 6, designs for reverse debugging [19, 28] (the first column) sacrifice time and data for minimal space overhead. They use small circular buffers (e.g., 64KB per thread) to record the microsecond-scale traces just before the failure and applies frequent tracer operations to manage the tiny buffer. Meanwhile, designs for security enhancement [27] (the second column) sacrifice time overhead for better space overhead and data coverage. They manipulate tracers at each context switch and dump data every time the tiny buffer gets full. Worse, conventional designs for tracing [73, 102] (the third column) aim for full execution tracking and sacrifice time and space overhead for maximized data coverage, contradicting the efficiency requirements.

The gap between the low-overhead of hardware tracing and per-mille level efficient tracing systems if mainly caused by the control operations. During tracing, users need to manipulate specific Model-Specific Registers (MSR) to control the trace precision, target process, and buffer destination. Worse, tracing control must be done with tracing disabled [37], so every control needs to: (1) disable tracing, (2) modify the settings, and (3) enable tracing, which interrupts the execution of the traced application. Meanwhile, tracing control may incur costly switches between user and kernel mode if the trace settings rely on user-level information. Overall, reducing the complexity of runtime control is crucial to achieving a per-mille level efficient tracing facility.

We seek to provide appropriate abstractions for efficient intra-service tracing. Firstly, extreme time efficiency at the per-mille level causes little effect on the monitored applications. Further, it is important to ensure high performance under the given resource limits. For space overhead, it is not critical to keep kilobytes-scale memory occupation. In practice, node-level facilities occupying around 1% memory are tolerable for deployment, typically 1e3 MB for our servers. Flexibly organizing the limited memory space based on application information is preferable for prioritizing the time efficiency. In compensation, the intra-service tracing span reduces to orders of milliseconds, which is sufficient for performance debugging. As for software profiling demanding extended coverage, we can utilize multiple trace repetitions in the datacenter to obtain the complete profile.

***Summary**: Building the right abstractions of hardware tracing capability is attractive for intra-service tracing but requires a careful design for extreme efficiency.*

3 Design of EXIST

To fulfill the intra-service tracing void, we propose **EXIST**, an **E**xtremely **E**fficient **I**ntra-**S**ervice **T**racing system in large-scale clusters. This section describes the technical details of EXIST which allows intra-service tracing with extreme efficiency and balanced space overhead and data coverage.

3.1 Design Overview of EXIST

In Figure 7, we give an overview of EXIST that contains three cooperative components for our pursuit of a three-dimensional design optima.

1. **Operation-aware Tracing Controller** (OTC) extends the operating system and optimizes intra-service tracing for better time efficiency. It reduces the number of critical control operations of the tracing facility from the number of context switches to the number of processing cores during the tracing period to minimize performance slowdowns.
2. **Usage-aware Memory Allocator** (UMA) cooperates with OTC at the node level to fully utilize memory space for storing more useful traces. Faced with space

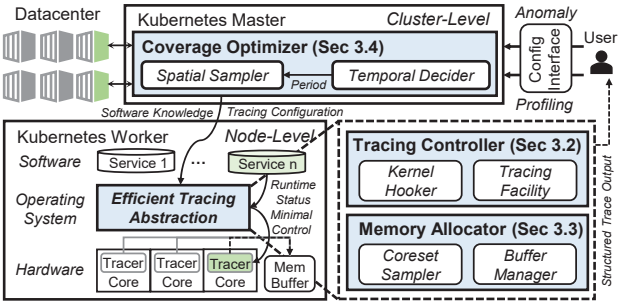


Figure 7. Design overview of EXIST. The solid lines are control paths and the dotted lines are data paths.

constraints, it adjusts the traced set of processor cores, dynamically allocates buffers for them, and adjusts the buffer settings for each core based on runtime application usage information.

3. **Repetition-aware Coverage Optimizer** (RCO) works at the cluster level to orchestrate the intra-service tracing. It pursues better data coverage and cluster-wide cost-efficiency. Given the high costs and marginal benefits of exhaustive tracing on all servers, we designed a two-dimensional temporal-spatial sampling mechanism to select appropriate tracing repetitions to optimize the trace data coverage.

Notably, EXIST is triggered on demand via an easy-to-use interface on a user request or when abnormal metrics are detected. EXIST collects the change of control flow trace packets at every branch and timing information at runtime. Afterward, EXIST uses an off-the-shelf software decoder [39] to reconstruct the execution flow from the trace packets and returns the human-readable application traces to users for anomaly analysis. EXIST focuses not only on putting existing hardware tracing into use but also on how to construct an efficient intra-service tracing facility to support better observability in realistic datacenters.

3.2 Operation-Aware Tracing Controller

To pursue per-mille level efficiency, we identify the critical register switching operations that affect the execution of the monitored applications. Compared to conventional heavy control, OTC adopts lightweight control by reducing costly operations and proactively limiting the tracing process. The conventional paradigm

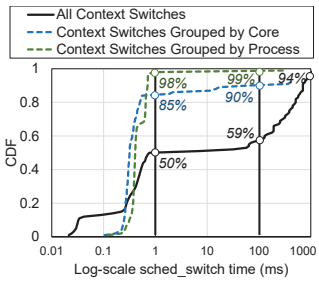


Figure 8. Distributions of context switch periods.

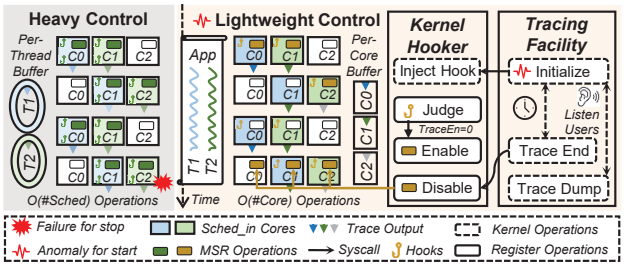


Figure 9. Details of operation-aware tracing controller.

generates traces continuously into a circular per-thread buffer and relies on signals like failures to stop tracing [19, 27, 28]. It requires modifying the control MSRs at every context switch to adjust the trace settings, producing redundant costly operations. Figure 8 presents the Cumulative Distribution Function (CDF) of context switch periods collected by eBPF [1] on a realistic server. Most cores and threads experience a context switch in under less than 1ms, meaning the conventional schemes cause 1000 times more operations compared with an order of seconds control period. Context switches of some processes on few cores are abnormally longer, hence distribution of all switches are higher than those grouped by core/process. Such designs also raise the stability risks of clusters since frequent unsafe MSR modifications may cause fail-stop servers [40].

OTC eliminates the unnecessary operations in tracing control as shown in Figure 9. Firstly, the *tracing facility* receives an anomaly request to initialize tracing and passes the tracing configurations, including the coreset and period, to the kernel hooker. The tracing facility proactively monitors the tracing process and uses a high-resolution timer (HRT) to limit the tracing period by terminating the tracing when the HRT expires. It then transfers the trace packets for analysis based on user requests.

Secondly, the *kernel hooker* injects hooks in sched_switch tracepoint to manipulate tracers. When the targeted process is scheduled onto core C, the hook enables the tracer of C by setting the *TraceEn* bit of the control MSR *IA32_RIIT_CTL* [37]. However, the hook does not manipulate tracing when scheduled out and naturally eliminates the unnecessary operation when scheduled in again. Data collection of unrelated threads are prohibited by process filter mechanism. Such a design reduces costly interrupts from the number of context switches to the number of processing cores, minimizing runtime overhead. Moreover, OTC operates purely at kernel mode to reduce additional overhead of user/kernel mode switch. At the end of tracing, the *kernel hooker* disables the tracers of all scheduled cores by clearing the *TraceEn* bit, which prevents infinite tracing and improves system robustness. Note that although on-demand control of tracing has been explored in prior works [42, 50, 63], EXIST innovates in reducing redundant control operations for extreme efficiency.

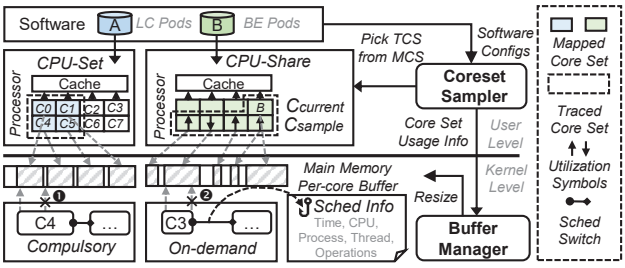


Figure 10. Design details of usage-aware memory allocator.

3.3 Usage-Aware Memory Allocator

To fully utilize the limited memory space, EXIST manages the physical memory with two components: *coreset sampler* and *buffer manager*, as shown in Figure 10. The former executes at the user level to select the traced coreset, and the latter executes at the kernel level to manage buffers.

Firstly, the *coreset sampler* uses application metadata to determine the set of traced cores. Hardware tracers require allocating fixed memory space for the processor core before tracing begins. Allocating a maximum per-core buffer (128 MB) to every processor core (128 cores) causes a great waste of host memory (16 GB). Figure 11 shows the memory usage on a typical server. Although the memory utilization is relatively low, the allocated memory almost reaches the ceiling [65, 83]. Given the limited memory for the tracing facility (0.5-1 GB), allocating them to all cores equally reduces the size of the per-core buffer and the trace information gathered. Thus, we must carefully allocate memory space for the cores that are actually executing.

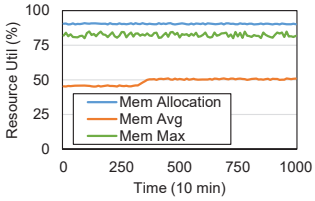


Figure 11. Host memory allocation and utilization rates.

Current software has two CPU provisioning modes: CPU-set mode maps the application to several CPU cores exclusively, while CPU-share mode maps the application to a large shared coreset [81]. We call the provisioned cores Mapped Core Set (MCS) and potential traced cores Traced Core Set (TCS). EXIST gets the software metadata at the user level. To minimize memory overhead, EXIST devises respective settings for the two provisioning modes. For CPU-set applications, MCS equals TCS, and we allocate memory buffers to the entire MCS equally. The status of the current node determines the buffer size for each core. For CPU-share applications, we design a core sampling mechanism to pick TCS selectively. The TCS includes the current core and randomly selected cores with varied utilization. Empirically, the low-utilization cores are more likely to be scheduled in and are assigned bigger buffers. The buffer settings are determined by the system status at the time OTC initializes tracing.

Secondly, EXIST assigns a cache-bypass memory buffer to each traced core instead of each traced thread, since we desire to reduce the frequency of MSR operations. Admittedly, per-thread buffer schemes [19, 27] isolate memory areas from the software's point of view. However, it requires MSR operations at every context switch because hardware tracing capability can only change the buffer base address with tracing disabled [37]. Moreover, we choose to adopt *compulsory tracing* instead of the conventional ring-buffer [19, 73], that is, ① the extra data is dropped when the buffer is full. This enables us to record more related traces closer to the performance anomaly and keep the amount of memory used under control. To keep the content of the process of interest, ② we adopt an *on-demand tracing* policy that stops tracing when the targeted thread schedules out based on CR3 filter [37], ensuring that EXIST causes no effect on the unrelated processes. To reason about the dependency across threads for multi-threaded applications with per-core settings, the hook injected in the sched_switch tracepoint records the information of context switches in a five-tuple [*Timestamp, CPUID, ProcessID, ThreadID, Operation*], and each context switch produces a 24-byte array to assist in multi-thread tracing.

3.4 Repetition-Aware Coverage Optimizer

Besides the node-level time and space optimizations, we also need to orchestrate intra-service tracing in clusters and enhance data coverage. Exhaustively tracing all identical workloads of interests on all nodes (repetitions) of all time results in high costs and marginal benefits since applications behave similarly without performance anomalies [77]. As shown in Figure 12, with the increase of trace repetitions, linearly growing tracing costs bring better trace coverage. A large proportion of application behaviors are identical so tracing multiple repetitions has diminishing and marginal benefits. Therefore, we want to select the tracing repetitions that are truly needed for reducing the cost of software profiling.

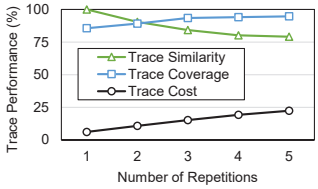


Figure 12. Performance of tracing multiple repetitions.

Firstly, given a tracing request on a specific application, the temporal decider selects suitable tracing periods based on the complexity of the target application, and more complex programs require more extended tracing periods to cover their execution. In practice, we adopt the weighted sum of three factors in application complexity measurement: priority pre-defined in the manager, size of the binary file, and the number of previous stability issues. Moreover, we measure

the reference monitoring overhead on the applications in advance to jointly decide the proper tracing settings.

Secondly, the spatial sampler selects a fraction of repetitions as tracing entities. For performance anomalies, we trace all the entities involved since the abnormal behaviors are distinct. For software profiling, we select repetitions based on deployment density and application priority, where higher-priority and broader-deployed applications are traced more. Also, we set a deployment threshold to guarantee necessary observation on applications deployed only once. For trace repetitions, we remove redundancy and complement the missing traces on each node. Although the sampling-based tracing entity selection has been used in previous works to reduce overheads [70, 77], EXIST innovates in considering software online and offline data to make just-right decisions.

4 Implementation and Deployment

We implement EXIST in realistic clusters. Our design mainly spans two system layers, namely node-level OS kernel and the cluster-level cloud manager. The former is implemented with roughly 14K LoCs in C, and the latter is implemented with roughly 3K LoCs in Go.

For node-level tracing, we set the *TraceEn* and *BranchEn* bits to enable Change of Flow Instruction tracing. We also set the *CYCEn* bit to enable cycle-accurate tracing for IPC computation. Further, we set the *CR3Filter* bit to enable filtering the process of interests. For buffer management, we set the *ToPA* bit to enable the Table of Physical Address (ToPA) output mechanism, which links variable-sized regions of memory with tables of pointers into one destination. The *STOP* bit of the last entry of ToPA is set. Other control bits are left as default [37]. As for hyperparameters, we allow for 5e2 memory space occupation for intra-service tracing and the per-core buffer size ranges from 4MB to 128MB. Empirically, the tracing period ranges from 0.1 to 2 seconds. We provide default tracing options but also support personalized tracing where users can adjust the configurations.

Further, we integrate EXIST into the datacenter management in a cloud-native manner. Developers and engineers can use EXIST through a unified interface. As for the control flow, user requests and tracing configurations are encapsulated as Custom Resource Definitions (CRD) [53] in the Kubernetes API server. We implement separate controllers for each CRD to finish the reconciliation process. As for the data flow, the traced data is uploaded to the unstructured object storage service (OSS) [18] directly instead of storing locally to reduce memory and file I/O overheads. We use an off-the-shelf software decoder [39] to reconstruct the control flow. The software decoder gets traces from OSS and program binaries from the binary repository. After decoding, it outputs the results to the structured open data processing service (ODPS) [17] for persistent storage, which could be queried easily by all users for analysis and reproduction.

We aim to implement a widely applicable facility for intra-service tracing in large-scale clusters. First, software-level instrumentation is portable but hard to apply to all the applications of interest. Alibaba clusters currently accommodate more than ten thousand applications, which are built upon 10+ programming languages and various development frameworks. The scale and diversity make it challenging to apply well-crafted but customized instrumentation methods for intra-service tracing observability [66, 72]. Although one can instrument the shared underlying operating systems to obtain kernel-level execution traces with eBPF tools [1, 41], the more vital user-level execution traces are still black boxes to be observed. Second, hardware-level domain-specific designs are promising but fail to deploy in realistic clusters. Alibaba clusters contain more than 100 thousand nodes with 28 million CPU cores across seven generations of Intel X86 processors, and therefore our tracing systems should be compatible with off-the-shelf hardware rather than customized hardware support [29, 30]. Therefore, we implement EXIST without additional support from software programmers and hardware vendors to enable its large-scale deployment.

5 Evaluations

Our evaluation wants to answer three questions:

1. How efficient is EXIST and what are its effects on the observed workloads? (§ 5.2)
2. How accurate is EXIST in different scenarios? (§ 5.3)
3. How can EXIST support tracing observability in realistic scenarios? (§ 5.4)

5.1 Evaluation Methodology

Platforms: We implement EXIST as described in Section 4 and evaluate it in two environments. One environment is used for offline experiments on standard benchmark suites. For this, we use two nodes with dual-socket 32-core Intel Xeon Platinum 8369B CPU, IceLake architecture, 1TB memory, and Linux 4.19.91 system. The other environment runs online business services, allowing us to evaluate the performance of EXIST in realistic scenarios. We select ten nodes with dual-socket 24-core Intel Xeon Platinum 8163 CPU, SkyLake architecture, 384GB memory, and Linux 5.10.112 system. Moreover, the end-to-end metrics (e.g., execution time, throughput) are obtained from application outputs, and the runtime metrics (e.g., user/sys utilization, memory usage) are obtained via the existing monitoring system [81]. The evaluation metrics presented are averages of repeated experiments on diverse nodes to reduce the result biases.
Workloads: We use both standard benchmarks and production applications to evaluate EXIST as shown in Table 1. On the one hand, we use SPEC CPU 2017 Integer benchmarks [11] for two reasons. Firstly, they can be easily traced

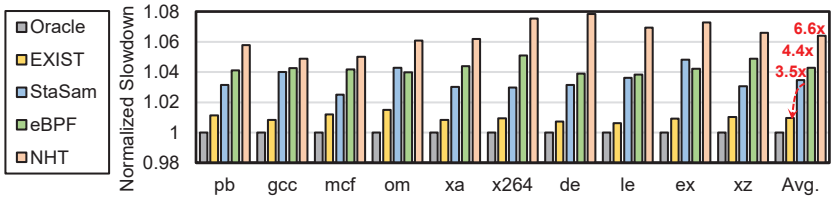


Figure 13. EXIST induces per-mille level tracing overhead on compute-intensive benchmarks compared to baselines.

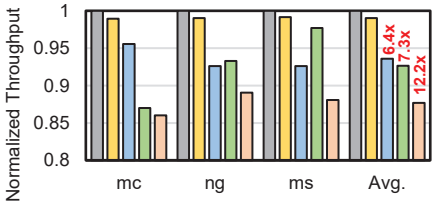


Figure 14. Comparison of tracing overhead on online benchmarks.

Table 1. Evaluated compute benchmarks, online benchmarks, and real-world applications.

Name	Applications	Descriptions
pb	600.perlbench_s	Perl interpreter
gcc	602.gcc_s	GNU C compiler
mcf	605.mcf_s	Route planning
om	620.omnetpp_s	Discrete Event simulation
xa	623.xalancbmk_s	XML to HTML conversion
x264	625.x264_s	Video compression
de	631.deepsjeng_s	Alpha-beta tree search
le	641.leela_s	Monte Carlo tree search
ex	648.exchange2_s	Recursive solution generator
xz	657.xz_s	General data compression
mc	Memcached	In-memory cache
ng	Nginx	Web server
ms	Mysql	Online database
Search1	Latency-sensitive CPU-set search engine service	
Search2	Latency-sensitive CPU-share search engine service	
Cache	Best-effort memory graph caching service	
Pred	ML-based click-through rate prediction service	
Agent	Node-level SLO management service	

offline for validation. Secondly, they represent compute-intensive scenarios to validate EXIST in high-utilization scenarios [13]. Also, we use three online benchmarks: (1)Memcached [24] for in-memory caches; (2) Nginx [76] for web servers; (3) Mysql [22] for online databases. For Memcached, we use the Memtier benchmark [56] to simulate ten concurrent clients with 1:1 set-to-get ratio. For Nginx, we use the Apache benchmark [5] to simulate ten concurrent clients constantly sending 20K requests, each of them requesting one 20B-sized file. For Mysql, we use Sysbench [52] to simulate read-write requests on ten 1M-sized tables. On the other hand, we select five real-world cloud applications from the Alibaba e-commerce pipeline: (1&2) two latency-sensitive search services based on the Havenask engine [2] with different CPU-provision modes; (3) a best-effort caching service based on iGraph [16]; (4) an ML-based prediction service based on the RTP engine [74]; (5) a node-level management facility for guaranteeing service-level objectives.

Baselines: To understand the performance of EXIST, we compare it with three state-of-the-practice baselines as shown in Table 2. Tracing systems are turned on for the entire experiments. The first, *StaSam*, uses a non-chronological sampling method and records statistical events with a default frequency of 4K [59]. The second, *eBPF*, is eBPF-based tracing method driven by the user-level bpftrace tool, which records the sys_enter events [1]. The third, *NHT*, uses native hardware tracing , enabled by the open-sourced Perf [73]. These three baselines represent popular monitoring methods in datacenters. Due to orthogonal design objectives and the public unavailability of conventional hardware tracing designs [19, 28, 60, 102], it is hard for us to perform a fair reproduction in our clusters, so we generally compare against them in our analysis using the results in their papers.

Table 2. Baselines used for comparison

Scheme	Descriptions	Usage
Oracle	Normal execution w/o tracing	runcpu intspeed
StaSam	Statistical Sampling	perf record -a -F 3999
eBPF	eBPF-based Tracing	bpftrace -e "sys_enter"
NHT	Native Hardware Tracing	perf record -e intel_pt
SHT	SOTA Hardware Tracing	Results in their papers

5.2 Efficiency

To answer the first question, we evaluate EXIST's time efficiency by comparing it against three state-of-the-practice methods and eight state-of-the-art (SOTA) schemes.

Time Efficiency on Benchmarks: Tracing SPECCPU applications causes execution time slowdown, which reflects the performance impact. We normalize the time with tracing by the time without tracing (Oracle). As shown in Figure 13, the slowdown of EXIST ranges from 0.4% to 1.5% for the entire SPECCPU benchmark. On average, EXIST reduces time overhead by 3.5x, 4.4x, and 6.6x over the *StaSam*, *eBPF*, and *NHT* baselines, respectively. As for the online benchmarks, we compare the time efficiency using throughput metrics in Figure 14. The results show that EXIST reduces tracing overhead by 6.4x, 7.3x, and 12.2x over the three baselines, respectively. We find that online benchmarks are more sensitive to tracing compared to compute-intensive benchmarks since tracing disturbances cause cascaded slowdowns of subsequent queries in such applications. EXIST achieves 1.1% overhead with minimal intrusion into applications.

Table 3. Time efficiency comparison with SOTA results. *c* and *o* represent results on compute and online benchmarks.

Schemes	Hardware tracing-based methods				Instrumentation-based tracing methods				Our results	
	REPT[28],*o*	FlowGuard[60],*c*	Upgradvisor[21],*c*	JPortal[102],*o*	Log20[98],*o*	Hubble[68],*c*	DMon[50],*o*	Argus[88],*o*	EXIST,*c*	EXIST,*o*
Average	5.35%	3.79%	6.4%	11.3%	-0.2%	5%	1.36%	3.36%	**0.9%**	**1.1%**
Worst	9.68%	30%	16%	16.5%	0.9%	25%	4.92%	5%	**1.5%**	**1.6%**

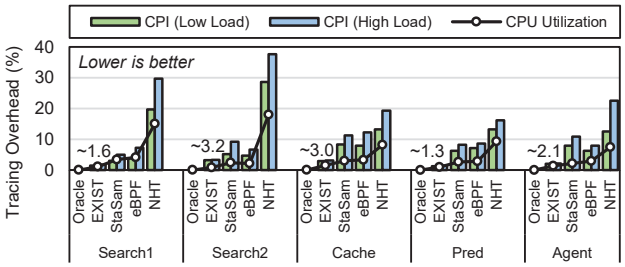


Figure 15. EXIST induces little tracing overhead on cloud applications under stressed scenarios.

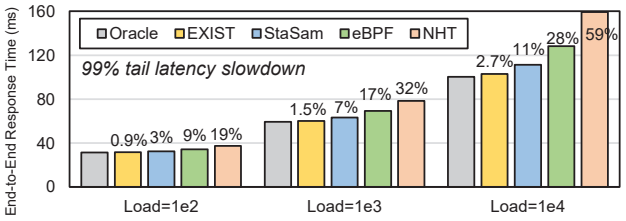


Figure 16. EXIST induces negligible overall performance degradation compared to the baselines.

Table 4. Comparison of space efficiency (MB). *StaSam* provides no chronological information and *eBPF* traces the sys_enter tracepoints, so they occupy little space.

Schemes	*pb*	*gcc*	*mcf*	*om*	*xa*	*x264*	*de*	*le*	*ex*	*xz*	*mc*	*ng*	*ms*
StaSam	4.4	4.5	4.4	4.6	4.7	5.1	5.0	4.9	4.2	32.1	1.9	0.8	4.1
eBPF	0.1	0.2	0.1	0.2	0.2	0.2	0.1	0.1	0.2	0.2	0.2	0.1	0.2
NHT	60.2	64.1	70.2	72.1	73.2	75.2	74.8	71.9	75.2	1173	224	47.9	578
EXIST	55.4	56.2	55.1	54.9	55.3	56.3	57.2	57.1	58.1	456.3	203.1	42.8	498.1

Further, as shown in Table 3, we compare EXIST's time efficiency with SOTA works on similar benchmarks. Compared with hardware tracing-based methods, EXIST greatly reduces the tracing overhead by up to 12x. Compared with more efficient instrumentation-based methods, EXIST achieves lower tracing overhead over most SOTA methods. Note that Log20 [98] aims to find more informative logging locations within a user-specified overhead threshold, and therefore, they can eliminate more logs to reduce overheads further.

Time Efficiency on Real-World Applications: To evaluate EXIST on long-running applications in our clusters, we choose CPU utilization and Cycles Per Instruction (CPI) metrics instead of end-to-end execution time. The former reflects a relative resource perspective and the latter reflects an absolute hardware perspective. For diverse applications, EXIST causes a 1.1% utilization increase on average, which is 2.4x, 2.8x, 12.2x better than the three baselines. As for CPI, EXIST induces 2.2% overhead over Oracle at low workload stress, while StaSam, eBPF, and NHT induce 5.1%, 4.9%, and 20.8% overheads, respectively. EXIST improves time efficiency by eliminating the tedious interrupts and operations and thus causes low overheads on cloud applications, allowing us to deploy it in realistic clusters.

End-to-End Performance Improvements: To further analyze the end-to-end performance improvements via efficient tracing, we evaluate end-to-end response times. We trace Search1 in its corresponding request and record the 99% tail latency of request response times. As shown in Figure 16, per-mille level EXIST induces 1-3% end-to-end performance degradation compared to the $\gg$10% slowdown via single-digit-range overhead of the baselines. Moreover, under stressed scenarios, EXIST has a negligible effect on end-to-end performance and far outperforms the baselines. Note that the end-to-end performance slowdown amplifies the single-point tracing overhead due to inter-service interactions and we will optimize it further in the future.

Space Efficiency: The second critical aspect of the efficiency question is the space overhead, i.e., the amount of main memory used. We set the number of threads and cores to 4 and trace the subjects for 0.5s. As shown in Table 4, *StaSam* samples the function stack and *eBPF* just traces the sys_enter tracepoints. They have a smaller memory footprint but cannot support intra-service tracing. In general, the space overhead is related to program logic, the number of occupied cores, and the utilization of each core. For single-threaded benchmarks, EXIST just traces the occupied cores and collects traces within memory space limits, which maintains most of the traces. For multi-threaded situations like 657.xz, we gradually enable the tracers with scheduling process and are able to obtain most traces within the memory threshold. NHT covers the entire execution on all cores and incurs time-proportional space overhead. As for the online benchmarks, the memory space used is restricted to the memory limit, and they produce fewer trace packets compared with the compute benchmarks due to lower processor utilization.

Impact of System Stress: To analyze the efficiency of EXIST in stressed environments, we present the overhead induced by EXIST under low and high workload stress in Figure 15 for the online benchmarks. The low and high loads indicate roughly 1e2 and 1e4 requests per second, respectively. By reducing the number of costly operations, EXIST achieves

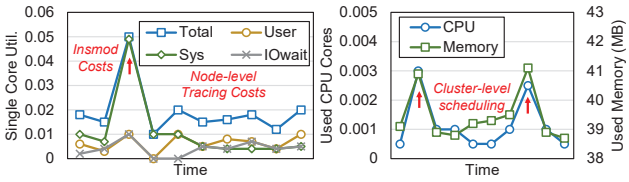


Figure 17. EXIST startup and orchestration overheads.

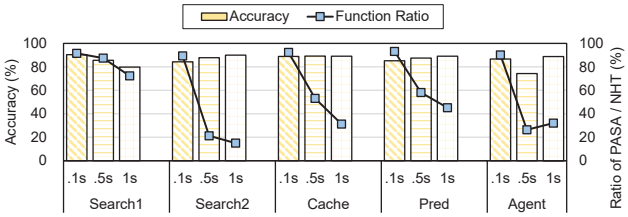


Figure 18. Accuracy of EXIST on real-world applications.

similar time efficiency under high stress to that under low stress, making EXIST suitable for diagnosing more frequent performance anomalies in stressed scenarios. In contrast, the conventional methods all cause greater waste of CPU cycles due to the intrusiveness of normal execution.

Impact of Provisioning Modes: We further analyze the performance of EXIST for applications under different provisioning modes. For CPU-set Search1, EXIST has more negligible overhead than the other CPU-share applications as shown in Figure 15 since the scheduling is bound to specific cores. Also, Search1 experiences less buffer overflow than the CPU-share ones since we can increase the buffer size of each core to the maximized 128 MB. Moreover, although CPU-share Search2 is mapped to more cores, they tend to execute on a few cores, indicating the necessity of core sampling mechanism to enlarge per-core buffer.

Deployment Overhead: We present the realistic deployment overhead of EXIST on a ten-node cluster. As shown on the left side of Figure 17, isolated execution of EXIST incurs negligible CPU usage for tracing, except for the highest 0.05 core occupation during startup to install the kernel module for tracing (Insmod), which aligns with the analysis above. Apart from intra-service tracing efficiency, we present the overhead of cluster-level orchestration. For the evaluated ten-node cluster, the RCO management pod consumes less than 3e-3 cores and 40MB for management under high workload stress. As for the periodical tracing scenarios, it consumes 2e-3 cores and 40MB to trigger and control intra-service tracing. Expanding to the thousand-scale cluster, EXIST achieves less than 1‰ management overhead to orchestrate the large-scale intra-service tracing.

5.3 Effectiveness

To answer the second question, we evaluate EXIST's accuracy compared with the exhaustive tracing method of *NHT* which has significant overhead. It can be used as ground truth and the comparison is similar to prior work [102]. We do not compare the accuracy with *StaSam* and *eBPF* since they cannot provide chronological instruction traces.

Accuracy on Benchmarks: Standard benchmarks have similar behaviors in different executions, so we can compare the reconstructed execution directly. EXIST's accuracy is obtained by measuring the degree of matching between each EXIST-reconstructed execution path of 0.5s period and its corresponding path collected by NHT. For single-threaded compute benchmarks from 600.pb to 648.ex, EXIST achieves 87.4%-95.1% accuracy and 90.2% accuracy on average. On multi-threaded 657.xz, EXIST achieves 62.2% accuracy. For online benchmarks, EXIST achieves 92.4%, 93.1%, and 89.4% accuracy, respectively. Such accuracy can support execution flow reconstruction and application behavior analysis. The accuracy gap is mainly caused by data loss due to the memory space threshold, but the reconstructed execution paths of concurrent threads are accurate. Given the same memory space constraints as NHT, EXIST could achieve more than 95% accuracy on all benchmarks.

Compared with the accuracy of SOTA solutions listed in Table 3, we can further analyze the effectiveness of EXIST. As for hardware tracing-based methods, the first three methods focus on function behavior analysis, and their millisecond-scale coverage falls short when comparing tracing accuracy. JPortal achieves 80% accuracy on Java benchmarks with optimizations on bytecode mapping, and EXIST achieves comparable accuracy with better efficiency. As for instrumentation-based methods, the accuracy of Log20 relies on instrumentation accuracy, while Hubble and Argus automatically obtain function entry&exit traces without considering instruction-level information. DMon collects memory access instruction traces without considering control flow. Overall, besides having satisfactory accuracy compared with NHT on standard benchmarks, EXIST achieves comparable accuracy with SOTA solutions within instruction-level execution tracing.

Accuracy on Real-World Applications: Long-running cloud applications are too dynamic to capture exactly the same periods with EXIST and NHT, and we cannot fully instrument the software to get the ground truth as we did earlier [102]. Thus, we adopt a relative comparison similar to Wall's weight matching scheme [70]. We define the accuracy as *(maxerror - error)/maxerror* where *error* denotes the sum of functions' occurrence differences between NHT and EXIST. For the worst-case results where the functions are all missed, the *maxerror* sums up to 2. Figure 18 shows the accuracy of three cloud applications under different tracing configurations. In general, EXIST achieves 83.7%, 82.6%, 86.2% accuracy on average for tracing 0.1s, 0.5s, and 1s periods, respectively. In particular, EXIST achieves 87.4% for CPU-share *Search2* on average, validating accurate tracing of concurrent threads scenarios. The abnormal case for *Agent*

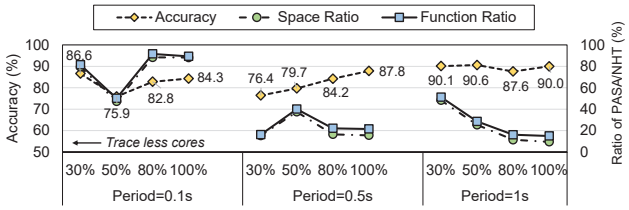


Figure 19. Impact of core sampling mechanism on accuracy.

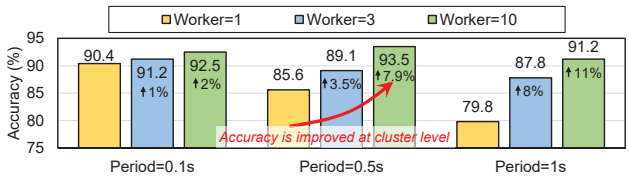


Figure 20. Accuracy of EXIST under cluster-level sampling and trace augmentation.

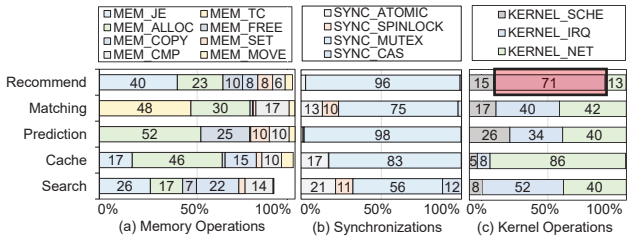


Figure 21. Kernel operations of typical applications.

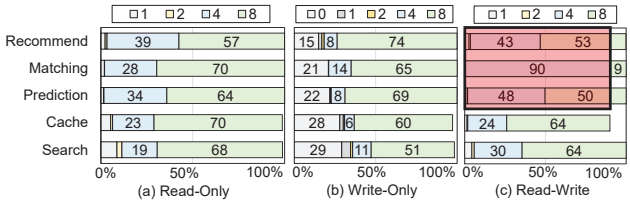


Figure 22. Memory access bandwidth analysis.

under the 0.5s period results from the periodical behaviors of the application, and the two methods capture different phases of the system daemon in the experiments.

Impact of Core Sampling: To analyze the impact of the core sampling mechanism described in Section 3.3, we show the accuracy of EXIST on CPU-share *Search2* under different core sampling ratios in Figure 19. Core sampling ratios denotes the actually traced cores divided by all mapped cores. For 30%-100% ratios, we trace 14, 24, 38, 48 cores with 32MB, 16MB, 8MB, 8MB per-core buffer, respectively. The results show that the core sampling mechanism rarely decreases the tracing accuracy under various settings, but the mechanism significantly affects space occupation. With a 1s period, the selected 30% cores cover all executed cores and EXIST traces more with bigger buffer sizes. In practice, the target process uses just a few cores rather than all cores during the tracing period, so assigning the buffers intelligently and precisely to just the used cores could further increase the tracing efficiency and accuracy.

Impact of Trace Augmentation: Figure 20 shows the accuracy under varying numbers of tracing workers to evaluate the cluster-level trace augmentation mechanism described in Section 3.4. Traces from different workers remove the redundancy and complement the missing parts. We present the results on relatively stable *Search1*. The results of *Worker=1* are the average accuracy of ten workers, and the results of *Worker=3* are merged with the results of selective workers. It shows that synthesizing the traces from more workers could improve the accuracy on a single worker due to better trace data coverage. Such a mechanism produces up to 11% improvements and no additional adverse effect on the node-level intra-service tracing efficiency.

5.4 Case Study of EXIST

To answer the third question, we present a case study that utilizes EXIST to understand applications in our clusters.

Firstly, we can use EXIST to get software execution profiles at instruction and function levels. We present an accurate bottleneck analysis of five critical applications in Figure 21. *Search* and *Cache* are traditional CPU-intensive and memory-intensive applications; the other three are emerging AI-powered applications. Matching is based on the BE engine [15]; and Recommend is based on Machine Vision Application Platform (MVAP) [82]. Compared with statistical observability, we can present accurate application behaviors via text matching of functions and instructions. Further, we can diagnose the reasons behind the differences.

At the function level, we present the ratios of costly functions of three critical categories: memory, synchronization, and kernel and the categorization is similar to prior works [80]. Although the results align with those of traditional applications [44, 80], we find that three ML-based applications perform differently. Take *KERNEL_IRQ* of *Recommend* as an example. It is heavily multi-threaded, so more rescheduled interrupts are followed by mutex synchronizations, increasing the ratios of corresponding leaf functions. It provides us with new opportunities to design specific accelerators and system extensions to optimize it for *Recommend* applications. To dive deeper into instruction-level behaviors, we analyze memory access operations in Figure 22. It shows that ML-based applications have significantly higher quad-width accesses (25% to 70%), which may result from the reduced accuracy in high-throughput inference serving.

Secondly, we can use EXIST to diagnose performance anomalies. For the *Recommend* application, we can monitor the increase in execution time, the number of concurrent

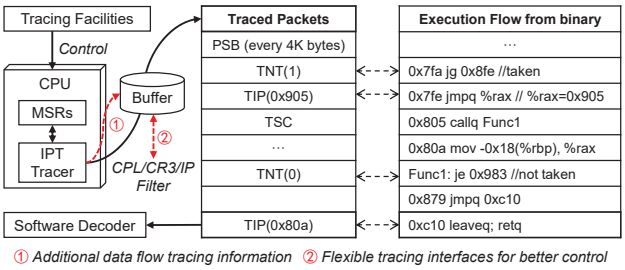


Figure 23. Workflow and potentials of hardware tracing.

threads, and disk I/O by monitoring metrics, but we cannot diagnose the actual reasons of encountered metric anomalies. With EXIST, we can trace its abnormal execution. We identify a *file_write* syscall that consumes 3.7 seconds and multiple syscalls waiting for a *mutex lock*. By mapping the syscall location to user functions, we can find a logging operation that synchronously writes logs and is blocked by disk IO. We can use EXIST to explain the metric anomaly: the synchronous logging thread blocks all other co-located threads from logging, causing abnormal response times and the number of threads. Also, we can improve the scheduling logic by isolating the disks of similar applications or modifying their logging logic in the future. It is worth mentioning that EXIST focuses on improving tracing efficiency instead of improving diagnosing accuracy, so these anomalies could mostly be diagnosed by well-crafted methods with higher overheads [19, 50, 64, 88, 102]. With the help of EXIST, we have diagnosed many unsolved puzzles due to the absence of efficient and practical tools.

6 Discussion

6.1 Hardware Tracing Capability Enhancements

Hardware tracing is a feature provided by major hardware vendors that supports real-time tracing bypassing normal processor execution, e.g., ARM Embedded Trace Macrocell [6], Intel Processor Trace (IPT) [37], and RISC-V Trace [55]. We focus on IPT in this paper since it accommodates most applications in our clusters, but the efficient abstraction designs can be easily extended to other platforms to support universal intra-service tracing observability.

As the successor of the Branch Trace Store (BTS) [38] mechanism from Broadwell generation, IPT features digit-level overhead and branch-level information. It overcomes the widely-used Last Branch Record (LBR) [4] on the tracing accuracy and coverage since LBR can only record the 16 or 32 most recent branch pairs. Figure 23 shows that IPT tracers can be configured and controlled using specific model-specific registers. It collects all indirect branching information with different packets and potentially produces hundreds of megabytes of trace data per CPU per second. Each conditional branch output a TNT packet to record whether it was taken with a single bit (0x7fa) and each indirect branch outputs a TIP packet to record the target address (0x7fe). The traced packets are written to the exclusive memory buffer using various filter principles like current privilege level (CPL) and instruction pointer (IP).

However, current hardware tracing capabilities still have potential improvements to support better tracing observability. *Firstly, the data flow can enhance control flow tracing.* Currently, we can use *PTWRITE* operand [37] or watchpoints [48] to supplement data-flow information. If the underlying hardware natively supports data tracing of specific variables or addresses, performance debugging would be more accurate. *Secondly, more flexible tracing interfaces can further improve tracing efficiency.* As stated in Section 3.3, the modification of tracer settings must be done with tracing disabled. If IPT supports hot switching, we can design more software-friendly abstractions and achieve lower runtime overhead and stability risks. If IPT supports a unified memory buffer setting shared across CPU cores, we can design more portable abstractions and achieve better coverage compared with per-core design.

6.2 Limitations and Future Work

The current designs of EXIST still have three potential improvements which we have left for future work. *The first is the applicable platforms.* We plan to extend EXIST to ARM and RISC-V processors in the future, which covers all hardware architectures in our clusters. *The second is worst-case optimization.* EXIST achieves average per-mille level intra-service tracing overhead at present, but in worst case scenarios, the overhead of EXIST can be higher. We will continue to optimize EXIST for corner cases in the future. *The third is downstream optimization.* EXIST has the ability to optimize more downstream management like scheduling and compilation, so we will continue to find ways to make use of the chronological traces in the future.

7 Related Works

7.1 Research on Observability in Datacenters

Large-scale observability involves collecting execution data from multiple sources to analyze system performance and further guide system optimization. Generally, observability solutions can be categorized into two groups: One is inter-service observability such as Dapper [79], Pivot Tracing [69], and Canopy [42]. They usually insert tracepoints in network events to record the behavior of requests while ignoring the intra-node performance and architecture analysis [35, 97]. The other is intra-service observability, which collects software and hardware metrics [92], traces [102], and logs [93].

Two categories of intra-service observation methods have been widely studied. The first consists of non-chronological sampling methods, which collect data periodically through

Table 5. Functionality comparison with other tracing tools. *InstTrace* means tracing instruction sequences, *UserTrace* means tracing user-level execution, *NoIntrusion* means no injection into software binary, *Continuity* allows for continuous tracing, *Usability* means the difficulty to use the tool.

Properties	Linux-native tools			SOTA tracing tools		
	eBPF	dTrace	sTrace	Hubble[68]	Argus[88]	**EXIST**
InstTrace	✓	✓	×	✓	×	✓
UserTrace	×	✓	×	✓	✓	✓
NoIntrusion	✓	×	✓	×	×	✓
Continuity	×	×	×	✓	✓	✓
Usability	×	×	✓	✓	✓	✓

probes [23, 46, 50, 59, 70, 77, 94]. The samplers can trace hardware counters and software stacks to gather necessary information. They can adjust the overhead by tuning the sampling frequency but have poor trade-off between informativeness and performance [68, 89]. Google Wide Profiling [77] is the most well-known profiling infrastructure that constantly samples the system to provide statistically accurate analyses [44]. However, it focuses on the statistical characteristics of applications and ignores causality information [13].

The second category consists of chronological instrumentation methods, which inserts tracepoints into the program [10, 66, 72]. Schemes like DynamoRIO [10], Valgrind [72], and LLVM [61] assist in implementing dynamic analyses on different abstraction layers (assembly [68] or library [88]). Compared to the five instrumentation-based methods in Table 5 (three Linux-native and two academic tools), EXIST can easily capture user-level instruction-granularity traces continuously with no intrusion, while providing satisfactory intra-service tracing functionality with low overhead. There is also prior work [98] optimizing the accuracy-overhead trade-off of instrumentation methods by selecting the most information instrumentation points. Overall, most tracing systems focus on tracing accuracy rather than tracing overhead, and they are orthogonal and complementary to EXIST.

Besides software tracing methods, there are some domain-specific hardware designs for processors, which can directly produce program traces [29, 71]. For example, TIP [29, 30] is a new hardware component that achieves cycle-level profiling with minimal overhead. These works have well-crafted hardware designs to further improve the state-of-the-practice hardware tracing capabilities. EXIST could also serve as the abstraction over these domain-specific architectures towards realistic cluster-scale deployment. There is also some prior work on intelligent trace analysis and automated root-cause diagnosing [7, 20], which optimize the downstream tasks of EXIST and can make better use of EXIST in datacenters.

In summary, EXIST is different from existing work and pursues efficient and general intra-service tracing observability in shared and stressed datacenters.

7.2 Research on Utilizing Hardware Tracing

The hardware tracing capabilities are popular on commercial CPUs, and much prior work has proposed utilizing this feature for different objectives, which can be divided into three categories. The first is program debugging [19, 28, 47, 48, 90, 101] by storing program execution snapshots for post-mortem analysis. Their use of hardware tracing focuses on post-mortem reconstruction of crash failures, while we concentrate on runtime tracing for performance debugging. Therefore, the system designs of using hardware capabilities are totally different. The second is security enhancement [27, 60, 78] such as kAFL [78] that utilizes code coverage information by IPT to guide fuzzing processes and Griffin [27] that utilizes IPT to enhance control flow integrity. The third is exhaustive runtime tracing [13, 95], which reports the execution flow of programs in datacenters. Prior work like ProRace [95] and DWT [13] focus on combining tracing with dataflow detection to investigate data-related problems. There are also prior work that optimizes the tracing infrastructure with emerging memory technology [87] and more complex programming languages [102], which can further improve EXIST in interfaces and applications.

To the best of our knowledge, we are the first work aimed at building efficient and flexible intra-service tracing abstractions in shared and stressed clusters.

8 Conclusions

Datacenters critically need a full understanding of performance anomalies under shared and stressed scenarios, and the primary challenge is to enable per-mille level efficient intra-service tracing facilities. We analyze the cause and effect of tracing overheads in complex scenarios, and introduce EXIST, an extremely efficient intra-service tracing system using off-the-shelf hardware capabilities. EXIST optimizes the control and data paths at the node level and flexibly orchestrates tracing at the cluster level. Through extensive evaluations, we show that EXIST achieves up to 10x efficiency improvements and over 90% accuracy in realistic clusters. We hope the extremely efficient intra-service tracing observability enabled by EXIST could enhance the explainable management of datacenters.

9 Acknowledgments

We sincerely thank our shepherd Michael Stumm and other anonymous reviewers for their valuable comments that helped us to improve the paper. We thank Yechen Xu and Linhang Li for their feedback on earlier versions of this manuscript. This work is supported by the National Key R&D Program of China (No. 2022YFB4501702), the National Natural Science Foundation of China (No.62122053), and the Alibaba Innovative Research Program (AIR). The corresponding authors are Chao Li and Guoyao Xu.

References

[1] 2024. eBPF. https://ebpf.io/, Last accessed on 2024-04-01.

[2] Alibaba. 2023. Havenask. https://github.com/alibaba/havenask, Last accessed on 2024-04-01.

[3] Amazon Web Services. 2023. Recommender System Solution. https://www.amazonaws.cn/en/solutions/technology/ai-ml/recommender-system-solution/, Last accessed on 2024-04-01.

[4] Andi Kleen. 2023. An Introduction to Last Branch Records. https://lwn.net/Articles/680985/, Last accessed on 2023-10-27.

[5] Apache. [n. d.]. ab - Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/docs/2.4/programs/ab.html,.

[6] ARM. 2019. Arm® Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.5. (2019).

[7] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, USA). USENIX Association, USA, 307–320.

[8] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems* (1st ed.). O'Reilly Media, Inc.

[9] Stefanos Boccaletti, Celso Grebogi, Y-C Lai, Hector Mancini, and Diego Maza. 2000. The Control of Chaos: Theory and Applications. *Physics reports* 329, 3 (2000), 103–197.

[10] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph. D. Dissertation. Massachusetts Institute of Technology, Department of Electrical Engineering.

[11] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)* (Berlin, Germany). Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3185768.3185771

[12] 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 (CGO)* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 12–23. doi:10.1145/2854038.2854044

[13] Jian Chen, Ying Zhang, Xiaowei Jiang, Li Zhao, Zheng Cao, and Qiang Liu. 2020. DWT: Decoupled Workload Tracing for Data Centers. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 677–688. doi:10.1109/HPCA47549.2020.00061

[14] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 107–120. doi:10.1145/3297858.3304005

[15] Alibaba Cloud. 2023. BE Engine for Matching. https://www.aliyun.com/activity/bigdata/airec/be, Last accessed on 2024-04-01.

[16] Alibaba Cloud. 2023. Graph Compute Terms. https://www.alibabacloud.com/help/en/graph-compute/latest/basic-concepts, Last accessed on 2024-04-01.

[17] Alibaba Cloud. 2023. MaxCompute. https://www.alibabacloud.com/en/product/maxcompute, Last accessed on 2024-04-01.

[18] Alibaba Cloud. 2023. Object Storage Service. https://www.alibabacloud.com/en/product/object-storage-service, Last accessed on 2024-04-01.

[19] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (Carlsbad, CA, USA). USENIX Association, USA, 17–32.

[20] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code that Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (Monterey, California). Association for Computing Machinery, New York, NY, USA, 184–197. doi:10.1145/2815400.2815409

[21] Yaniv David, Xudong Sun, Raphael J. Sofaer, Aditya Senthilnathan, Junfeng Yang, Zhiqiang Zuo, Guoqing Harry Xu, Jason Nieh, and Ronghui Gu. 2022. Upgradvisor: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, Carlsbad, CA, 751–767. https://www.usenix.org/conference/osdi22/presentation/david

[22] Paul DuBois. 2013. *MySQL.* Addison-Wesley.

[23] eJTechnologies. 2023. JProfiler: A Commercial Java Profiler. https://www.ej-technologies.com/products/jprofiler/overview.html, Last accessed on 2024-04-01.

[24] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux journal* 2004, 124 (2004), 5.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 3–18. doi:10.1145/3297858.3304013

[26] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 19–33. doi:10.1145/3297858.3304004

[27] 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 (ASPLOS)* (Xi'an, China). Association for Computing Machinery, New York, NY, USA, 585–598. doi:10.1145/3037697.3037716

[28] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse Debugging of Kernel Failures in Deployed Systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC).* USENIX Association, USA, Article 19, 12 pages.

[29] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-Proportional Instruction Profiling. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Virtual Event, Greece). Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/3466752.3480058

[30] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2023. TEA: Time-Proportional Event Analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA, Article 23, 13 pages. doi:10.1145/3579371.3589058

[31] Brendan Gregg. 2024. Stack Trace Visualizer. https://github.com/brendangregg/FlameGraph.

[32] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields,

Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. *ACM Trans. Storage* 14, 3, Article 23 (Oct. 2018), 26 pages. doi:10.1145/3242086

[33] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, 145–155.

[34] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, 73–90.

[35] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance Profiling via Structural Aggregation and Automated Analysis of Distributed Systems Traces. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (Seattle, WA, USA). Association for Computing Machinery, New York, NY, USA, 76–91. doi:10.1145/3472883.3486994

[36] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (Carlsbad, CA, USA). USENIX Association, USA, 1–16.

[37] Intel. 2022. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, Chapter 33: Intel® Processor Trace* (2022).

[38] Intel. 2022. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, Chapter 18.4.5: Branch Trace Store (BTS)* (2022).

[39] Intel. 2024. libipt - an Intel(R) Processor Trace Decoder Library. https://github.com/intel/libipt, Last accessed on 2024-04-01.

[40] Intel. 2024. Reading and Writing Model Specific Registers (MSRs) in Linux. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/reading-writing-msrs-in-linux.html, Last accessed on 2024-02-25.

[41] Jim Keniston, Prasanna S Panchamukhi, Masami Hiramatsu. 2023. Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html, Last accessed on 2024-04-01.

[42] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (Shanghai, China). Association for Computing Machinery, New York, NY, USA, 34–50. doi:10.1145/3132747.3132749

[43] Rudolf E Kalman. 1960. On the General Theory of Control Systems. In *Proceedings First International Conference on Automatic Control, Moscow, USSR*. 481–492.

[44] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)* (Portland, Oregon). Association for Computing Machinery, New York, NY, USA, 158–169. doi:10.1145/2749469.2750392

[45] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the Fourteenth European Conference on Computer Aystems (EuroSys)* (Dresden, Germany). Association for Computing Machinery, New York, NY, USA, Article 34, 16 pages. doi:10.1145/3302424.3303958

[46] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. 2014. Efficient Tracing of Cold Code via Bias-free Sampling. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)* (Philadelphia, PA). USENIX Association, USA, 243–254.

[47] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (Shanghai, China). Association for Computing Machinery, New York, NY, USA, 582–598. doi:10.1145/3132747.3132767

[48] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (Monterey, California). Association for Computing Machinery, New York, NY, USA, 344–360. doi:10.1145/2815400.2815412

[49] The Linux Kernel. 2023. ftrace - Function Tracer. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html, Last accessed on 2024-04-01.

[50] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 163–181. https://www.usenix.org/conference/osdi21/presentation/khan

[51] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-guided Instruction Cache Replacement for Data Center Applications. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)* (Virtual Event, Spain). IEEE Press, 734–747. doi:10.1109/ISCA52012.2021.00063

[52] Alexey Kopytov. 2012. Sysbench Manual. *MySQL AB* (2012), 2–3.

[53] Kubernetes. 2023. Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, Last accessed on 2024-04-01.

[54] Kubernetes. 2024. Pods. https://kubernetes.io/docs/concepts/workloads/pods/, Last accessed on 2024-10-07.

[55] Halil Kükner, Gökhan Kaplayan, Ahmet Efe, and Mehmet Ali Gülden. 2022. RISC-V Processor Trace Encoder with Multiple Instructions Retirement Support. In *Proceedings of the 2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 1–6.

[56] Redis Labs. [n. d.]. Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark,.

[57] Long Zhou Shujie Liu Dongmei Wang Xiaofei Wang Midia Yousefi Yanmin Qian Jinyu Li Lei He Sheng Zhao Michael Zeng Leying Zhang, Yao Qian. 2024. CoVoMix: Advancing Zero-Shot Speech Generation for Human-like Multi-talker Conversations. *Proceedings of the 38th International Conference on Neural Information Processing Systems (NeurIPS)* (2024).

[58] Mingyu Liang, Wenyin Fu, Louis Feng, Zhongyi Lin, Pavani Panakanti, Shengbao Zheng, Srinivas Sridharan, and Christina Delimitrou. 2023. Mystique: Enabling Accurate and Scalable Generation of Production AI Benchmarks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA, Article 37, 13 pages. doi:10.1145/3579371.3589072

[59] Linux. 2023. Linux Perf Tool. https://perf.wiki.kernel.org/index.php/Main_Page, Last accessed on 2024-04-01.

[60] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *Proceedings of the 2017 IEEE International Symposium on High performance computer architecture (HPCA)*. IEEE, 529–540.

[61] LLVM. 2023. The LLVM Compiler Infrastructure. https://llvm.org/, Last accessed on 2024-04-01.

[62] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)* (Portland, Oregon). Association for Computing Machinery, New York, NY, USA, 450–462. doi:10.1145/2749469.2749475

[63] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. 2022. RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 109–125.

[64] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 559–574.

[65] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2023. Understanding and Optimizing Workloads for Unified Resource Management in Large Cloud Platforms. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*. 416–432.

[66] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Chicago, IL, USA). Association for Computing Machinery, New York, NY, USA, 190–200. doi:10.1145/1065010.1065034

[67] 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 (SoCC)* (Seattle, WA, USA). Association for Computing Machinery, New York, NY, USA, 412–426. doi:10.1145/3472883.3487003

[68] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. 2022. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 787–803.

[69] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (Monterey, California). Association for Computing Machinery, New York, NY, USA, 378–393. doi:10.1145/2815400.2815415

[70] Scott Mahlke, Tipp Moseley, Richard Hank, Derek Bruening, and Hyoun Kyu Cho. 2013. Instant Profiling: Instrumentation Sampling for Profiling Datacenter Applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, USA, 1–10. doi:10.1109/CGO.2013.6494982

[71] Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder, and George Varghese. 2003. Catching Accurate Profiles in Hardware. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, USA, 269.

[72] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Diego, California, USA). Association for Computing Machinery, New York, NY, USA, 89–100. doi:10.1145/1250734.1250746

[73] Perf Wiki. [n. d.]. Perf Tools Support for Intel® Processor Trace. http://bit.ly/3GHgJ6l,.

[74] Qi Pi, Weijie Bian, Guorui Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Practice on Long Sequential User Behavior Modeling for Click-through Rate Prediction. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. 2671–2679.

[75] Emerson Network Power. 2014. Data Center 2025: Exploring the Possibilities.

[76] Will Reese. 2008. Nginx: The High-performance Web Server and Reverse Proxy. *Linux J.* 2008, 173, Article 2 (Sept. 2008).

[77] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (July 2010), 65–79. doi:10.1109/MM.2010.68

[78] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium (Security)* (Vancouver, BC, Canada). USENIX Association, USA, 167–182.

[79] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf

[80] Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 733–750. doi:10.1145/3373376.3378450

[81] Kangjin Wang, Ying Li, Cheng Wang, Tong Jia, Kingsum Chow, Yang Wen, Yaoyong Dou, Guoyao Xu, Chuanjia Hou, Jie Yao, and Liping Zhang. 2023. Characterizing Job Microarchitectural Profiles at Scale: Dataset and Analysis. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP)* (Bordeaux, France). Association for Computing Machinery, New York, NY, USA, Article 47, 11 pages. doi:10.1145/3545008.3545026

[82] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (Seattle, WA, USA). Association for Computing Machinery, New York, NY, USA, 639–653. doi:10.1145/3472883.3486987

[83] Xinkai Wang, Hao He, Yuancheng Li, Chao Li, Xiaofeng Hou, Jing Wang, Quan Chen, Jingwen Leng, Minyi Guo, and Leibo Wang. 2023. Not All Resources are Visible: Exploiting Fragmented Shadow Resources in Shared-State Scheduler Architecture. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*. 109–124.

[84] Xinkai Wang, Chao Li, Lingyu Sun, Qizheng Lyu, Xiaofeng Hou, Jingwen Leng, and Minyi Guo. 2024. SHEEO: Continuous Energy Efficiency Optimization in Autonomous Embedded Systems. In *Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD)*. IEEE, 496–503.

[85] Xinkai Wang, Chao Li, Lu Zhang, Xiaofeng Hou, Quan Chen, and Minyi Guo. 2022. Exploring Efficient Microservice Level Parallelism. In *Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 223–233.

[86] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. 2022. SOL: Safe On-Node Learning in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 622–634. doi:10.1145/3503222.3507704

[87] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao. 2022. Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*. 639–654.

[88] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. 2021. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing.. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 193–207.

[89] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can We Trust Profiling Results? Understanding and Fixing the Inaccuracy in Modern Profilers. In *Proceedings of the ACM International Conference on Supercomputing (ICS)* (Phoenix, Arizona). Association for Computing Machinery, New York, NY, USA, 284–295. doi:10.1145/3330345.3330371

[90] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In *Proceedings of the 26th USENIX Security Symposium (Security)*. USENIX Association, Vancouver, BC, 17–32. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-jun

[91] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)* (Tel-Aviv, Israel). Association for Computing Machinery, New York, NY, USA, 607–618. doi:10.1145/2485922.2485974

[92] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–44.

[93] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, California, USA). Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1950365.1950369

[94] Lu Zhang, Chao Li, Xinkai Wang, Weiqi Feng, Zheng Yu, Quan Chen, Jingwen Leng, Minyi Guo, Pu Yang, and Shang Yue. 2023. FIRST: Exploiting the Multi-Dimensional Attributes of Functions for Power-Aware Serverless Computing. In *Proceedings of the 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 864–874.

[95] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Xi'an, China). Association for Computing Machinery, New York, NY, USA, 149–162. doi:10.1145/3037697.3037708

[96] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. 2022. Workload Consolidation in Alibaba Clusters: The Good, The Bad, and The Ugly. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC)* (San Francisco, California). Association for Computing Machinery, New York, NY, USA, 210–225. doi:10.1145/3542929.3563465

[97] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*. USENIX Association, Carlsbad, CA, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

[98] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (Shanghai, China). Association for Computing Machinery, New York, NY, USA, 565–581. doi:10.1145/3132747.3132778

[99] Chen Zhi, Shuiguang Deng, Jianwei Yin, Min Fu, Hai Zhu, Yuanping Li, and Tao Xie. 2019. Quality Assessment for Large-Scale Industrial Software Systems: Experience Report at Alibaba. In *Proceedings of the 2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. 142–149.

[100] Zipkin. [n. d.]. Zipkin. https://zipkin.io/. 2024.

[101] 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 (PLDI)* (Virtual, Canada). Association for Computing Machinery, New York, NY, USA, 1155–1170. doi:10.1145/3453483.3454101

[102] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: Precise and Efficient Control-Flow Tracing for JVM Programs with Intel Processor Trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (Virtual, Canada). Association for Computing Machinery, New York, NY, USA, 1080–1094. doi:10.1145/3453483.3454096