# Enabling Efficient Mobile Tracing with BTrace

Jiawei Wang
Huawei Dresden Research Center
Dresden, Germany
Huawei Central Software Institute
Hangzhou, China
Technische Universität Dresden
Dresden, Germany
wangjiawei135@huawei.com

Nian Liu*
Huawei Central Software Institute
Hangzhou, China
nian.liu@huawei.com

Arnau Casadevall-Saiz
Huawei Dresden Research Center
Dresden, Germany
Huawei Central Software Institute
Hangzhou, China
Vrije Universiteit Brussel
Brussel, Belgium
arnau.casadevall.saiz@huawei.com

Yutao Liu
Huawei Dresden Research Center
Dresden, Germany
Huawei Central Software Institute
Hangzhou, China
liuyutao2@huawei.com

Diogo Behrens
Huawei Dresden Research Center
Dresden, Germany
Huawei Central Software Institute
Hangzhou, China
diogo.behrens@huawei.com

Ming Fu
Huawei Central Software Institute
Hangzhou, China
ming.fu@huawei.com

Ning Jia
Huawei Central Software Institute
Hangzhou, China
ning.jia@huawei.com

Hermann Härtig
Technische Universität Dresden
Dresden, Germany
hermann.haertig@tu-dresden.de

Haibo Chen
Huawei Central Software Institute
Hangzhou, China
Shanghai Jiao Tong University
Shanghai, China
haibochen@sjtu.edu.cn

## Abstract

With the growing complexity of smartphone systems, effective tracing becomes vital for enhancing their stability and optimizing the user experience. Unfortunately, existing tracing tools are inefficient in smartphone scenarios. Their distributed designs (with either per-core or per-thread buffers) prioritize performance but lead to missing crucial clues with high probability. While these problems can be overlooked in previous scenarios (e.g., servers), they drastically limit the usefulness of tracing on smartphones.

To enable efficient tracing on smartphones, we propose *BTrace*: a tracing tool that combines the performance benefits of per-core buffers with the capability of retaining longer continuous traces by partitioning a global buffer into multiple blocks, which are dynamically assigned to the most demanding cores. *BTrace* further gracefully handles unique requirements of modern smartphones, e.g., core oversubscription and resizing.

*BTrace* has been deployed in production, recording an average of 2× continuous traces compared to the current best-performing tracer (Linux ftrace) and improving performance by 20%. Using *BTrace*, we successfully identified numerous bugs that require traces of long duration and are challenging to locate with existing tracers.

***CCS Concepts:*** • **Theory of computation → Data structures design and analysis**; **Concurrent algorithms**; • **Software and its engineering → Software testing and debugging**.

***Keywords:*** Tracing, Software Debugging, Mobile

**ACM Reference Format:**
Jiawei Wang, Nian Liu, Arnau Casadevall-Saiz, Yutao Liu, Diogo Behrens, Ming Fu, Ning Jia, Hermann Härtig, and Haibo Chen. 2025. Enabling Efficient Mobile Tracing with BTrace. 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, 14 pages. https://doi.org/10.1145/3676641.3715994

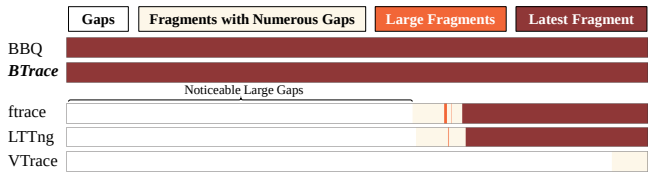
*Nian Liu is the corresponding author.


*ASPLOS '25, Rotterdam, Netherlands.*

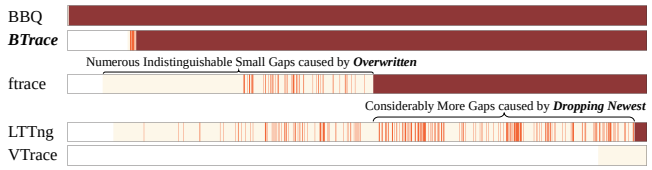
ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3715994

## 1 Introduction

Tracing is a vital technique in the development of operating systems and applications. It allows developers to gain insight




**(a)** Lock screen scenario. Big/Middle cores are idle.




**(b)** Running shopping app. Imbalanced trace-producing speed.

**Figure 1.** Comparison of the effectiveness of different tracers by replaying real-world traces. BBQ [45] uses a global buffer. Ftrace [25], LTTng(-UST) [12], and V(ampir)Trace [30] are state-of-the-art tracers using distributed buffers. Detailed setup can be found in §5. All tracers are configured with the same total buffer size (denoted as $N$). The X-axis shows whether the last $N$ written events are retained in the buffer (newer to the right). Ideally, all $N$ most recent events should be kept — no events should be dropped, and no outdated events should remain. However, existing tracers often drop events within the last $N$ (see §2.2), leading to gaps and fragmented traces. In contrast, *BTrace* (ours) records significantly longer latest traces and eliminates gaps that exist in other tracers while achieving the same performance.

into the system's behavior, which in turn supports the analysis of performance bottlenecks, the optimization of resource utilization, and the improvement of overall stability.

In general, tracing tools (*tracers*) record events in ring buffers. When such a buffer is full, event *producers* overwrite the oldest entries. Upon request, the contents of the buffers are dumped to the screen or other devices, so that the developers can inspect them [37, 39]. To avoid altering the system's behavior, tracing must incur little latency overhead when recording events [1, 5, 11, 38]. Consequently, instead of letting producers contend in a single global buffer, state-of-the-art tracers such as Linux ftrace [25], LTTng [11–13], VampirTrace [30], and others [5, 19, 36, 43, 47, 52] employ *distributed buffers* (i.e., per-core or per-thread).

Although smartphones represent a valuable scenario[1] for the software industry, existing tracers support software developers less effectively in the smartphone scenario than in more traditional scenarios such as servers. In particular, to capture smartphone-related issues, tracers have to store detailed traces over a long time duration. One such example is the analysis of energy consumption, which tends to be measured across different scenarios (e.g., lock screen or while using apps) over extended periods. Storing the required detailed scheduling and frequency adjustment decisions can produce traces of an average of 100 MB per core per minute (§2.2). Another example is the analysis of unexpected frame drops and silent defects (§6), which can only be identified by a series of sparsely spread events in the trace. Existing tracers introduce gaps, i.e., drops of events, in the event sequences surprisingly often, causing such sparse events to be missed.

To demonstrate this effect, we conduct an experiment that replays real-world smartphone workloads using various tracers (see Fig. 1). We employ a simple tracer based on an efficient (in space) but slow global buffer (BBQ [45]) as the baseline. Although tracers with distributed buffers introduce 10x shorter recording latency compared to BBQ (§5), the resulting traces are often discontinuous, missing events that were overwritten or dropped unexpectedly and unnoticed. While such issues were subtle and overlooked in server and desktop scenarios, they become profound limitations in smartphone scenarios due to their unique characteristics.

***Highly skewed trace production speed among cores.*** Smartphones often employ asymmetric multicore processors [3] to implement energy-aware strategies, such as schedulers that frequently idle the big cores [16]. In per-core buffer tracers, when big cores idle to save energy, little cores (power-efficient cores) continuously fill their trace buffers, overwriting old entries; meanwhile idling big cores keep their older traces intact, causing noticeable large gaps (see Fig. 1a) and numerous indistinguishable small gaps (see Fig. 1b), which ultimately fragment the traces. These fragmented traces degrade the effectiveness of the tracers, posing a major challenge in analyzing complex issues, such as those related to scheduling strategies. The indistinguishable small gaps, in particular, can be misleading for developers, making it unclear whether a gap results from non-executed code (e.g., a non-taken branch) or trace drops.

Therefore, the effectiveness of a tracer in locating defects is enhanced by maintaining a longer *latest fragment* — the most recent sequence of events that contains no dropped entries. Compared to the global-buffer tracer (based on BBQ), the latest fragment of ftrace (with per-core buffers) is more than 50% shorter. Although one might be tempted to simply increase buffer sizes, that would be infeasible. Our empirical studies show that buffers would require 2–3× more memory (§5.2) — i.e., over 1 GB — to avoid losing events within 30 seconds; however, such memory usage is impractical given the *limited capacity of smartphones*, which typically have only 4 to 8 GB memory [44].

***Core oversubscription is rather the rule, not the exception.*** Smartphones typically have a significant core oversubscription: The system has many more threads running

[1]The global smartphone market size reached at USD 527 Billion in 2023 [40].

than physical cores available. In our experiments, we observe more than 30 distinct trace-generating threads per core across various scenarios (§2.2). Therefore, threads are likely to be preempted while writing to the trace buffer at arbitrary program locations.

In such situations, existing tracers either block other threads (like BBQ), drop the newest entries (like LTTng [10]), or disable preemption in the kernel (like ftrace). Due to that, LTTng introduces considerably more trace gaps compared to ftrace (see Fig. 1b). Unfortunately, disabling preemption is not a viable option for tracing components in userspace, as it incurs a prohibitive cost from kernel round-trip operations — often exceeding the buffer tracing latency itself. With the growing demand for debugging complex userspace frameworks like AOSP [22] and OpenHarmony [20], alongside the increased deployment of OS services in userspace — both in Linux [31, 34] and in modern microkernel-based OSes [7] — the effectiveness of existing tracers is drastically limited.

***In-production tracing requires dynamic buffer resizing.*** The growing trend of enabling tracing in production to detect hard-to-trigger defects [4, 29, 32, 46, 51] is also poorly supported by current tracers. In addition to emphasizing minimal recording latency to avoid influencing user experience, hard-to-detect defects typically occur during specific periods of long executions in production, such as browsing certain web pages or during application startup. However, the limited smartphone memory cannot be wasted with tracing buffers. Therefore, tracers must efficiently resize buffers in runtime, allocating larger buffers only during critical execution phases, and thereby minimizing the impact on user experience.

The main challenge of resizing is safely reclaiming the unused memory when shrinking the buffer. Concurrent access to per-core buffers can easily trigger use-after-free bugs, and to avoid that, ftrace disables preemption while reclaiming memory; that is unacceptable in userspace, as discussed above. Userspace tracers lack this capability altogether; to implement concurrent buffer shrinking, they would have to employ safe memory reclamation (SMR) mechanisms in the critical path of the producers, significantly impacting their performance and the system's behavior.

To enable efficient tracing on smartphones, we propose *BTrace*, an efficient block-based tracer applicable in kernel and userspace. Unlike existing tracers, which often produce fragmented traces and might miss crucial events, *BTrace* records continuous traces, fully utilizing the available buffer, while maintaining equal or even lower recording latency. Specifically, *BTrace* partitions a global buffer into multiple equally sized *blocks* (§3). Each core retrieves a block to record events; the block is exclusive to the core. Once the current block is filled up, the core moves to the next available block. This approach takes the best of both worlds. It preserves the high memory efficiency of the global buffer while reducing contention, thereby minimizing write latency (§3.1). *BTrace* further enhances the *effectivity ratio*, i.e., the ratio of the latest fragment to the overall buffer capacity, by closing blocks that are likely to be overwritten soon, preventing newer entries from being recorded in those blocks (§3.2). To provide a non-blocking interface while not dropping events, *BTrace* allows some operations to be performed out-of-order (e.g., trace writing inside blocks, advancing to the next block) and skips blocks currently occupied by preempted threads (§3.4). Finally, *BTrace* supports runtime resizing with an implicit memory reclamation scheme (§3.3), neither introducing synchronization overhead nor disabling preemption.

Our experiments (§5) over several popular smartphone applications and typical usage scenarios demonstrate that *BTrace* achieves 2× better effectivity ratio than the best-performing tracer (Linux ftrace) while reducing recording latency by 20%. By allocating a 450 MB tracing buffer, *BTrace* successfully captures detailed and continuous 30-second traces, whereas ftrace(the best of the existing tracers) records intermittent data totaling 10 seconds. In production (§6), *BTrace* has allowed our developer teams to efficiently diagnose over 200 bugs that require traces of long duration and are challenging to locate with state-of-the-art tracers.

## 2 Background and Motivation

### 2.1 System Tracing

**Logging vs. Tracing vs. Profiling.** Logging, tracing, and profiling are distinct yet complementary techniques for debugging and analyzing complex systems, each with specific data collection requirements. Logging primarily aids functional testing by recording arbitrary strings from various system components [18]. Tracing, on the other hand, focuses on performance analysis by capturing detailed system states to reconstruct event timelines. As a more specialized form of logging, tracing generates significantly larger data volumes and incurs certain runtime overhead. Profiling involves sampling high-frequency events like CPU and memory usage or call stacks at periodic intervals. The key distinction between tracing and profiling is that tracing is designed to be *non-droppable* (other than the oldest) for locating the root causes, while profiling allows for data to be missed or dropped. Although *BTrace* is primarily designed for tracing, it can also be applied to logging and profiling to improve performance and memory efficiency.

**Persist vs. In-memory.** To minimize recording latency, most tracers store traces in an in-memory, non-persistent ring buffer in overwrite mode. When suspicious symptoms are detected, a daemon collector dumps the buffer [26, 32, 51]. Some userspace tracers persist all traces to flash storage. They first store traces in an in-memory ring buffer and then persist them either synchronously when the buffer fills [30] or asynchronously via a reader [12]. However, persisting all traces increases write frequency, which adds energy and

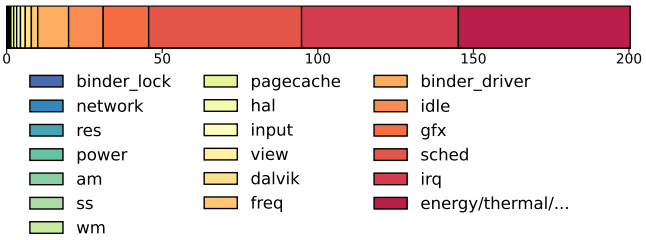



**Figure 2.** Trace producing speed of different *atrace* categories (MB per core per min). *energy/thermal/...* are custom tracepoints that provide detailed reasons. Locating defects in smartphones typically requires enabling multiple tracepoints to locate system-wide issues.

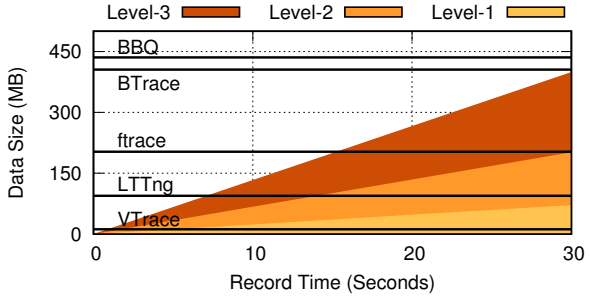



**Figure 3.** Levels of traces can be recorded by different tracers. Horizontal lines show the latest and continuous traces (the latest fragment) that can be recorded by different tracers with the same fixed 450 MB buffer. A higher level of traces includes more details and is required to locate system-wide performance issues. *BTrace* can store all level-3 traces in 30 seconds (without exceeding the horizontal line), while ftrace can only store level-2.

performance overhead [15], shortens flash storage lifespan, and degrades write speed during concurrent reads. As a result, smartphones typically rely on in-memory tracing for issue diagnosis, both in development and in production.

### 2.2 Calls for An Efficient Mobile Tracer

Using a global buffer leads to unavoidable contention of the tracing buffer [12]. Therefore, state-of-the-art tracers adopt distributed buffers, either per core [5, 11, 12, 19, 25, 36, 43, 47] or per thread [30, 32, 33, 50–52], to reduce the contention and prioritize performance. Yet, all of them fail to efficiently locate defects issues on smartphones due to the following observations.

**Observation 1: Smartphones require detailed trace recording over a long duration within a limited buffer capacity.** We observe a substantial increase in trace size when diagnosing defects and issues on smartphones. Fig. 2 shows the trace generation rates for various Android atrace[2] [17] categories. Identifying energy and performance issues on smartphones requires tracking high-frequency events, such as idle decisions (idle), frequency altering (freq), scheduling actions (sched), and energy-aware strategies (energy / thermal /...), where each core generates approximately 100 MB of trace data per minute on average.

Beyond the sheer volume of data, some defects necessitate recording traces over an extended period. Based on an analysis of over 4,000 issues in the defect tracking system of a beta release of the smartphone system (§6), the following three primary defect types make up 5% of the total issues. First, *energy efficiency defects* arise not from a single decision but from the interplay of multiple strategies over time, such as scheduling, frequency altering, and idle decisions. Second, diagnosing *performance issues*, like frame drops, also requires long-duration traces to identify causes such as priority inversion or suboptimal scheduling decisions of a period. Finally, daemon processes are deployed to detect *silent defects*, which only report issues after long timeouts to avoid false positives. Therefore, extended trace recording is crucial for diagnosing the root causes of these problems. We present detailed case studies of in-production defects associated with these three types in §6.

However, smartphones have strict restrictions on tracing buffer size due to constrained memory capacity, making it challenging to store all the necessary information. For instance, many recently released flagship devices are still equipped with only 8 GB of memory [2, 23, 42], while lower-end models may have as little as 4 GB [35, 48]. Moreover, memory consumption by applications, including background processes, continues to rise. As a result, tracing buffers are typically restricted to under 500 MB to avoid negatively impacting system-wide behavior and user experience.

Nevertheless, existing tracers with distributed buffers are inefficient at recording all necessary traces due to poor memory efficiency. Fig. 3 illustrates the trace size over a 30-second period on a 12-core smartphone, comparing various tracers using the same 450 MB buffer. We categorize the enabled tracepoints into different levels based on their frequency and relevance for diagnosing various types of bugs. Level-1 traces include minimal events like the binder (the binder_driver category), which help establish thread dependencies and locate issues like thread hangs. Level-2 traces capture additional information, such as scheduling decisions or IRQs (sched or irq category in atrace), necessary for diagnosing performance issues like frame drops or audio stuttering (often linked to IRQ handling). Level-3 traces add further custom details, such as thread migration triggered by energy or thermal considerations, which are difficult to explain without deeper insights. As shown in the figure, with existing tracers and a 450 MB buffer, only level-2 traces can be reliably stored continuously. Recording all level-3 traces over the

[2]Atrace uses ftrace to trace system events.

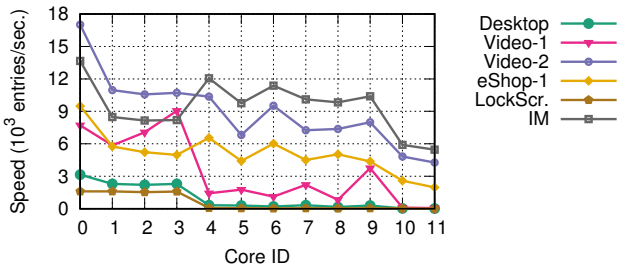



**Figure 4.** Average trace speed (in thousands of entries per second) for each core in a 12-core smartphone [24] across selected typical workloads from §5. Core 0-3 are little cores. Core 4-9 are middle cores. Core 10-11 are big cores.

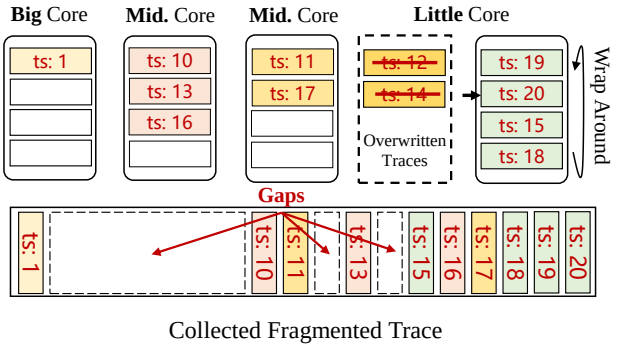



**Figure 5.** An example of buffer under-utilization and data continuity issues caused by skewed trace production speeds across per-core buffers. Traces with timestamps (ts) 12 and 14 are overwritten due to buffer wraparound in the little core, while nearby traces (e.g., ts-11 and ts-13) are retained in the middle and big cores, resulting in indistinguishable gaps in the collected traces.

past 30 seconds requires over 1GB of buffer space, which is impractical for smartphone deployment.

**Observation 2: Thousands of threads with highly skewed trace-producing speeds lead to unexpected trace overwrites or drops.** Smartphones with asymmetric processors [3, 27] often idle the big cores to save energy, causing highly skewed speeds in producing traces. Fig. 4 demonstrates the varying trace-producing speeds in smartphones across different scenarios over a 30-second period. For example, workloads such as instant messaging (IM) tend to generate similar trace speeds across cores, while scenarios such as playing online video (Video-1) exhibit significant differences, with the little cores producing considerably more traces than the big cores.

This skewed trace-producing speed leads to unexpected overwrites, which degrades the **effectivity ratio**, indicating the proportion of the latest fragment (in size) retained in the buffer. Fig. 5 illustrates an example: while the buffer for a slow-producing big core remains half-filled, the buffers for

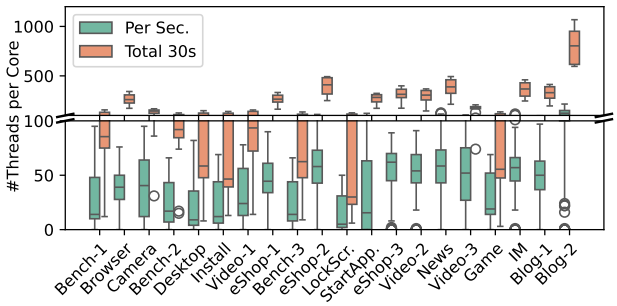



**Figure 6.** Box plot of the distinct thread count per core producing traces simultaneously across different scenarios. Smartphones experience significant over-subscription in most scenarios. *Total* represents the thread count over 30 seconds, while *Per Sec.* indicates the count within a second.

the little core wrap around, dropping some entries, leading to **low buffer utilization**. Furthermore, the skewed speed also results in indistinguishable gaps in the collected traces. For example, traces in the middle cores' (slower producers') buffers, such as trace entries with timestamps (ts) 11 and 13, are retained, while nearby traces in the little core's (faster producer) buffers, such as ts-12 and ts-14, are overwritten, resulting in an effectivity ratio of $\frac{6}{16} = 37.5\%$, where 6 is the size of the latest fragment (from ts-15 to ts-20), and 16 is the total capacity of the buffer. Moreover, these gaps are often subtle and hard to detect, unlike the larger gaps (from ts-2 to ts-9, which are also overwritten by the little core).

Even more critically, smartphones experience significant thread over-subscription, causing the most recent entries to be dropped to avoid blocking the producer. Fig. 6 shows the number of threads per core recording to the trace buffer, both on a per-second basis and over a 30-second period. Under heavy load, for each core, an average of 400 threads write to the same buffer over 30 seconds, and an average of 30 threads per second, significantly increasing the likelihood of a thread being scheduled out during the recording of the trace, which can block subsequent recording.

To avoid blocking, state-of-the-art tracers employ various strategies. Per-thread tracers naturally avoid this issue by allowing each thread to write independently. However, they suffer from extremely low buffer utilization when handling thousands of threads, making them suitable only for scenarios with a limited number of threads (e.g., two threads in Hubble [32]). The ftrace disables preemption in the kernel to prevent threads from being preempted during trace writes, but this approach is costly and, therefore, not feasible for userspace tracing. Other tracers, such as LTTng, **sacrifice buffer availability** by discarding the newest data, leading to significant data loss, as shown in Fig. 1b.

Given the above issues, in real-world production environments, a state-of-the-art tracer typically needs to reserve

**Table 1.** Comparison of *BTrace* with state-of-the-art tracers. Here, $C$ and $T$ represent the number of cores and threads, respectively. $N$ denotes the total number of data blocks, and $A$ represents the number of active blocks, which is equivalent to the number of metadata blocks.

| | Contention | Utilization | Effectivity Ratio | Resizing | Availability |
|---|---|---|---|---|---|
| **BBQ** | High (Global Buffer) | 1 | 1 | Not support | Blocking |
| **ftrace** | Low (Core Local) | 1/C | 1/C | Disable Preemption | Disable Preemption |
| **LTTng** | Low (Core Local) | 1/C | 1/C | Not support | Dropping Newest |
| **VTrace** | Low (Thread Local) | 1/T | 1/T | Not support | Separating to Threads |
| ***BTrace*** | ***Low (Core Local)*** | ***~1-(C-1)/N*** | ***~1-A/N*** | ***Implicit Reclaiming*** | ***Skipping Blocked*** |

a buffer that is approximately 3× larger than the actual usage. However, this approach is often impractical due to the limited memory capacity of smartphones.

**Observation 3: Increasing demands of in-production tracing and the temporal locality of defects require efficient runtime buffer resizing.** Recently, there has been a growing trend toward enabling tracing in production environments to diagnose performance and functional issues that are difficult to reproduce during development. For instance, Google employs sampling-based feedback-directed optimization [6, 33] in both data centers and mobile devices to analyze performance issues based on traces collected in production. Hubble enables in-production tracing in beta releases of deployed smartphones [32], recording method entries and exits to help locate bugs (reported anonymously with user consent). Hindsight [51] enables always-on tracing in distributed systems to detect rare edge-case requests, capturing detailed traces that are collected after problematic symptoms emerge. Other debugging tools for desktops [9, 28, 46] or servers [4, 29, 49] also enable in-production tracing to address hard-to-reproduce bugs.

Since defects in smartphones typically occur during specific periods of prolonged execution, dynamic runtime buffer resizing, where larger buffers are allocated only during critical execution phases, is crucial for minimizing the impact on system-wide behavior and user experience. For example, an application's cold start time significantly affects user satisfaction. When investigating unexpected delays in launching certain apps in production (as detected by anomaly detectors like Hubble [32]), a large trace buffer is allocated to store detailed traces. Once the app's main activity has finished loading, the buffer can be dumped and reduced in size. Similar requirements apply when analyzing issues related to specific browsing actions or functional operations.

However, resizing the trace buffer, particularly when it is shrinking, involves safe memory reclamation (SMR) mechanisms, such as epoch-based reclamation (EBR) [21] and reference counting [8]. These mechanisms ensure that producers and consumers do not access the portion of the buffer that has been shrunk during or after the reclamation process, which necessitates additional synchronization. Similar to locking, SMR mechanisms mark the buffer as non-reclaimable before accessing it, and update its status to reclaimable afterward, which introduces significant overhead for each read or write operation. Ftrace employs a mechanism similar to EBR by disabling and enabling preemption before and after trace writing [41]. This approach ensures that if a core is preempted by a resizing task (also non-preemptable), no producer on that core is writing the buffer and will not access it during the resizing, allowing the buffer to be safely reclaimed. However, disabling preemption introduces substantial overhead in userspace. This limitation, along with similar drawbacks in other SMR mechanisms, prevents existing userspace tracers from supporting dynamic resizing, despite its importance for debugging userspace frameworks [20, 22] and OS services in multi-server microkernels [7].

## 3 Design of *BTrace*

*BTrace* enables efficient tracing through several key techniques: *block partitioning*, *block closing*, *implicit reclaiming*, and *block skipping*. These techniques will be introduced in the following subsections. Table 1 compares *BTrace* with existing tracers.

### 3.1 Improving Utilization via Block Partitioning

*BTrace* statically partitions the memory space into multiple data blocks. During execution, each data block is assigned to a particular core for a specific duration, allowing its producers (i.e., the threads running on it) to generate traces within that block, thereby reducing the contention. Once a data block is filled, the producer will advance to the next available block. For instance, as illustrated in Fig. 7, data blocks D0, D1, D3, and D4 are assigned to Core 3, Core 1, Core 0, and Core 2, respectively. Once D4 reaches capacity, producers of Core 2 advance to D5 according to the position indicator (the global metadata Pos) of the next block (skipping to D6 to avoid blocking, further explained in §3.4). Data blocks are used in a wrap-around manner. For example, once D7 is filled, producers advance to D0, overwriting and reusing it in the next round (a new round begins when the buffer wraps around). Importantly, data blocks are assigned to cores rather than threads, as smartphones often run thousands of short-lived threads simultaneously. A per-thread design would require reserving thousands of blocks, leading

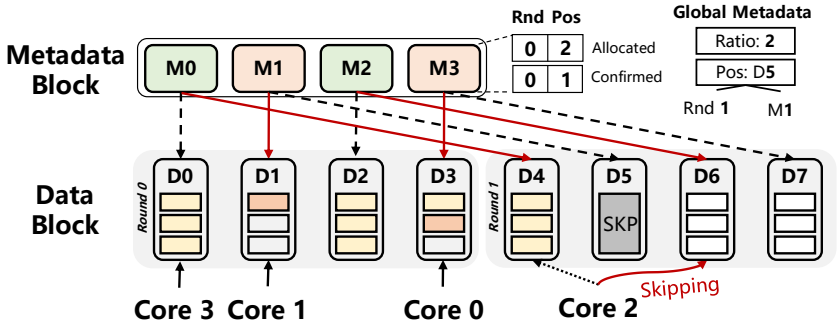



**Figure 7.** Overview of *BTrace*: Each core produces traces to its own exclusive data blocks; the metadata of these data blocks is stored in corresponding metadata blocks according to a global ratio (2 in this example). When a data block is filled, the producer advances to the next available metadata block and its associated data block.

to significant memory wastage since many would remain only partially filled.

Through block partitioning, *BTrace* achieves high memory utilization similar to global buffers, while maintaining low recording latency similar to distributed buffers. In most cases, producers write data within their assigned block (fast path), which effectively eliminates contention between cores. In contrast to distributed buffers, where memory utilization can drop to as low as $\frac{1}{C}$ (for per-core tracers) or $\frac{1}{T}$ (for per-thread tracers) as shown in Table 1, *BTrace* maintains a worst-case memory utilization rate of $1 - \frac{C-1}{N}$, where $N$ is the number of data blocks, $C$ is the number of cores, and $T$ is the number of threads. This worst case occurs when all other $C-1$ cores each occupy one block but generate no data, leaving only one core to utilize the remaining available space. Given a setup with $C$=12 and $T$=500, where each data block is sized as a 4 KB page, and the total buffer size is 12 MB, the memory utilization rate increases dramatically from 8.3% for per-core buffers (0.2% for per-thread buffers) to 99.6% with *BTrace*. Compared with BBQ, although it can achieve 100% utilization, it inherently causes high contention on the buffer, resulting in increased recording latency (see §5), which is mitigated through partitioning in *BTrace*.

### 3.2 Enhance Effectivity via Block Closing

Although partitioning improves buffer utilization, the tracer may still exhibit a low effectivity ratio. For example, as shown in Fig. 7, consider a case where Core 3 has just completed writing data to D0 (the lagging block) after Core 2 has filled D6 and D7. Subsequently, Core 2 wraps around, causing the most recent data (in D0) to be overwritten.

To address this issue, we limit the number of data blocks that all cores can operate on simultaneously to $A$, defined as *active blocks*. $A$ should be greater than or equal to the number of cores to ensure sufficient concurrency. Whenever a producer advances to the next block, it closes the lagging block that is $A$ blocks behind it by filling the remaining space with *dummy* data, preventing producers from producing new traces to that block. In Fig. 7, with $A$=4, after Core 2 advances to D6, it will close D1, and D3-D6 will become the active blocks. At this point, Core 1 can no longer use D1 and must advance to another data block.

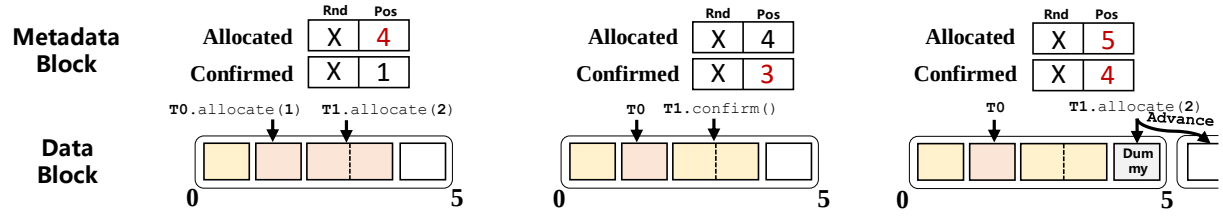
As listed in Table 1, while per-core and per-thread tracers guarantee the effectivity ratio of $\frac{1}{C}$ and $\frac{1}{T}$, *BTrace* achieves an effectivity ratio of $1 - \frac{A}{N}$ under an ideal scenario where all closed blocks are fully utilized. For instance, with a setup where $A$ is set to $8\times C$, the effectivity ratio can reach 96.88%. However, in practice, there is a trade-off when selecting the value of $A$. A smaller $A$ may achieve a higher theoretical effectivity ratio but increases the likelihood that a data block may not be fully utilized before being closed by another core, resulting in reduced memory utilization and effectivity ratio. We examine the suitable setting of $A$ in §5.1.

### 3.3 Enable Resizing via Implicit Reclaiming

*BTrace* adopts implicit reclaiming to achieve safe memory reclamation in producers without introducing any additional synchronization. Specifically, *BTrace* utilizes the semantic that a producer has filled a data block as the indication of ending an epoch[3] of the EBR, since the data block will no longer be accessed by that producer during this round. Consumers, on the other hand, use a simple EBR directly since it is off the critical path. This allows us to safely reclaim the data block. However, due to the core-exclusive design (§3.1), other producers on the same core may still access the block's metadata (not the data) even after the block is filled. Therefore, *BTrace* only reclaims the data and leaves the metadata unreclaimed.

To enable the reclamation of partially filled blocks, *BTrace* further utilizes the semantic of producing traces within a data block as implicit reference counting. Specifically, producing traces involves first *allocating* the demand space, then writing the event to that space, and finally *confirming* the completion of the writing. The allocation and confirmation can be treated as increasing and decreasing the reference

[3]A time period during which someone may access the data.




(a) In-order allocation. (b) Out-of-order confirmation. (c) Filling dummy at tail.

**Figure 8.** Producing traces within a data block (fast path operation).

count, respectively. Once their corresponding values are equal (Allocated.Pos equals to Confirmed.Pos), it indicates that the data block is no longer referred to by any producer; it can be closed as described in §3.2. Subsequently, *BTrace* employs the same method mentioned above for reclaiming these filled blocks.

Nevertheless, with only the above approach, metadata must be reserved for all possible data blocks, leading to *significant memory overhead* and is, therefore, considered unrealistic. Specifically, it must retain the metadata that supports the maximum capacity, even when we shrink *BTrace*'s capacity to a very small size. For example, each metadata block in *BTrace* is 128 bytes. Given a buffer capacity of 2 GB and a data block size of 4 KB, the memory footprint of the metadata would amount to 64 MB. This memory cannot be reclaimed, even if the size of *BTrace* is reduced to as small as 4 KB.

*BTrace* reduces the number of metadata through *mapping*. Specifically, since *BTrace* adopts the closing mechanism (§3.2), the number of active blocks, including their metadata, that producers can operate on simultaneously is fixed. Therefore, instead of assigning each data block its own metadata, *BTrace* restricts the metadata count to be equal to the number of active blocks and maps each metadata block to multiple data blocks. These metadata blocks are initialized at the start and will not be reclaimed.

Specifically, the metadata of $N$ data blocks is mapped to $A$ metadata blocks, with the ratio ($N$:$A$) stored in the global metadata Ratio. For example, as illustrated in Fig. 7, eight data blocks are managed by four metadata blocks, where the ratio is set to two. Every four of the data blocks share the same metadata block (e.g., D3 and D7 share M3). The metadata block uses an indicator Rnd (the round of metadata) to specify its current managed data block. For any other data blocks it manages, *BTrace* considers them to be filled. For example, the Rnd in the metadata block M3 is 0, indicating that it manages the data block D3 (round 0) rather than managing D7 (round 1). With such, resizing *BTrace* can be achieved by altering the Ratio.

### 3.4 Ensure Availability via Block Skipping

*BTrace* ensures availability both within the block and when advancing across blocks.

**Within the block.** Within the block, we enable *out-of-order confirmation* [45]. Specifically, after threads allocate space within the block, they do not need to confirm their writes sequentially according to the allocation order. Instead, each thread confirms its writes independently by incrementing the total confirmed entries. Once the total confirmed entries match the total allocated entries, all allocated spaces are considered finished.

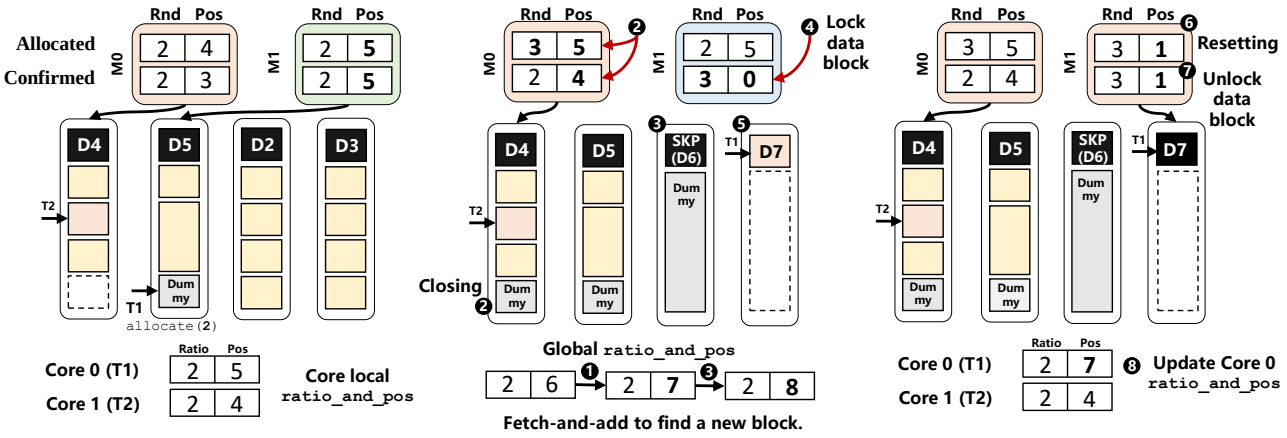
**Advancing the block.** Producers can also encounter blocking during block advancement. For example, a producer may be blocked when advancing to a data block that has been assigned but not yet prepared (i.e., its metadata has not been updated) by another preempted thread on the same core, or when wrapping around to a data block that has unconfirmed entries. However, such cases occur relatively infrequently. Therefore, to address this, we adopt a skipping method that trades some extra memory for increased availability.

Specifically, when a producer $P$ encounters the above cases, *BTrace* forces the lagging producer to leave by closing the block. Subsequently, $P$ skips the block, sacrificing it to maintain high availability. For example, in Fig. 7, Core 2 skips D5 and utilizes D6 instead, because the corresponding metadata block (M1) for D5 is not yet complete. The skipped data block can be reused in the subsequent rounds once all its writes are finished. Experimental results demonstrate that even under significant over-subscription, the sacrificed memory is negligible (§5.1).

## 4 Implementation

### 4.1 Single Data Block Operations

Figure 8 illustrates the trace production procedure within a block (the fast path). The metadata block contains two variables: one for allocating entries (Allocated) and one for confirming entries (Confirmed). Only the lower bits of these variables (Pos) track the corresponding information, while the higher bits of each (Rnd) indicate how many rounds this metadata block has been used and also indicate its associated data block (see Sec. 3.3). In this example, the data block has a capacity of five, while the metadata block manages the underlying data block. The value of Rnd is marked as X for simplicity.




(a) Before advancement. (b) Skipping block D6. (c) Finding available block D7.

**Figure 9.** Data block advancement (slow path operation).

As shown in Fig. 8(a), threads T0 and T1 are from the same core, and this data block is currently assigned to them. When T0 produces traces, it allocates space by atomically performing a fetch-and-add (FAA) operation on the Allocated variable. However, before T0 executes the subsequent confirming operation after finishing copying the data into the allocated space, T1 may be scheduled to run and allocate another space. To avoid blocking T1 from confirming its traces, instead of letting Confirmed records the position that points to the boundary of confirmed and unconfirmed spaces, *BTrace* permits out-of-order confirmation by changing it to a counter that records how many spaces have been confirmed. Therefore, T1 can directly update Confirmed.Pos from 1 to 3 as depicted in Fig. 8(b). Consumers can only read the data of this data block when all allocated spaces have been confirmed, i.e., the Pos of Allocated is equal to that of Confirmed.

*BTrace* supports non-fixed-size traces, so the remaining space at the end of a data block might be insufficient to store the incoming trace. As shown in Fig. 8(c), T1 attempts to produce a trace with a size of 2, while the remaining space in the current data block is only 1. In this case, *BTrace* requires T1 to fill the rest of the block with a dummy node, updating the corresponding Allocated and Confirmed variables, and then advances to the next available block (by following the procedure in §4.2). Notice that the block advancement can also be out of order to avoid blocking producers. For example, when T1 advances to the next data block, it is acceptable that T0 has still not confirmed its entries on this data block.

### 4.2 Data Block Advancement

Figure 9 illustrates the block advancement procedure after the depletion of the current data block (the slow path).

**Structures.** The example shows four data blocks, managed by two metadata blocks (due to the metadata mapping introduced in §3.3, the ratio here is 2). Fig. 9(a) represents the initial state before the advancement. At this stage, Rnd in both metadata blocks M0 and M1 is equal to 2, meaning they are managing data blocks D4 and D5, respectively. Each core has its own core-local variable ratio_and_pos, which combines the Ratio in the higher bits with the Pos in the lower bits to allow atomic updates, indicating the position of their assigned metadata and data blocks.

**Block advancement.** In Fig. 9(a), T1 from Core 0 is producing traces to D5, while T2 from Core 1 is producing traces to D4. When T1 attempts to allocate space of size 2, the remaining space in D5 is insufficient. Thus, T1 fills its remaining space with dummy data and advances to the next block. To find a new data block, T1 first performs an FAA operation on the global ratio_and_pos (❶), identifying D6 as the next candidate block, which is managed by metadata block M0. However, Confirmed.Pos of M0 is 3 rather than the block capacity of 5 (Fig. 9(a)), indicating that its previous producer has not finished writing traces to D4. Therefore, D6 cannot be used in this round. To improve the effectiveness, T1 then closes D4 (see *Block Closing* in §3.2) by filling its remaining space with dummy data and updating the Allocated.Pos and Confirmed.Pos in M0 to values 5 and 4 (❷). To avoid blocking T1, after double-checking that producer (T2) has not finished confirming (Confirmed.Pos in M0 does not equal to 5), T1 skips D6 (see *Block Skipping* in §3.4) by marking its header as SKP and advances to D7 (❸).

Once T1 confirms that the metadata block of D7 (M1) indicates the previous data block has been filled (Confirmed.Pos of M1 is 5 in Fig. 9(a)), it attempts to lock the data block by performing a compare-and-swap operation (❹), setting Confirmed.Rnd to 3 and Confirmed.Pos to 0 atomically. This prevents the data block from being overwritten or being

reclaimed. If the compare-and-swap operation fails, it means some wrap-around producer has already successfully locked the metadata block. In that case, T1 moves on to find the next candidate block and repeats the same procedure. Otherwise, upon success, T1 updates the header of D7 (❺).

Subsequently, T1 resets the metadata block to reuse the data block in this round by updating Allocated using compare-and-swap (❻, with Pos updated to 1, which corresponds to the header's size). A failed compare-and-swap means some wrap-around producer has already closed this block, which requires T1 to try again. Otherwise, upon success, T1 then confirms the writing of the header (❼) to unlock the data block. Finally, T1 uses an atomic compare-and-swap to update the core-local ratio_and_pos (❽), allowing other threads of Core 0 to use D7. Afterward, T1 can produce traces to D7 following the fast-path procedure in §4.1. If the compare-and-swap fails, it means that other threads on Core 0 have already allocated a new data block. T1 will fill D7 with dummy data and use that block (not shown in the Fig. 9).

### 4.3 Speculative Consumer

When reading traces, the consumer first catches up to the latest un-overwritten block by referencing the ratio_and_pos of the producer. Since the trace producer may wrap around and try to overwrite the same data block that the consumer is reading, *BTrace* adopts a speculative method to avoid blocking the producer. Specifically, when the consumer reads a filled data block, the consumer speculatively reads the block and then re-checks the corresponding metadata after finishing the read. If the metadata indicates that the data block is not overwritten and the block is not marked as SKP, the consumer reads all the data in the block except for the dummy entries. Otherwise, the consumer abandons the current block and reads the next one.

For a non-filled data block, the consumer can only read the data if Allocated and Confirmed are equal. After reading, the consumer closes the block by filling the remaining space with dummy data and proceeds to read the next.

### 4.4 Buffer Resizing

To support buffer resizing, all data blocks should reside in contiguous memory addresses. We preserve the virtual memory address of the buffer equal to its maximum size, using the mmap and munmap syscalls to allocate and free physical memory while maintaining the virtual memory address. After that, *BTrace* can be easily resized by adjusting the ratio (§3.3). To ensure that all producers correctly observe the new ratio and that subsequent traces are properly placed in data blocks in accordance with the updated ratio, the resize procedure closes all active data blocks by executing the advancement procedure outlined in §4.2 after updating the global ratio_and_pos.

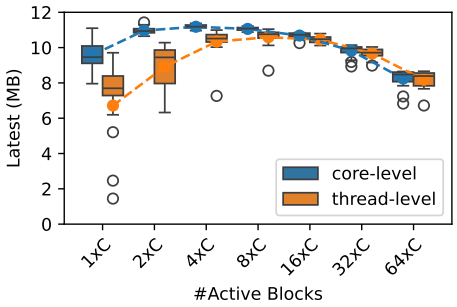



**Figure 10.** The size of the latest fragment of *BTrace* under different numbers of active blocks and replay methods. Here, *C* represents the number of cores.

When shrinking the buffer, it is essential to ensure that all producers and consumers have exited the shrunk blocks before they are unmapped and reclaimed. To guarantee that all producers have left, the shrinker traverses all metadata blocks and checks whether they have been updated to the newer round. For consumers, *BTrace* employs an epoch-based reclamation method. The shrinker traverses all consumers to ensure they are not in the shrinking epoch and have left. Once these conditions are met, the shrinker can safely unmap and reclaim the memory.

## 5 Evaluation

We conducted a trace replay benchmark to evaluate *BTrace* with existing tracers under real-world smartphone loads.

**Workloads.** We collected 20 traces from three kinds of typical scenarios, where traces are needed to analyze performance and energy issues, including (1) the top 10 applications and games by number of downloads in the app store, (2) commonly used memory, CPU, and system performance testing software utilized by developers, and (3) several typical usage scenarios, such as at lock screen and desktop. Traces are collected at level 3 (see Fig. 3) over a duration of 30 seconds on a 12-core production smartphone [24].

**Replaying setup.** During the replay, we pin specific threads to each core, generating corresponding traces based on timing, core assignments, and the lengths of each entry. To examine the impact of oversubscription, we conduct two replay methods, including *core-level* and *thread-level* replaying. Thread-level replaying creates the same number of threads per core as counted in the collected traces, while for core-level replaying, each core runs only one thread responsible for producing all traces of that core. To further identify dropped traces, during the writing phase, we assigned each trace a unique, monotonically increasing logic stamp. Data with logic stamps that do not appear in the readout are considered missing. We allocate a 12 MB buffer for each tracer. For *BTrace*, we set the size of each data block to be one memory page (4 KB).

**Table 2.** The size of the latest continuous entries (in MB), the loss rate, the number of fragments, and the geometric mean of the recording latency for each tracer under different workloads are presented. G.M. refers to the geometric mean.

| | | Bench-1 | Browser | Camera | Bench-2 | Desktop | Install | Video-1 | eShop-1 | Bench-3 | eShop-2 | LockScr. | StartApp. | eShop-3 | Video-2 | News | Video-3 | Game | IM | Blog-1 | Blog-2 | G.M. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Latest | BTrace | 10.8 | 10.6 | 10.6 | 11.2 | 11.0 | 10.7 | 11.0 | 10.8 | 10.6 | 10.6 | 10.8 | 10.9 | 10.8 | 10.6 | 10.4 | 11.0 | 11.1 | 10.6 | 10.2 | 10.9 | 10.8 |
| | BBQ | 11.6 | 11.6 | 11.7 | 11.7 | 11.6 | 11.6 | 11.6 | 11.5 | 11.6 | 11.5 | 11.6 | 11.6 | 11.5 | 11.5 | 11.5 | 11.6 | 11.6 | 11.6 | 11.6 | 11.5 | 11.6 |
| | ftrace | 3.4 | 7.1 | 5.2 | 5.6 | 3.3 | 4.3 | 3.3 | 6.8 | 4.1 | 5.8 | 3.8 | 3.2 | 7.6 | 7.2 | 6.5 | 4.9 | 7.5 | 7.5 | 7.6 | 8.0 | 5.4 |
| | LTTng | 2.8 | 7.3 | 6.5 | 5.4 | 3.2 | 3.7 | 0.1 | 6.7 | 4.2 | 0.3 | 3.7 | 3.1 | 7.9 | 1.8 | 0.7 | 5.2 | 6.8 | 7.4 | 0.4 | 1.2 | 2.5 |
| | VTrace | 0.1 | 0.8 | 0.3 | 0.3 | 0.5 | 0.1 | 0.2 | 0.8 | 0.3 | 0.2 | 0.7 | 0.0 | 0.4 | 0.2 | 0.2 | 0.2 | 0.6 | 0.4 | 0.1 | 0.4 | 0.3 |
| Loss Rate | BTrace | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | BBQ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | ftrace | 0.81 | 0.41 | 0.83 | 0.92 | 0.76 | 0.84 | 0.84 | 0.58 | 0.90 | 0.51 | 0.78 | 0.83 | 0.43 | 0.38 | 0.56 | 0.91 | 0.15 | 0.57 | 0.49 | 0.49 | 0.60 |
| | LTTng | 0.81 | 0.38 | 0.82 | 0.92 | 0.76 | 0.84 | 0.84 | 0.57 | 0.90 | 0.46 | 0.78 | 0.83 | 0.31 | 0.40 | 0.53 | 0.91 | 0.19 | 0.53 | 0.56 | 0.62 | 0.60 |
| | VTrace | 0.89 | 0.85 | 0.95 | 0.93 | 0.79 | 0.85 | 0.91 | 0.90 | 0.93 | 0.92 | 0.80 | 0.86 | 0.95 | 0.96 | 0.96 | 0.95 | 0.75 | 0.95 | 0.93 | 0.94 | 0.90 |
| #Fragments | BTrace | 47 | 86 | 87 | 53 | 34 | 49 | 60 | 101 | 49 | 73 | 35 | 83 | 69 | 111 | 87 | 68 | 73 | 70 | 86 | 54 | 65 |
| | BBQ | 2 | 21 | 106 | 3 | 3 | 4 | 7 | 243 | 12 | 169 | 1 | 105 | 222 | 66 | 45 | 236 | 122 | 152 | 76 | 385 | 34 |
| | ftrace | 2e4 | 1e4 | 2e4 | 1e4 | 1e4 | 1e4 | 3e4 | 1e4 | 2e4 | 1e4 | 2e3 | 3e4 | 8e3 | 1e4 | 1e4 | 2e4 | 6e3 | 1e4 | 9e3 | 1e4 | 1e4 |
| | LTTng | 2e4 | 9e3 | 1e4 | 1e4 | 1e4 | 1e4 | 3e4 | 9e3 | 2e4 | 1e4 | 3e3 | 3e4 | 6e3 | 1e4 | 1e4 | 2e4 | 7e3 | 1e4 | 2e4 | 2e4 | 1e4 |
| | VTrace | 7e4 | 8e4 | 7e4 | 7e4 | 4e4 | 4e4 | 7e4 | 8e4 | 7e4 | 9e4 | 1e4 | 8e4 | 8e4 | 9e4 | 7e4 | 8e4 | 6e4 | 9e4 | 8e4 | 9e4 | 6e4 |
| G.M. Lat. (ns) | BTrace | 54 | 52 | 52 | 53 | 55 | 56 | 56 | 51 | 52 | 52 | 56 | 53 | 52 | 52 | 53 | 54 | 53 | 52 | 51 | 52 | 53 |
| | BBQ | 111 | 138 | 495 | 119 | 95 | 102 | 148 | 510 | 115 | 894 | 88 | 485 | 828 | 738 | 794 | 447 | 520 | 822 | 763 | 776 | 324 |
| | ftrace | 62 | 63 | 64 | 62 | 62 | 62 | 63 | 63 | 61 | 64 | 61 | 63 | 64 | 64 | 65 | 64 | 66 | 65 | 63 | 64 | 63 |
| | LTTng | 263 | 251 | 246 | 260 | 266 | 266 | 260 | 250 | 251 | 227 | 271 | 251 | 236 | 238 | 239 | 250 | 250 | 232 | 243 | 228 | 249 |
| | VTrace | 278 | 296 | 283 | 264 | 238 | 229 | 235 | 360 | 264 | 359 | 207 | 290 | 355 | 308 | 332 | 321 | 300 | 341 | 297 | 359 | 292 |

**Tracers.** We compared *BTrace* with four state-of-the-art tracers, including the Function Tracer framework from the Linux kernel v5.15 (ftrace) [25], the Linux Trace Toolkit Next Generation Userspace Tracer v2.13 (LTTng) [12], the VampirTrace framework v5.14.4 (VTrace) [30], and the recently proposed block-based bounded queue (BBQ) [45], whose overwrite mode can be considered a core buffering mechanism for tracing frameworks.

### 5.1 Self Comparison

As explained in Sec. 3, both the closing and skipping mechanisms can lead to a drop in memory utilization and effectivity ratio, which in turn reduces the size of the latest fragment. To quantitatively investigate this, we ran *BTrace* with a varying number of active blocks ($A$), ranging from 1× (i.e., $A$=12) to 64× (i.e., $A$=768) of the core number (i.e., 12), under both core-level and thread-level replay. We measured the size of the latest fragment across these 20 workloads, and the results are presented as a box plot in Fig. 10.

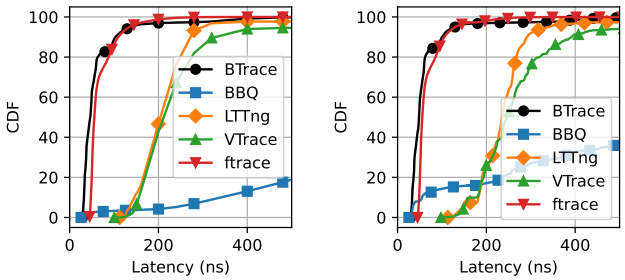
As shown in the figure, both a small and a large number of active blocks lead to a decrease in the size of the latest fragment. When the number of active blocks is small, the varying trace production speeds between cores result in frequent closures of partially filled blocks, thereby lowering buffer utilization. For thread-level replaying, the outcomes are significantly worse compared to core-level replaying, as it also experiences frequent skipping, further reducing buffer utilization, resulting in more outliers in the figure. Conversely, with a large number of active blocks, the primary issue is the reduction in effectivity ratio. For instance, when the number of active blocks is set to 64×, the size of the latest fragment drops to 8 MB, which is dominated by the theoretical value of 9 MB (75%×12 MB) according to the equation in Table 2. Therefore, we select a sweet spot (i.e., at 16×) that has the maximum effectivity ratio across all scenarios as our empirical parameter and apply it in subsequent experiments, as well as in production.

### 5.2 State-of-the-art Comparison

We then compare *BTrace* with state-of-the-art tracers using the thread-level replaying. Table 2 shows the results across workloads. A darker color indicates a better result.

**Latest fragment (the higher the better).** The size of the latest fragment highlights the effective and reliable traces available for locating defects. BBQ shows the nearly ideal cases where the geometric mean size of the latest fragment for all workloads is 11.6 MB, close to the buffer size of 12 MB. The difference is due to the memory overhead of the buffer's metadata. *BTrace* achieves the second-best size of the latest fragment, with an average size of 10.8 MB, which is 6.90% lower than BBQ (yet *BTrace* exhibits much lower latency, as shown in the latency comparison below), mainly due to the closing and skipping mechanism discussed in §3.

Per-core buffers, such as ftrace and LTTng, have average sizes of 5.4 MB and 2.5 MB, respectively, 55% and 78% lower than *BTrace*. Moreover, although they yield similar results for some workloads (e.g., Browser and eShop1), LTTng has significantly lower results for others (e.g., Video-1)




**(a)** eShop-2 workload. **(b)** Overall latency.

**Figure 11.** Recording latency CDF (in nanoseconds).

due to oversubscription, which causes entries to drop unexpectedly. In workloads characterized by a more skewed trace-producing speed, as evidenced by Fig. 4, e.g., LockScr., the results for ftrace and LTTng drop significantly to 3.2 MB and 3.1 MB, while *BTrace* has a size of 10.8 MB, which is 2.4× larger than both. VTrace performs the worst, with an average size of only 0.3 MB.

**Loss rate (the lower the better).** We then measure the loss rate, which indicates how much data is lost within the collected range, i.e., from the oldest to the newest, as well as the probability of losing crucial clues. The ideal case is 0%.

As shown in the table, *BTrace* and BBQ achieve results close to the ideal case, with a loss rate of less than 1% (omitted to 0% due to precision rounding). For ftrace and LTTng, the loss rate is 60%, while for VTrace, the value reaches as high as 90%. For some heavy workloads like Video-3, the loss rates for ftrace, LTTng, and VTrace are 91%, 91%, and 95%, respectively. The high loss rate significantly increases the probability of losing crucial clues, making trace analysis and issue localization more challenging.

**Fragments number (the lower the better).** The table further illustrates the number of fragments for each tracer, highlighting the potential for unnoticeably dropping crucial clues. The results reveal that, on average, distributed buffers contain over $10^4$ fragments across all scenarios, whereas *BTrace* averages only 65 fragments. Additionally, when considering the latest continuous results, it is notable that most fragments in *BTrace* are concentrated towards the tail of the traces, which reduces their impact on locating defects.

**Recording latency (the lower the better).** We evaluate the recording latency for each tracer. Threads may be preempted during trace writing, which can cause significantly longer latency (e.g., 1000×). To mitigate the impact of these outliers and make the results more reasonable, we calculate the geometric mean of the latency for each recording.

As shown in the table, *BTrace* achieves the lowest latency (53 ns per entry), which is 20% lower than that of the second-best, ftrace (63 ns). The mean latency of LTTng and VTrace is 249 ns and 292 ns, respectively, which are 4.7× and 5.5× longer than that of *BTrace*. The global buffer tracer BBQ exhibits the highest latency, with a mean latency of 324 ns. However, for workloads with high oversubscription, the latency of BBQ increases significantly. For example, for the e-shop2 workload, BBQ reaches a mean latency of 894 ns.

To better highlight the latency differences, we present the latency Cumulative Distribution Function (CDF) for the e-shop2 workload, along with the overall CDF results, in Fig. 11. As shown in the figure, *BTrace* consistently exhibits the lowest latency at the 50th and 99th percentiles, whereas BBQ shows the highest latency.

## 6 Case Study from Production

*BTrace* has been successfully deployed in production smartphones and is activated anonymously during specific suspicious scenarios reported by users, helping to identify hard-to-trigger defects. Tracing is enabled only in beta releases and requires user consent, similar to previous works [32]. It involves over 100,000 users and can uncover extremely rare bugs — such as those occurring once every million hours — that were not detected during pre-deployment testing. However, it is strictly not enabled in the general availability release to protect users' privacy. All traced data is anonymized, securely stored, and managed by the operating system, with its security ensured through isolation mechanisms.

By reserving a 450 MB buffer in *BTrace*, we could store traces for over 30 seconds without noticeably influencing user experiences. An analysis of over 4,000 issues from the defect tracking system for beta releases confirms that *BTrace* effectively identifies over 200 long-duration cause-effect bugs, which are challenging to locate using ftrace due to its impractical memory requirements for such long-duration traces. We present three representative real-world issues that were uncovered through *BTrace*.

**Energy defects.** Energy defects are located by measuring energy consumption across various scenarios over extended periods. Using *BTrace*, we successfully captured relevant events (e.g., frequency adjustments, idle decisions, thermal information) in suspicious scenarios during beta releases, which enabled us to identify numerous corner cases that contribute to energy defects. One notable case is that middle cores frequently enter a deep idle state in certain scenarios, and subsequently, user-experience-critical threads (e.g., render threads [14]) are scheduled to run on them, triggering the cores to wake up. However, before the cores fully activate, these threads experience timeouts and are prematurely migrated to the big cores due to an overly aggressive scheduling strategy. This frequent migration leads to substantially higher energy consumption in such scenarios. *BTrace* helps capture events over an extended period, enabling us to locate such an issue through statistical analysis.

**Frame drops.** Frame drops are complex issues arising from multiple factors. While some stem from single-point defects

(e.g., deadlocks), many originate from root causes that occur long before the symptoms appear. For instance, *BTrace* identified a periodically misbehaving thread (e.g., busy looping) in beta releases, which increased chip temperatures before silently terminating. This temperature rise triggered the heat management daemon to unexpectedly downscale the CPU frequency, leading to frame drops. Since the thread has already terminated long before the frame drop occurs, the root cause is likely to be overwritten with existing tracers.

Another case involves the intricate dependencies among render threads, which can become blocked in rare scenarios. Trace analysis revealed that render threads occasionally acquire a lock held by a memory reclaim thread, which, in turn, may be blocked by yet another thread. These dependencies often span extended durations and can only be revealed by recording all the necessary traces using *BTrace*.

**Silent defects.** We deploy several daemons to detect silent defects, which do not exhibit observable behavior and can only be identified through timeout strategies. Daemons responsible for drivers are configured with a timeout of approximately 10 seconds, while those monitoring freezing and wake-up events have timeouts exceeding 20 seconds. By utilizing *BTrace*, we can store traces throughout the timeout processes, enabling the identification of several hard-to-trigger silent defects in beta releases. One notable example occurs when smartphones fail to freeze, prompting the daemon to report the issue after 20 seconds. Investigation of the traces revealed that specific bound CPU threads failed to migrate in corner cases, resulting in starvation after the userspace driver hot-unplugs the CPU.

## 7 Conclusion and Future Work

Amid the growing complexity of smartphone systems, developers face the severe limitations of existing tooling, in particular tracers. Taking into account the unique characteristics imposed by smartphone systems, *BTrace* revisits the state-of-the-art tracers and finds a novel design between the memory efficiency of global buffers and the performance of per-core buffers.

Nevertheless, *BTrace* is not limited to smartphones. It can also be applied to emerging servers with hundreds of cores, where varying core utilization may lead to significant space wastage in existing tracers with distributed buffers. Specifically, most tasks in servers are executed on only a few cores but tend to migrate frequently across cores. As a result, tracing these tasks requires allocating sufficient space across all cores, which leads to considerable space wastage. *BTrace* can efficiently trace these tasks in such scenarios.

## Acknowledgments

We would like to express our gratitude to our shepherd, Alec Wolman, and the anonymous reviewers for their thoughtful and constructive comments.

## References

[1] The LTTng Documentation. https://lttng.org/docs/v2.13/#doc-what-is-tracing. Accessed 17 October 2024.
[2] Apple. iPhone 16. https://www.apple.com/my/iphone-16/.
[3] ARM. ARM DynamIQ Redefines Multi-Core Computing. https://www.arm.com/technologies/dynamiq. Accessed 17 October 2024.
[4] Mona Attariyan, MIchael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, Hollywood, CA, October 2012. USENIX Association.
[5] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *2004 USENIX Annual Technical Conference (USENIX ATC 04)*, Boston, MA, June 2004. USENIX Association.
[6] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.
[7] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 465–485, Santa Clara, CA, July 2024. USENIX Association.
[8] George E Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.
[9] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 17–32, Carlsbad, CA, October 2018. USENIX Association.
[10] Mathieu Desnoyers. Implementation of LTTng ringbuffer. https://github.com/lttng/lttng-ust/blob/a6a42d4fcf48abb9f6ea6331cb6208e279366a28/src/common/ringbuffer/ring_buffer_frontend.c#L2260. Accessed 17 October 2024.
[11] Mathieu Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009.
[12] Mathieu Desnoyers and Michel Dagenais. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Linux symposium*, volume 101, 2008.
[13] Mathieu Desnoyers and Michel R Dagenais. Lockless multi-core high-throughput buffering scheme for kernel tracing. *ACM SIGOPS Operating Systems Review*, 46(3):65–81, 2012.
[14] Android Developers. Rendering in Android. https://developer.android.com/topic/performance/rendering. Accessed 17 October 2024.
[15] Olivier Dion. LTTng: The challenges of user-space tracing. In *Tracing Summit*, 2023.
[16] Linux Kernel Docs. Energy Aware Scheduling. https://docs.kernel.org/scheduler/sched-energy.html. Accessed 17 October 2024.
[17] Perfetto DOCS. ATrace: Android system and app trace events. https://perfetto.dev/docs/data-sources/atrace. Accessed 17 October 2024.
[18] Perfetto DOCS. Tracing 101. https://perfetto.dev/docs/tracing-101. Accessed 17 October 2024.
[19] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 311–326, New York, NY, USA, 2011. Association for Computing Machinery.
[20] OpenAtom Foundation. OpenHarmony Project. https://docs.openharmony.cn/pages/v4.1/en/OpenHarmony-Overview.md. Accessed 17

October 2024.
[21] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.
[22] Google. AOSP overview. https://source.android.com/docs/setup/about. Accessed 17 October 2024.
[23] Huawei. Huawei p60. https://consumer.huawei.com/cn/phones/p60/.
[24] HUAWEI. HUAWEI Mate 60 Pro Specs. https://consumer.huawei.com/cn/phones/mate60-pro/specs/, 2023.
[25] Red Hat Inc. ftrace - Function Tracer. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html.
[26] Red Hat Inc. Monitoring processes for performance bottlenecks using perf circular buffers. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/creating-custom-circular-buffers-to-collect-specific-data-with-perf_monitoring-and-managing-system-status-and-performance. Accessed 17 October 2024.
[27] Intel. Performance Hybrid Architecture. https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-architecture.html. Accessed 17 October 2024.
[28] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 582–598, New York, NY, USA, 2017. Association for Computing Machinery.
[29] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: a technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 344–360, New York, NY, USA, 2015. Association for Computing Machinery.
[30] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pages 139–155. Springer, 2008.
[31] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.
[32] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 787–803, Carlsbad, CA, July 2022. USENIX Association.
[33] Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. Break dancing: low overhead, architecture neutral software branch tracing. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 122–133, New York, NY, USA, 2021. Association for Computing Machinery.
[34] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.
[35] OPPO. OPPO A18. https://www.oppo.com/en/smartphones/series-a/a18/, 2023. Accessed 17 October 2024.
[36] Insung Park and Ricky Buch. Event tracing-improve debugging and performance tuning with ETW. *MSDN magazine*, page 81, 2007.
[37] Perfetto. System profiling, app tracing and trace analysis. https://perfetto.dev/. Accessed 17 October 2024.
[38] Mateusz Piotrowski. Benchmarking performance overhead of dtrace on freebsd and ebpf on linux.
[39] The Chromium Projects. The Trace Event Profiling Tool. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/. Accessed 17 October 2024.
[40] Precedence Research. Smartphones Market Size, Share and Trends 2024 to 2034. https://www.precedenceresearch.com/smartphones-market. Accessed 17 October 2024.
[41] Steven Rostedt. Code of ftrace ring buffer. https://github.com/torvalds/linux/blob/master/kernel/trace/ring_buffer.c. Accessed 17 October 2024.
[42] Sumsung. Galaxy S23. https://www.samsung.com/my/smartphones/galaxy-s23/.
[43] The Linux Kernel documentation. Perf ring buffer. https://docs.kernel.org/userspace-api/perf_ring_buffer.html. Accessed 17 October 2024.
[44] Anvinraj Valiyathara. How much ram is ideal for optimal performance? https://www.croma.com/unboxed/how-much-ram-is-good-for-a-phone. Accessed 17 October 2024.
[45] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 249–262, Carlsbad, CA, July 2022. USENIX Association.
[46] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 193–207. USENIX Association, July 2021.
[47] R.W. Wisniewski and B. Rosenburg. Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 3–3, 2003.
[48] Xiaomi. Redmi Note 12. https://www.mi.com/global/product/redmi-note-12/specs/, 2023. Accessed 17 October 2024.
[49] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 320–336, New York, NY, USA, 2021. Association for Computing Machinery.
[50] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 335–350, Boston, MA, July 2018. USENIX Association.
[51] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, Boston, MA, April 2023. USENIX Association.
[52] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581, New York, NY, USA, 2017. Association for Computing Machinery.