2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Scaler: Efficient and Effective Cross Flow Analysis

Jiaxun Tang
jtang@umass.edu
University of Massachusetts Amherst
Amherst, MA, USA

Mingcan Xiang
mingcanxiang@umass.edu
University of Massachusetts Amherst
Amherst, MA, USA

Yang Wang
wang.7564@osu.edu
Meta/The Ohio State University
Columbus, OH, USA

Bo Wu
bwu@mines.edu
Colorado School of Mines and HiTA
AI Inc
Golden, CO, USA

Jianjun Chen
jianjun.chen@bytedance.com
Bytedance
San Jose, CA, USA

Tongping Liu
tongping.liu@bytedance.com
Bytedance
San Jose, CA, USA

ABSTRACT

Performance analysis is challenging as different components (e.g., different libraries, and applications) of a complex system can interact with each other. However, few existing tools focus on understanding such interactions. To bridge this gap, we propose a novel analysis method–"Cross Flow Analysis (XFA)"– that monitors the interactions/flows across these components. We also built the `Scaler` profiler that provides a holistic view of the time spent on each component (e.g., library or application) and every API inside each component. This paper proposes multiple new techniques, such as Universal Shadow Table, and Relation-Aware Data Folding. These techniques enable `Scaler` to achieve low runtime overhead, low memory overhead, and high profiling accuracy. Based on our extensive experimental results, `Scaler` detects multiple unknown performance issues inside widely-used applications, and therefore will be a useful complement to existing work.

The reproduction package including the source code, benchmarks, and evaluation scripts, can be found at https://doi.org/10.5281/zenodo.13336658.

ACM Reference Format:
Jiaxun Tang, Mingcan Xiang, Yang Wang, Bo Wu, Jianjun Chen, and Tongping Liu. 2024. Scaler: Efficient and Effective Cross Flow Analysis. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3691620.3695473

1 INTRODUCTION

Modern systems are enormously complex. A user program typically interacts with different libraries in the whole system, such as standard libraries, the memory allocator, and other third-party libraries. Studying the interactions between these components has a twofold indication on the performance. On the one hand, application developers may use some inappropriate library APIs with hidden performance issues. On the other hand, the extraordinary behavior of API invocations can be utilized to infer inefficient algorithm design or configurations of applications and libraries.

Although a significant amount of profilers have been proposed in the past, none of them focuses on the interactions of components. Some existing tools focus on the performance issues related to hardware, such as cache misses [Chabbi et al. 2018; Liu and Berger 2011; Liu et al. 2014; Roy et al. 2018; Zhou et al. 2022]; Some detect the multithreading-related performance issues [Alam et al. 2017; Curtsin ger 2015; Zhou et al. 2018], mainly on thread-related APIs; Some may report the time spent on user functions via the sampling mechanism [corporation 2022; Graham et al. 1982a; Levon 2021]. `ltrace` is possibly the most related work [Linux Community 2013], but introduces over 6195× performance overhead. Such high overhead makes it implausible to identify issues caused by library APIs accurately.

Instrumentation Technique	Tool	Collection Frequency	Overhead	
			Runtime	Memory
Sampling via Hardware PMUs	`perf`	0.2%	22.5%	6.7×
Sampling via timer + `ptrace`	`vtune-ums`	0.0%	41.4%	18.3×
eBPF + software breakpoint	`bpftrace`	100%	28.8×	3.1×
`ptrace` + software breakpoint	`ltrace`	100%	> 6195.7×	-
Universal Shadow Table	`Scaler`	100%	20.3%	15.5%

Table 1: Compare with existing work.

We propose a novel method called ***Cross Flow Analysis*** **(`XFA`)** that monitors and analyzes the interactions between different components (called **cross flow**) in the system, where the component is either the application itself or any library. `XFA` helps identify potential performance problems caused by libraries (e.g., incorrect API uses or inappropriate configuration), and inefficient algorithm design of applications (e.g., data structures). More specifically, `XFA` proposes to trace all Application Interfaces (APIs) of involved libraries, including the number and runtime of invocations of each API. To assist the performance analysis, `XFA` summarizes the runtime/invocations of each API and each component and provides two views: a component view shows *the time (and percentage) of one component spending on other related components*, and an API view of a component shows *the time (and percentage) spent on any API inside this component.*

Figure 1 shows an example report for the `canneal` application of PARSEC [Bienia et al. 2008], which is a **new** bug reported by `Scaler`: the `libstdc++` library is the most time-consuming library (from the component view), and `strcmp` consumes 77% runtime

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695473

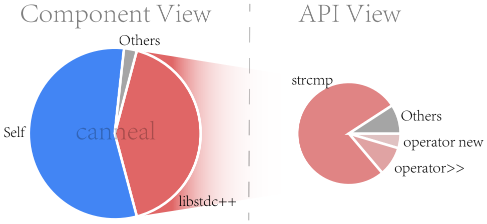


Figure 1: Scaler's report for canneal.

of libstdc++ (from the API view). Since canneal simulates an algorithm that optimizes the routing cost of circuit boards, it is certainly *abnormal* to have such a large portion of time spent on string comparisons. *Such an issue originates from an inappropriate design* when canneal counts the appearance of each string in the input file: the std::map (red-black tree) is used to store the strings, where the searching/inserting operations and tree balancing operations incur a significant number of string comparisons; Instead, a better method is to utilize std::unordered_map (hash map). Changing the data structure not only reduces string compares, but also reduces the last-level cache misses by 18%. The combined effect improves the performance by 52%. That is, the abnormal behavior of library APIs helps expose the inefficient algorithm design inside user programs. In contrast, perf fails to detect this issue because of its excessively coarse sampling rate, resulting in the inaccurate report of the runtime for libstdc++ and strcmp. It reports that only 6.32% of time is spent in libstdc++ and only 0.56% time is spent in strcmp.

Based on the idea of XFA, we further built a profiler – Scaler – that monitors API invocations. Our profiler requires overcoming the following technical challenges.

Challenge 1: how to trace different types of library API invocations without recompilation and knowing the signatures of APIs? Compiler-based instrumentation requires to recompile all involved components [Graham et al. 1982a], which is often infeasible because of the lack of source code, build scripts and proper build environments. General binary instrumentation (e.g., Pin [Luk et al. 2005]) and ptrace-based technique (e.g., ltrace [Linux Community 2013]) easily introduce orders of magnitude performance overhead. DITool [Serra et al. 2000] requires signatures of the profiled APIs to hook APIs. This paper proposes a light-weight binary instrumentation technique to trace different types of APIs. For dynamic linking, Scaler replaces binary entries of the related ELF sections in memory; For dynamic loading, Scaler intercepts dlsym (and dlopen) so that it can redirect API invocations to the universal interceptor, as described in Section 2.3.

Challenge 2: how to trace API invocations with low runtime overhead? Scaler utilizes the same interceptor to handle different types of API invocations, but requires storing the events of each API invocation separately and keeping the invocation hierarchy. To satisfy this requirement, **Universal Shadow Table** is proposed to efficiently intercept APIs, as shown in Figure 2. For each API defined by .rela.plt, .rela.dyn and dynamically loaded by dlsym, Scaler maps them to a shadow entry in the Universal Shadow Table. Each shadow entry consists of multiple binary instructions that can store API-specific information required by Scaler. By using Universal Shadow Table, Scaler only introduces around 20% overhead but intercepts around 63 million invocations per second. Key design choices will further be discussed in Section 2.3.

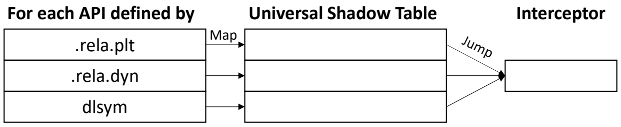


Figure 2: Overview of Universal Shadow Table.

Challenge-3: how to record intensive API invocations with minimal storage overhead while maintaining accuracy? One common method of tracing used by ltrace [Linux Community 2013] is to append one event after the other. However, recording all API invocations will impose high storage and performance overhead. This overhead is caused by frequent API invocations at around 63 million per second. Instead, Scaler proposes the **Relation-Aware Data Folding** with the following attributes: (1) During the recording phase, all API invocations will be summarized at runtime and output at the end. This mechanism prevents a proportional increase in storage/memory volume over time. (2) The recording will maintain the relative relationship between APIs and libraries. In cases where the same API is invoked by different libraries, the summary will only group invocations originating from the same library together. This attribute helps identify the performance problem inside one specific component, that is, maintaining the accuracy. Overall, Scaler only imposes around 16% memory overhead via its Relation-Aware Data Folding.

Scaler also includes other contributions, such as handling the runtime attribution of the multithreaded applications, and handling corner cases like irregular API invocation. Based on our extensive evaluations, Scaler imposes around 20% performance overhead. Scaler's memory overhead is around 16%, which is orders of magnitude less than perf, where perf imposes around 7× memory overhead. Overall, Scaler identified six bugs in the evaluated applications, whereas perf could only find one of them. Among these bugs, two were unknown bugs. Overall, this paper makes the following contributions:

(1) It proposes a novel **Cross-Flow Analysis (XFA)** method to help users understand cross-component interactions. Such cross-flow analysis will benefit the identification of inappropriate API usage, and help identify some incorrect algorithm designs and configurations.
(2) It proposes a series of novel techniques together to address the performance and memory challenges, including but not limited to **Universal Shadow Table**, and **Relation-Aware Data Folding**.
(3) Extensive experiments have been performed on a range of real-world applications. These experiments show that Scaler imposes low performance and memory overhead while effectively detecting multiple unknown performance issues related to cross-component interactions.

The remainder of this paper is organized as follows. Section 2 provides an overview of our approach. Section 3 further discusses implementation details. Section 4 presents our effort to evaluate the effectiveness and runtime/memory overhead of Scaler. Section 5 discusses the compatibility, extensibility, and limitations of Scaler. Finally, Section 6 reviews related work, and Section 7 concludes the paper.

2 BACKGROUND AND OVERVIEW

In this section, we briefly discuss the background of API invocation mechanisms. Then, we introduce the basic idea of XFA and Scaler.

2.1 Background of API Invocations

In current software systems, programmers are not required to write all programs from scratch. Instead, they could employ external libraries to quickly develop the software system by invoking “Application Programming Interface (API)”. APIs define the methods and data formats that applications can use to request and exchange information with external components or libraries. There are two common mechanisms for invoking external APIs: dynamic linking and dynamic loading, as detailed in the following.

2.1.1 Dynamic Linking. In dynamic linking, libraries are not included directly in the executable binary during compilation (unlike static linking). Instead, the references to the specific functions or symbols in external libraries are resolved at runtime. It typically relies on a dynamic linker (`ld-linux.so`), often referred to “linker” interchangeably in this paper, to resolve symbol addresses in executable files and shared libraries during program execution. Dynamic linking related APIs can be categorized by two sections in the ELF file: `.rela.plt` and `.rela.dyn`.

For APIs defined by the `.rela.plt`, there are two address resolution modes [Levine 2001]: in eager mode, the dynamic linker resolves the address of all APIs before the program execution. In lazy mode (the default mode), the dynamic linker will postpone the address resolution of an API until the first invocation time. Every time a component (the application or a library) invokes an API, the linker will first determine whether the memory address of that API has been resolved. If not, it will resolve the address with the help of ELF section `.plt` and `.got.plt`.

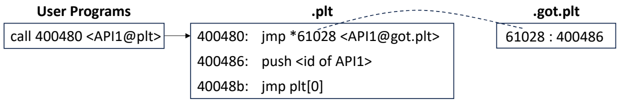


Figure 3: `.rela.plt` API Invocation via `.plt` and `.got.plt`.

The whole process is illustrated in Figure 3. Each `.plt` entry has three executable instructions, while each entry in `.got.plt` only stores one address. In particular, each `.plt` entry will have the following three instructions: the first instruction is a jump instruction (like “`jmp *0x61028`”) that fetches the memory address stored in the `.got.plt`. The other two instructions will push the index of the `.plt` entry onto the stack (e.g., “`push x`”) and then jump to the starting entry of the `.plt`, which is used to invoke the linker. The linker will replace the `.got.plt` entry with the resolved address before jumping to the API, so subsequent API invocations will only need to execute the first jump instruction in the `.plt` entry. For eager mode, the `.got.plt` entry will be filled with the real API address before the main function starts. In this way, the jump instruction (like “`jmp *0x61028`”) will directly invoke the API’s address.

For APIs defined by the `.rela.dyn`, the linker will always resolve the API address before the main function starts. But the linker will store resolved address inside `.plt.got` or `.got` rather than the ELF sections corresponding to the `.rela.plt`. The instructions used to call APIs defined by `.rela.dyn` are also different. The whole process is illustrated in Figure 4. The call instruction will fetch the address stored in `.plt.got` or `.got` and directly jump to it. Another prominent difference between `.rela.dyn` and `.rela.plt` is that `.rela.dyn` not only defines API functions but also global variables, while `.rela.plt` only defines API functions.

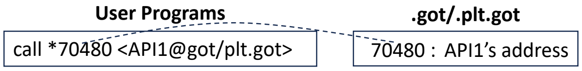


Figure 4: `.rela.dyn` API Invocation via `.got` and `.plt.got`.

2.1.2 Dynamic Loading. Dynamic loading is the process of loading a shared library into memory at runtime after the program has started executing. In particular, the user program first invokes `dlopen` to get the handle of a specific library (by passing the name of the shared library file), and then explicitly requests the address of a function by invoking the `dlsym` function.

2.2 Cross-Flow Analysis (XFA)

As discussed above, XFA monitors interactions between different components in the whole system stack, where each component (either the application or a library) is treated as an island. We observe that API invocation is the way for a component to interact with the external world. Therefore, we propose to intercept all API invocations in order to collect the data for performance analysis. More specifically, XFA collects the accurate runtime and number of invocations for each API. In the end, XFA provides two views to help the user diagnose the performance issue. One is *component view*, which provides the following information: how much time one component spends on itself and other components. Another is *API view* that presents the runtime distribution information of all APIs in a library.

2.3 Overview of Scaler

Scaler is a detailed implementation/profiler of XFA. Scaler aims to satisfy the following requirements:

- **Least Manual Effort:** To use Scaler, users do not need to recompile or change any code or use custom OS or hardware.
- **Maximum Generality:** Scaler should support different types of API invocations, such as dynamic linking and loading, but does not rely on customized linker or the availability of symbol information or the source code.
- **Low Recording Overhead:** Scaler should not introduce high performance and memory overhead that can prevent its usage in the production environment, even given a significant number of API invocations.

In order to achieve the least manual effort, we could not employ the compilation-based approach as that requires the source code, build scripts, and the proper system environment, which is not always accessible. We also cannot utilize the preloading technique, as that will require Scaler to know all function signatures beforehand. To satisfy maximum generality, we should not rely on a particular version of dynamic linker/loader or hardware performance counters. Further, existing hardware performance counters typically utilize sampling as the basic method, which cannot provide the full tracing functionality. After excluding all profiling methods, only binary instrumentation (like Pin [Luk et al. 2005] or DynamoRIO [Bruening et al. 2003]) and software-based process tracing (e.g., ptrace [Wikipedia 2023]) exist in existing work. However, both general binary instrumentation (like Pin [Luk et al. 2005] or DynamoRIO [Bruening et al. 2003]) and process tracing are notoriously known for their high performance overhead, easily over 20×, which is even prohibitively high for development phases.

To meet all of these requirements, Scaler employs a Selective Binary Instrumentation strategy, that only instruments the locations related to API invocations. By using selective binary instrumentation, there is no need to recompile the code, rely on the custom hardware, and couple with a specific version of the linker or loader. By limiting the scope of instrumentation, Scaler reduces the instrumentation overhead compared to general binary instrumentation that instruments or checks the whole binary. Further, Scaler's selective binary instrumentation supports different modes of dynamic linking and dynamic loading. For APIs defined by .rela.plt, Scaler instruments the binary of the corresponding .plt entries. For APIs defined by .rela.dyn, the address resolution occurs before the program's execution begins. For these scenarios, Scaler performs the binary instrumentation by changing the corresponding .got/.plt.got entry directly. Scaler instruments within the program's startup code, just before the main() function, so that the address resolution of these APIs has been completed. For dynamic loading, since dlsym() typically returns the resolved address to the program, Scaler re-direct the return addresses to the custom common interceptor by instrumenting and intercepting the dlsym invocations.

Note that selective binary instrumentation alone is not sufficient to guarantee the low runtime overhead. In addition to the instrumentation, Scaler requires handling each API differently, such as storing the events of each API invocation, and then returning back to the invocation site. Unfortunately, it is not easy to complete these tasks efficiently. For instance, existing library interposition approach, like DITool [Serra et al. 2000], redirects .got.plt entries to custom functions directly. DITool requires the user to redefine each custom function with the same signatures so that they can handle the above-mentioned tasks, which is clearly not generalizable enough. In the development of Scaler, we have tried to utilize the hash table to locate the location of storing events, but this method imposed a large overhead when multiple entries are mapped to the same bucket.

To overcome the generalization and performance issue, Scaler proposes **Universal Shadow Table** that maps each API of different types to one shadow entry in the Universal Shadow Table, as shown in Figure 2. The shadow entry encloses all necessary information for each specific API, such as the storing location, the jumping target, and the returning target after the interception. That is, Scaler only needs to rely on the shadow entry to parse the callee (API) information in constant time. Since the size of each shadow entry is much larger than the size of a normal entry (e.g., 16 bytes for a .plt entry), it can include more information inside, overcoming the size limit of the original entry. Universal Shadow Table also makes it possible to utilize a common interceptor (as shown in Figure 2) to handle different types of APIs. Overall, the Universal Shadow Table achieves the maximum generality and low performance overhead.

Another potential issue for the profiling is the memory or storage overhead, as existing work typically appends the recorded events one after the other, causing a proportional increase in storage/memory volume over time. Scaler proposes the **Relation-Aware Data Folding** to reduce its memory/storage overhead while maintaining its accuracy. This design is based on the observation of Scaler's major purpose: Scaler requires the understanding of the time (and percentage) one component spent on other related components (component view), and the time (and percentage) spent on a specific API inside a library (API view). Therefore, Scaler could summarize the events of each API together, instead of appending the events into the log file and performing the analysis offline. By summarizing the events of each API together, Scaler's storage overhead does not increase proportionally over the total runtime. As mentioned above, Scaler's recording maintains the relative relationship between APIs and libraries. In cases where the same API is invoked by different libraries, the summarization will only group invocations originating from the same library together. That is the reason why such a method is called 'Relation-Aware Data Folding".

3 DESIGN AND IMPLEMENTATION

In this section, we introduce the major components and the implementation of Scaler.

As shown in Figure 5, Scaler includes a runtime library and an offline visualizer. To use Scaler, there is no need to recompile and change user programs and any library. Scaler can be linked with the user program by specifying Scaler's executable via the LD_PRELOAD environment variable. The runtime library further includes multiple components: Interceptor, Tracer, Universal Shadow Table, and Online Data Folder. These components interact as follows: The interceptor handles different modes of API invocation and redirects the execution to the Universal Shadow Table. The Universal Shadow Table contains assembly code that can record the necessary information needed by the Tracer. The Tracer is responsible for tracing and collecting the information of each API invocation. The Online Data Folder is responsible for storing the events in a memory-efficient way and outputs all events to an external log file at the end of the execution. Scaler also includes an offline visualizer to analyze the data and generate component and API views offline. As Scaler's online data folder already summarizes the recorded data online, the visualizer can analyze the recorded data very quickly.

3.1 Interceptor

As mentioned above, Scaler intercepts different types of API invocations caused by dynamic linking and dynamic loading. The purpose of the interceptor is to redirect different types of API invocation to the Universal Shadow Table.

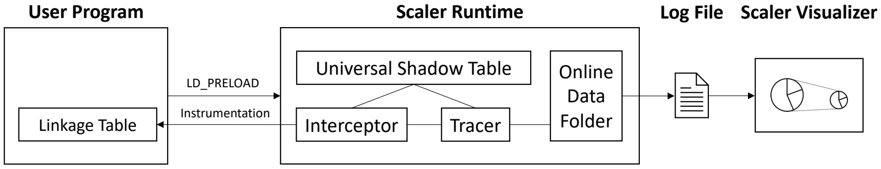


Figure 5: Overview of `Scaler`.

3.1.1 Dynamic Linking. In UNIX-like systems, dynamic linking defines API in two areas: `.rela.plt` and `.rela.dyn` as detailed in Section 2.1.

For APIs defined by the `.rela.plt` section, there are two different modes: eager and lazy mode. In eager mode, the APIs' addresses have been resolved before entering the main routine, whereas in lazy mode, the address resolution will not happen until the program reaches the `.plt` entry. Despite this difference, the user programs always use the same instruction (e.g., "`call func@plt`") to invoke APIs. That is, both modes will execute instructions in the `.plt` section. The `Interceptor` intercepts these APIs by replacing each entry of `.plt` with two instructions, which will redirect the execution to the `Universal Shadow Table`.

For APIs defined by the `.rela.dyn` section, the user program will use a different instruction (e.g., "`call *func@plt.got`") to invoke the API. That is, the API invocations will bypass the `.plt`. The linker will always resolve the APIs' address before the `main` routine and place them in `.plt.got`/`.got`. To intercept APIs defined by the `.rela.dyn` section, `Scaler` directly replaces the content in `.plt.got`/`.got` with the address of the `Universal Shadow Table`.

3.1.2 Dynamic Loading. API invocations of dynamic loading typically go through `dlopen` and `dlsym`. These two functions are intercepted by instrumenting the corresponding `.plt` entries as discussed above.

`Scaler` defines its custom `dlopen` function. When `dlopen` is invoked, `Scaler` not only checks the just opened library but also the dependency libraries imported implicitly by the new library. `Scaler` will scan all newly imported libraries and hook their APIs as well.

`Scaler` defines its custom `dlsym` function. When `dlsym` is invoked, `Scaler` allocates an entry in the `Universal Shadow Table` and then returns the entry's address to the user programs. That is, whenever user programs invoke the function pointer returned by `dlsym`, the control logic will be passed to the `Universal Shadow Table`. Note that for APIs invoked by `dlsym`, there is no way to know beforehand which APIs will be opened this way. Therefore, `Scaler` can only allocate shadow entries on demand.

3.1.3 Handling Abnormal Cases . In order to ensure high reliability, `Scaler` also handles the following abnormal cases.

Supporting irregular API invocation: Some compiler optimizations may cause the program to use `jmp API@plt` instruction rather than `call API@plt` instruction when invoking APIs. With `jmp` instruction, the return addresses will not be pushed to the stack and the API will never return. `Scaler` detects this problem by comparing the return address location on the stack. `call` instruction will always push the address of the next instruction as the return address to the stack, so if we observe that two consecutive API invocations has the same return address stored at the same location on the stack, then `Scaler` will know the API is invoked with `jmp` instruction and will not return as well.

Support no-return APIs: Some `glibc` APIs never return back to the caller. One typical no-return API is `exit()`, which terminates the program or a thread and will not cause performance overhead. `Scaler` chooses not to intercept all functions marked by "__noreturn" in the `glibc` library, because like `exit`, no-return APIs in `glibc` will not be the root cause of performance problems and often does not work like usual functions. If the user program invokes a no-return API, `Scaler` can still work correctly by comparing the return location as mentioned above.

Identify functions in .rela.dyn: As mentioned in Section 2, `.rela.dyn` not only defines API functions but also global variables. And there is no flag to effectively distinguish the two. `Scaler` detects whether an entry defined by `.rela.dyn` is API by checking whether it points to an executable memory region.

3.2 Universal Shadow Table (UST)

The Universal Shadow Table holds one shadow entry for each API defined in `.rela.plt`, `.rela.dyn` and dynamically loaded via `dlsym()`, as shown in Figure 2. The UST is critical in `Scaler`'s runtime.

Each shadow entry includes a set of assembly instructions (134 bytes in total) that implements the following functionalities, before jumping to the `Tracer`.

- **Reading per-thread context in TLS (20 bytes):** `Scaler` checks the per-thread context at first. If the context is not initialized, `Scaler` skips the tracing and invokes the corresponding API directly. Note that `Scaler` keeps the per-thread data for all API invocations. If the per-thread context is not initialized, then it is infeasible to update the tracing data.
- **Increment the number of API invocations and record timestamp (45 bytes):** `Scaler` increments the number of invocations for the current API. By default, `Scaler` always records the number of invocations. At the end of this code section, `Scaler` checks whether it is necessary to perform the timing, if yes then `UST` will invoke `Tracer` to record the timestamp before API execution. For extensibility, `Scaler` allows users to configure the frequency of collecting the runtime.
- **Invoking the real API (31 bytes):** `Scaler` only invokes the real API directly when the per-thread context is not initialized or timing is not required based on the setting. `Scaler` handles differently for different types of API invocations. For

dynamic loading (via dlsym) or after address resolution in dynamic linking, the execution will jump to the API address directly. For API invocation in dynamic linking (before address resolution), the UST will simulate the behavior of the original .plt entries and invoke ld-linux.so to resolve the address. The return address of the real API is recorded and replaced with the current UST entry so that the real API will return to the UST after execution. Note that before invoking the real API, the UST needs to save necessary registers (context) beforehand, and then recover them before returning to the caller.

- **Record the duration and return (**38 **bytes):** UST will invoke Tracer again to record the timestamp after API invocation finishes. UST passes information required by the Tracer by pushing them onto the call stack. After Tracer finishes, UST will jump back to the real return address.

3.3 Tracer

The Tracer component is responsible for tracing and collecting the information of each API invocation by the Universal Shadow Table.

Scaler keeps separate data for each thread. In this way, there is no need to utilize mutex locks for the update, as different threads are not updating the same tracing data simultaneously. Further, such a design reduces cache misses caused by true/false sharing [Liu and Berger 2011], but at the cost of more memory/storage consumption. The per-thread data will be stored in separate files in the end, and the Offline Visualizer will integrate all data from different threads together in the end. Note that Scaler employs initial-exec TLS model [PeterDing [n. d.]] to store per-thread variables, which only require a single mov instruction to access. In contrast, the default model (dynamic TLS) requires an extra function invocation called __tls_get_addr, which is very inefficient.

To collect the execution time, Scaler employs the light-weight rdtsc instruction that can read CPU clock cycles in user space efficiently, avoiding expensive system calls. It collects the timestamp before and after each API execution, and the difference between these two timestamps is the execution time of the current API invocation. The Tracer will invoke the Online Data Folder to record the collected information.

3.4 Online Data Folder

Scaler designs a data structure that folds the traced data online. The data folding will follow two principles: (1) it reduces the memory/storage as much as possible, which also helps reduce the time of offline analysis. (2) It preserves the accuracy of traced data. In the end, The data folder outputs the traced data to the disk files at the end of the execution or upon receiving the signal from users. Each thread outputs one copy of data.

Efficient data folding without losing the accuracy: Scaler aims to report the time consumption and number of invocations for each API invoked by any component (library or application). We have the following **two observations** on API invocations: (1) the number of APIs that can be invoked by a component is constant, equaling to the total of its linkage table entries, and dlsym invocations, which remains unchanged at different execution time. (2) One API can be invoked by different components; For example, pthread_mutex_lock can be invoked by the application itself or different libraries. Scaler's design is built on these two observations. Based on the first observation, it utilizes an array-based structure to track the invocation information accumulatively for APIs, which could be determined by analyzing the corresponding elf files. That is, Scaler's memory/storage overhead is proportional to the number of linkage table entries and the number of dlsym-opened functions. Based on the second observation, Scaler tracks the invocations of the same API by different components separately, preserving the accuracy.

Handle abnormal program exits: As mentioned above, Scaler persists per-thread data to the disk when a thread exits. To intercept thread exits, the intuitive method is to replace the standard thread creation with a wrapper function so that it can invoke the real thread function inside and handle thread exits correspondingly. However, this method cannot intercept abnormal exits of threads. For instance, some programs may invoke pthread_exit() explicitly to terminate threads (e.g., aget [PeterDing 2022]), and some children threads never exit (e.g., OpenMP). To handle these abnormal cases, Scaler registers a common exiting handler via __cxa_thread_atexit. For never-exiting threads, the main thread persists all remaining threads on their behalf when it exits.

Attributing the runtime of invocations differently for serial and parallel phase: Obviously, an API invoked in the serial phase or in the parallel phase makes different performance impacts on the end-to-end performance. However, existing profilers, e.g., perf, typically summarize (and calculate the percentage) the execution time of different APIs, which cannot reveal the inherent performance issues of multithreaded programs inside [Curtsin ger 2015]. Scaler takes into account the difference for invocations occurring in serial and parallel phases. More specifically, in the recording phase, Scaler divides the execution time of API invocation by the number of active threads in the parallel phase, and utilizes the original execution time for invocations in the serial phase.

3.5 Offline Visualizer

Scaler's offline visualizer includes Python scripts used to generate component views and API views based on the log files generated by Scaler runtime. As discussed above, the offline visualizer integrates the per-thread data together. Since Scaler already attributes the runtime of invocations for multithreaded programs as discussed in Section 3.4, it simply summarizes the data from different threads together in its offline analysis.

The component view shows the execution time for itself ("Self") and other components. For instance, the component view for the application shows the percentage of runtime spent in the application and other libraries that are invoked directly by the application, while the component view for a library also includes the runtime spent in the library ("Self") and other libraries called by this library. In particular, the runtime of "Self" equals the total runtime of this component minus the runtime spent on all APIs invoked by this component. For the runtime of a specific library, Scaler simply summarizes the runtime of all APIs belonging to this library, which can be collected by analyzing the corresponding elf file.

For the API view, Scaler simply aggregates per-thread data together for all APIs. As mentioned in Section 3.3, each thread has a copy of the data that records API invocations by this thread. Since different threads employ the same array-based data structure for tracking invocations, Scaler only needs to summarize the data with the same index together (indicating the same API) in different per-thread arrays.

Note that Scaler's component view displays the waiting time (shown as "Wait") separately instead of counting them as normal API invocations of pthreads library. The API invocations related to the waiting (e.g., condition/barrier waits) indicate that programs are not actively doing useful work, leading to serious performance issues. Separating the waiting time into a distinct category helps identify such issues as mentioned in Section 4.2. Further, Scaler summarizes the waiting time from different types of threads together, reporting load imbalance issues when different types of threads have significantly different amounts of waiting time. Scaler learns such a mechanism from SyncPerf [Alam et al. 2017]. Note that Scaler is able to achieve this due to its capability of the full trace.

4 EVALUATION

This section aims to answer the following research questions, by comparing with other existing work:

- Can Scaler detect some performance bugs? (Section 4.2)
- What is the runtime overhead of Scaler? (Section 4.3)
- What is the memory overhead of Scaler? (Section 4.4)

4.1 Experimental Setting

Hardware/Software Platform. We performed all experiments on a machine with dual 20-core,40-hyperthread Intel® Xeon® Gold 6230 CPUs, installed with 256GB memory. The system version is Ubuntu 18.04.6, with the kernel version 5.4.0-146-generic. The compiler used is gcc/g++ 7.5.0-3, with the optimization level "-O3".

Evaluated Applications: We chose PARSEC benchmark suite [Bienia et al. 2008] version 3.0-20150206, including 13 multithreaded programs, and multiple real-world applications, including memcached-1.6.17, MySQL-8.0.31, nginx-1.23.2, Redis-7.0.4, and SQLite-3.39.4. For the evaluation, most PARSEC applications (except dedup and ferret) use 80 threads and the default parameters. Both dedup and ferret use 16 threads for each pipeline stage, with 51 and 67 threads in total. For Memcached, we use the memtier-1.4.0 client that runs for 60 seconds. For MySQL, we use sysbench-1.0.20 client to run oltp_read_write test for 10 tables of size 10000. For nginx, we use the wrk-4.1.0.3 client that runs 10, 240 connections for 60 seconds. For Redis, we use redis-benchmark-5.6.0.16 to send 100, 000 requests in total. For SQLite, we use threadtest3 for the evaluation.

4.2 Effectiveness Evaluation of Scaler

We evaluated all the above-mentioned applications to confirm Scaler's effectiveness and compared the result with perf. Such comparison shows the necessity of our proposed lightweight full-trace method. We only show the comparison with perf due to the following reasons: first, perf does not require recompilation of the application and libraries, which is as convenient as Scaler. Second, other full trace based profilers have significantly higher runtime overhead compared to Scaler, making it unnecessary for the comparison. Instead, perf, a sampling based profiler, has a similar performance overhead as Scaler. Other sampling based tools have both lower collection frequency and higher runtime overhead compared to perf.

Overall, Scaler detects 6 bugs in highly-optimized benchmarks, as shown in Table 2, while perf can only detect one bug. Note that 2 out of these 6 bugs are reported for the first time. After fixing these bugs, the performance improves between 25.5% and 76.7% (as shown in the Column "Speedup"). Among these bugs, 4 bugs are related to the design and implementation of the main application, indicating that *the abnormal behavior of APIs can be utilized to infer the inappropriate design or configuration of the application.* The other two performance bugs (dedup-3 and swaptions) are related to performance issues caused by external libraries. The results show the effectiveness of Scaler. Note that Scaler may miss bugs caused by non-API functions or hardware, as discussed in Section 5.1.

In the remainder of this section, we will study performance issues listed in Table 2. Since we already discussed the canneal bug in Section 1 and Figure 1, this example will be skipped. Further, since the root cause of dedup-2 [Alam et al. 2017] and ferret is similar, we only describe the ferret example here.

4.2.1 *Case Study 1: Inefficient I/O Operations of* dedup .

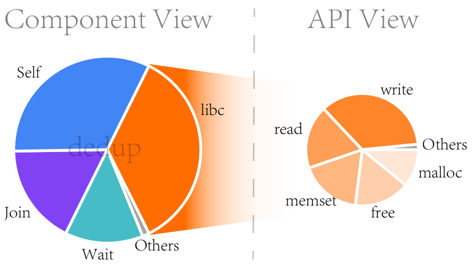
Scaler reports a **new** performance bug (dedup-1) in dedup, which deduplicates and compresses files through a four-stage pipeline, including fragmentation, deduplication, compression, and data reordering [Bienia et al. 2008]. Figure 7a shows the report of Scaler: the application spends 35% time on the glibc library, which is even higher than 33% of "Self"; In the API view of the glibc library, read accounts for 18% of the total execution time and write accounts for 35% of the execution time; Further, Scaler reports that write was invoked 1, 109, 852 times in total. From such a report, we could infer that the application invokes extensive read() and write() to process large files. In fact, a more efficient alternative is to utilize the mmap API to map files to the memory and then operate on it directly with pointers, since the alternative will eliminate the overhead of system calls, and leverage page caching and COW semantics of the memory system. After changing to the method of using mmap, the performance is improve by 49.1%. In contrast, perf cannot find this bug, because it only reports that 4.12% of time is spent on write(), possibly due to its coarse sampling rate.

4.2.2 *Case Study 2: Extensive* madvise *Invocations of* dedup.

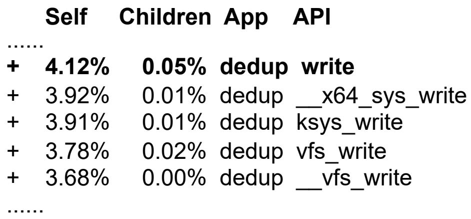
dedup has another known performance problem when linked with the memory allocator of glibc-2.21 [Gorman 2015]. For this case, Scaler's output can be seen in Figure 7a. The allocator spends 68% time on two APIs, e.g., madvise and mprotect. The underlying reason for this issue is that the allocator frequently releases the allocated virtual memory (madvise) back to the OS based on a threshold. But madvise needs to acquire the per-process lock of protecting memory regions inside the OS, and introduces extensive page faults when such a region is accessed again. The acquisitions of the lock introduce high kernel contention with page faults and other memory-related system calls (e.g., mprotect). By increasing the threshold (via the configuration), we can significantly reduce

BugId	Abnormal Behavior	Root Cause	perf	Scaler	New Bug	Speedup
canneal	Extensive string comparisons	Improper data structure of application	✗	✓	✓	51.6%
dedup-1	Extensive time on read()/write()	Inefficient I/O operations of application	✗	✓	✓	49.1%
dedup-2	Imbalance waiting time	Improper thread assignment of application	✗	✓	✗	25.5%
dedup-3	Extensive madvise calls	Improper configuration of library	✗	✓	✗	74.2%
ferret	Imbalance waiting time	Improper thread assignment of application	✗	✓	✗	76.7%
swaptions	Significant lock time	Improper configuration of library	✓	✓	✗	44.0%

Table 2: Effectiveness comparison between Scaler and perf.

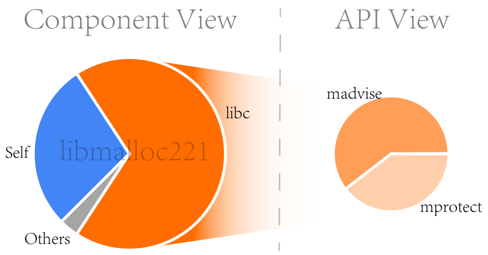


(a) Scaler's output.

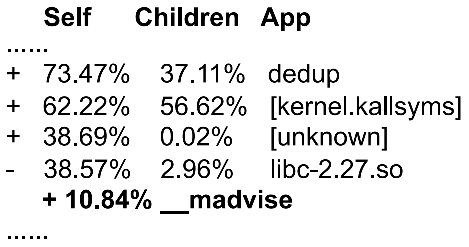

```
   Self    Children  App    API
......
+  4.12%   0.05%     dedup  write
+  3.92%   0.01%     dedup  __x64_sys_write
+  3.91%   0.01%     dedup  ksys_write
+  3.78%   0.02%     dedup  vfs_write
+  3.68%   0.00%     dedup  __vfs_write
......
```

(b) perf's output.

Figure 6: Profiling output for dedup-1.

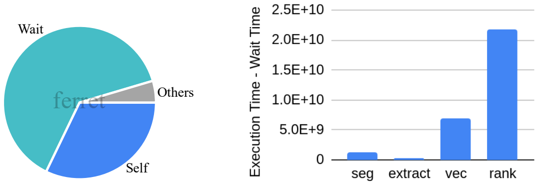


(a) Scaler's output.


```
   Self     Children   App
......
+  73.47%   37.11%     dedup
+  62.22%   56.62%     [kernel.kallsyms]
+  38.69%   0.02%      [unknown]
-  38.57%   2.96%      libc-2.27.so
     + 10.84% __madvise
......
```

(b) perf's output.

Figure 7: Profiling output for dedup-3.



(a) Wait time exceeds execution time

(b) Imbalance of thread execution time

Figure 8: Scaler's output for ferret

the amount of madvise invocations and kernel contention correspondingly. The fix improved the performance by 74.2%. perf cannot reveal this bug, since madvise only accounts for 10.84% time, which is even lower than 17.66% of memset.

4.2.3 Case Study 3: Thread Imbalance of ferret*.* Scaler reports a thread-imbalance problem in ferret that implements a content-based similarity search: Figure 8a shows that ferret spends over 50% time on the waiting, and Figure 8b further reveals that different thread groups have different effective execution time, where rank's effective execution time is about 16× higher than that of seg. Therefore, this is a clear indication of thread-imbalance problem, as observed by SyncPerf [Alam et al. 2017]. We fixed this issue by adjusting the thread assignment from the default 16:16:16:16 (related to seg:extract:vec:rank threads) to 3:1:15:45. This fix improves the performance by 76.7%. In contrast, perf's report cannot reveal this problem, since it does not summarize the runtime of different thread groups together and its too-coarse sampling mechanism.

4.2.4 Case Study 4: Improper Configuration of The hoard *Allocator.* swaptions of PARSEC has a known performance bug, caused by the improper configuration of the hoard allocator [Zhou et al. 2022]. The component view of libhoard shows that it spends 93% time on the pthread library. The API view of the pthread shows that the spin lock takes 99% of the time. Therefore, this is a clear lock contention issue caused by the allocator. We can fix this issue by changing the SUPERBLOCK_SIZE flag from 4096 to 65536 and EMPTINESS_CLASSES from 8 to 2. After the fix, the performance improved by 44%. Note that perf can also reveal this bug, since it also reports the large percentage of time is spent inside the spin lock of libhoard.

4.3 Performance Overhead of Scaler

In this section, we further evaluate the performance overhead of Scaler, and compare it with existing work. For perf, we use sampling rate 4000 and the "-g" option (in order to collect the callstack). For vtune, we use the default sampling interval – 10ms and select

Category	Application	perf	vtune-ums	bpftrace	ltrace	Scaler
PARSEC	blackscholes	16.3%	32.2%	183.0×	> 4953.8×	35.0%
	bodytrack	21.3%	28.4%	34.0×	N/A	16.1%
	canneal	12.9%	12.6%	14.8×	N/A	79.3%
	dedup	31.8%	73.0%	1.4×	N/A	6.2%
	facesim	11.8%	20.8%	26.3×	N/A	15.8%
	ferret	9.8%	17.5%	43.6×	N/A	7.9%
	fluidanimate	16.0%	24.5%	174.8×	N/A	37.8%
	freqmine	18.2%	35.4%	15.0×	N/A	28.6%
	raytrace	3.2%	5.2%	5.5×	N/A	10.1%
	streamcluster	44.7%	37.9%	5.3×	N/A	14.4%
	swaptions	31.6%	56.7%	4.4×	> 7437.7×	33.8%
	vips	1.3×	1.8×	2.4×	N/A	-0.6%
	x264	37.5%	1.7×	4.1%	N/A	14.3%
Real-world Application	memcached-1.6.17	0.8%	2.1%	5.5×	N/A	20.0%
	MySQL-8.0.31	4.5%	1.7%	1.8×	N/A	0.1%
	nginx-1.23.2	5.3%	30.9%	59.7%	N/A	5.1%
	Redis-7.0.4	9.4%	9.8%	46.7%	N/A	20.5%
	SQLite-3.39.4	2.0%	0.8%	48.6%	N/A	21.9%
Overhead	-	22.5%	41.4%	28.8×	> 6195.7×	20.3%

Table 3: Performance overhead of Scaler and others.

Category	Application	Scaler (Baseline)	perf
PARSEC	blackscholes	5.02E+09	1.13E+06
	bodytrack	2.23E+09	5.28E+06
	canneal	1.20E+09	1.75E+06
	dedup	1.19E+07	2.63E+05
	facesim	4.40E+09	1.55E+07
	ferret	2.17E+09	3.40E+06
	fluidanimate	1.35E+10	1.09E+07
	freqmine	2.60E+08	2.27E+06
	raytrace	1.67E+09	3.17E+06
	streamcluster	3.11E+08	2.79E+07
	swaptions	3.74E+09	2.90E+06
	vips	2.36E+07	1.72E+06
	x264	3.10E+05	4.28E+05
Real-world Applications	memcached-1.6.17	3.83E+08	9.34E+05
	MySQL-8.0.31	6.85E+08	1.84E+06
	nginx-1.23.2	8.02E+03	2.38E+05
	Redis-7.0.4	4.77E+08	9.23E+04
	SQLite-3.39.4	3.51E+08	1.46E+05
Avg Counts	-	2.03E+09	4.44E+06
Avg Freq	-	6.29E+07	1.05E+05

Table 4: Number of events recorded by Scaler and perf.

the default “User Mode Sampling (ums)” option. For bpftrace, we use Scaler to collect the list of all invoked APIs, and then write a bpftrace script to attach uprobe and uretprobes for all of these APIs. For ltrace, we use “–no-signals -o /dev/null” flag to minimize the impact of signal and output.

4.3.1 Overhead of Online Tracing. Table 3 shows the performance overhead of Scaler and other existing work. We evaluate Scaler, perf and vtune-ums all run 8 times and we report the average. bpftrace introduce daunting performance overhead, so we can only test it once. We randomly picked two applications in PARSEC to profile with ltrace and both cannot finish within 24 hours. We also observed freezes on real-world applications when profiling with ltrace. Consequently, we did not test ltrace for all programs, since two evaluated programs all require more than 24 hours to finish. Overall, Scaler introduces 20.3% performance overhead, which is the least among all evaluated tools.

Scaler’s lower overhead can be attributed to its efficient internal design, which includes the selective binary instrumentation method that minimizes binary instrumentation, the Universal Shadow Table that minimizes data attribution overhead, and the pure user-space profiling without involving the context switch overhead. In comparison, both bpftrace and ltrace employ the software breakpoint technique (e.g., INT3) that requires saving and restoring the user context for each API invocation. The difference is that ltrace involves context switching between the ltrace tool, the traced program, and the kernel while bpftrace uses ebpf to process data directly in the kernel to avoid the context switch between kernel space and user space. Even with in-kernel processing, the context switch overhead still imposes high overhead due to the extensive number of API invocations (around 62.9 million each second as shown in Table 4).

For sampling-based profilers, perf’s overhead is around 22.5%, and vtune-ums’s overhead is around 41.4%. Both of them impose a higher overhead than Scaler, even Scaler provides a full trace functionality and collects orders of magnitude more events. perf utilizes the hardware Performance Monitoring Units (PMUs) to sample the execution status, and collects the callstack at each sample. Similarly, it also involves context switches between user space and kernel space. vtune-ums is also a sampling-based tool but with the default sampling rate that is 10× lower than perf. The underlying reason for its high overhead is that it utilizes the slow ptrace system call to collect the information.

Difference of Recorded Events. Scaler provides the full trace functionality, collecting a significantly higher number of events than sampling-based tools, e.g., perf. Table 4 shows the difference between Scaler and perf. On average, perf only records 105 thousand events each second, while Scaler records 62.9 million events each second. That is, Scaler collects data 599× more frequently than perf, while imposing less performance overhead.

4.3.2 Overhead of Offline Analysis. We further compared the performance overhead of offline analysis between Scaler and perf. Scaler’s offline visualizer (writing with Python) only takes an average of 0.43 seconds to execute, while perf’s offline analysis takes 33.3 seconds on average. That is, perf’s offline analysis is around 76× slower than Scaler. The underlying reason is that Scaler performs the majority of computation online via its relation-aware data folding. In contrast, perf saves the call stack of every sample and aggregates the recorded events offline.

4.4 Memory Overhead of Scaler

We also evaluated the memory overhead of Scaler, perf [sandar Milenk ovic 2012], vtune-ums [corporation 2022] and bpftrace [ioviser 2013]. The results can be seen in Table 5. Overall, Scaler introduces orders of magnitude lower memory overhead compared to other existing work. Specifically, Scaler only imposes 15.5% memory overhead. In comparison, perf’s memory overhead is 6.7×, vtune-ums’s overhead is around 18.3×, and bpftrace’s overhead is 3.1×. We observe that perf and vtune-ums has relatively high memory overhead for application with small memory footprints, while Scaler still provides low memory overhead for these applications. The major reason for Scaler’s low overhead is due to its Relations-Aware Data Folding. The major memory overhead for Scaler is proportional to the number of APIs. For each API, Scaler allocates one universal shadow table (134 bytes) entry and one struct that records API-specific information (112 bytes). All runtime data are dynamically folded at runtime and do not need extra space for recording.

Category	Application	Original	perf	vtune-ums	bpftrace	Scaler
PARSEC	blackscholes	617MB	1.3%	8.0%	17.2%	2.6%
	bodytrack	43MB	16.1×	16.1×	2.5×	11.9%
	canneal	858MB	1.0%	8.4%	12.4%	1.6%
	dedup	1564MB	-0.7%	10.3%	13.3%	8.0%
	facesim	337MB	4.0×	1.4×	49.5%	3.2%
	ferret	142MB	2.0×	4.0×	75.5%	6.2%
	fluidanimate	1020MB	4.1%	11.0%	10.4%	1.1%
	freqmine	3388MB	0.8%	2.5%	4.5%	0.8%
	raytrace	1292MB	0.7%	5.5%	8.2%	0.2%
	streamcluster	116MB	20.9×	5.6×	92.2%	25.2%
	swaptions	13MB	24.9×	49.0×	8.3×	81.8%
	vips	225MB	3.6×	2.0×	41.5%	4.9%
	x264	1792MB	0.3%	3.5%	6.0%	2.6%
Real-world Application	memcached-1.6.17	7MB	34.3×	88.3×	15.4×	62.9%
	MySQL-8.0.31	666MB	1.2%	2.8×	17.2%	3.7%
	nginx-1.23.2	8MB	9.1×	78.8×	12.9×	25.0%
	Redis-7.0.4	12MB	3.4×	50.9×	8.6×	16.9%
	SQLite-3.39.4	21MB	1.6×	29.3×	4.9×	21.0%
Overhead	-	-	6.7×	18.3×	3.1×	15.5%

Table 5: Memory overhead of Scaler and others.

Category	Application	perf-8000Hz overhead	perf-4000Hz overhead	Max output difference
PARSEC	blackscholes	18.1%	16.3%	0.21%
	bodytrack	45.7%	21.3%	0.15%
	canneal	15.0%	12.9%	0.18%
	dedup	7.9%	31.8%	0.40%
	facesim	19.1%	11.8%	0.05%
	ferret	11.2%	9.8%	0.27%
	fluidanimate	17.0%	16.0%	0.09%
	freqmine	21.3%	18.2%	0.32%
	raytrace	4.5%	3.2%	1.33%
	streamcluster	69.5%	44.7%	2.24%
	swaptions	22.9%	31.6%	0.08%
	vips	83.2%	1.3×	0.41%
	x264	33.0%	37.5%	2.57%
Real-world Applications	memcached-1.6.17	3.4%	0.8%	0.51%
	mysql-8.0.31	9.9%	4.5%	0.39%
	nginx-1.23.2	14.6%	5.3%	0.15%
	redis-7.0.4	1.8%	9.4%	0.71%
	sqlite-3.39.4	2.9%	2.0%	0.14%
Avg Overhead	-	22.3%	22.5%	-
Avg Output Diff	-	-	-	0.57%

Table 6: The impact of increasing the sampling rate on the output and performance for perf.

4.5 Sampling rate of perf

We further evaluate the impact of perf sampling rate on the profiling result and the runtime overhead. In theory, setting a higher sampling rate will improve profiling accuracy at the cost of higher runtime overhead. However, in practice perf's sampling rate has limitations and will not strictly follow the frequency specified by the user. In Table 6, we increased the sampling rate from 4000Hz used in Section 4 to 8000Hz and profiled programs under two different sampling rates. We also adjusted the kernel flag "perf_event_max_sample_rate" accordingly to increase the system sampling rate limit. We then used the official "perf diff" tool to calculate the time difference for every function recorded under different sampling rates. This tool normalizes the difference value to 0%-100% and output a percentage value in the report[manual page 2022]. Finally, we parsed the report and find the function with the maximum difference value and reportd in column "Max output difference". From Table 6, we can see that using a 2× higher sampling rate on average only changed the maximum output difference by 0.57%. And the average performance overhead remains the same. This result indicates that using sampling rate 4000Hz in Section 4 is appropriate because it already reached the maximum feasible sampling rate in our experiment environment.

4.6 Corner case analysis

In the remainder of this section, we will analyze several corner cases to help facilitate the understanding of how Scaler components work together to overcome these challenges.

4.6.1 Case Study 1: Some APIs are invoked by the pthread library before the recording context is initialized. pthread library will call malloc to allocate the memory required to construct the newly created thread. These malloc API invocations will still be intercepted because Scaler has replaced the .plt of the pthread library at an earlier time. However, at this time Scaler's per-thread recording context will remain unallocated until the thread construction completes. Scaler can distinguish these pre-mature API invocations using the first 20 bytes of assembly code in Universal Shadow Table as mentioned in Section 3.2. After the recording context has been initialized, all subsequent mallocs invoked by the pthread library can still be recorded and thus skipping a few events during thread construction will not significantly change the profiling result.

4.6.2 Case Study 2: libstdc++6.0.32 *uses "jmp" instruction to invoke APIs.* Library such as libstdc++6.0.32 uses a single "jmp" instruction in the .plt entry for "operator delete(void*)", which is different from the regular .plt introduced in Section 2.1. Scaler can always handle such corner cases because "call" instruction will always change the stack pointer location and "jmp" instruction will not. Scaler can distinguish which API is called by "jmp" and which is not by comparing the location of the return address on the stack. Scaler's "jmp" detection mechanism can generalize to the case where there are multiple "jmp" instructions to invoke an API. For example: funcA -> call API1@plt -> jmp API2@plt -> jmp API3@plt. In this case, the return address will be somewhere inside funcA, and Scaler should return to funcA. Scaler's API interceptor will be invoked three times. In every invocation, Scaler's API interceptor will see that the location of the return address stays the same on the stack. So when Scaler returns to funcA, it will know that API1, API2, and API3 have all finished execution.

5 DISCUSSION

5.1 Limitation

Scaler cannot detect hardware-related performance bugs, e.g., cache issues, especially those ones caused by application code. Such performance bugs can be detected better using perf or CachePerf [Zhou et al. 2022]. Currently, Scaler does not intercept internal functions of applications or APIs of statically linked libraries, because these functions or APIs do not use the linkage table. Since Scaler works inside user space, it does not intercept kernel functions.

Scaler helps identify the number and percentage of API invocations, which share the same purpose as perf or gprof. However, all three tools still require human effort to actually find out the root cause of performance issues.

Supporting static linking libraries is a future research direction, but the proposed techniques such as Universal Shadow Table and

Relation-Aware Data Folding can still be applied to profile statically linked libraries. The major technical challenge is to identify statically linked APIs inside compiled programs. But this problem can be bypassed if the symbol table is provided to `Scaler`. Once the address of a static linking API is known, Scaler needs to replace the first two instructions in the function to make the program jump to the universal shadow table. The instructions used for replacement are similar to what `Scaler` currently uses to replace the `.plt`. Similar to simulating instructions inside the `.plt`, `Scaler` needs to simulate the behavior of the overridden instructions. This simulation can be achieved by copying the replaced instructions to other memory regions and modifying jump targets and relative addressing instructions accordingly. Since `Scaler` only overrides two instructions, the simulation should not significantly impact the performance overhead.

5.2 Extensibility

`Scaler` is easy to extend, and can be connected with different performance analysis tools. For example, it could collect all synchronizations (and parameters) for synchronization analysis [Alam et al. 2017; Zhou et al. 2018]; the whole sequence of API invocations will provide insights for tail performance analysis [Dean and Barroso 2013]. Despite these possibilities, `Scaler` is not designed as the replacement for sampling-based tools (e.g., `perf`) but as a complement to these existing profilers.

6 RELATED WORK

General Profilers. General profilers typically help identify the performance issues of applications, such as `gprof` [Graham et al. 1982a], `Oprofile` [Levon 2021], VIProf [Mousa et al. 2007], `Coz` [Curtsin ger 2015], vtune [corporation 2022], and `perf` [sandar Milenk ovic 2012]. As discussed in Section 1, they typically focus on performance issues of applications but fall short in diagnosing the inefficiency caused by external components. For instance, `gprof` [Graham et al. 1982b] instruments the entry and exit of each internal function with the compiler-assisted instrumentation in order to collect and analyze the execution time of the whole application. However, it explicitly skips the external libraries, as it assumes that libraries are not performance bottlenecks and it is not convenient to recompile all libraries. `perf` [sandar Milenk ovic 2012] cannot precisely diagnose the inefficiency of external libraries due to its coarse-grained sampling. In contrast, `Scaler` identifies the inefficiency of external libraries or internal design issues by abnormal API invocations, which could be complementary to existing profilers.

Holistic Profilers. Holistic profilers focus on providing a holistic view of performance to identify the performance inefficiency in the whole system stack. Stitch [Zhao et al. 2016] profiles the performance of the whole software stack based on the unstructured logs output. Its effectiveness highly relies on the comprehensiveness of logging. In contrast, `Scaler` does not rely on the program code. Caliper [Boehme et al. 2016] is a library-based approach that allows tool developers and users to collect hardware-related events and timestamp information. Caliper requires users to explicitly utilize their provided APIs to collect the data, which is not transparent to users (and therefore different from `Scaler`). `Scaler` complements these profilers with its transparency.

Library API Interposition: There are the following ways of intercepting library API invocations. **Ptrace-based Approaches:** Ptrace-based tools, such as `ltrace` [Linux Community 2013] and `strace` [Linux Community 2020], leverage the `ptrace` system call to intercept functions, without code change. However, typically such tools impose significant performance overhead, e.g., `ltrace` 's overhead is over $6,100\times$, which may introduce correctness issues for data collection. **Library Preloading**: The library preloading allows the interposition of library APIs without changing the source code, such as SyncPerf [Alam et al. 2017] or wPerf [Zhou et al. 2018]. The preloading mechanism requires users to provide the signatures of interception APIs. **Changing Dynamic Loader:** Zaslavskiy et al. [Zaslavskiy et al. 2013] provided a customized dynamic loader to perform performance profiling. However, this method may introduce compatibility issues when the loader is updated. **Binary Instrumentation:** Binary instrumentation, e.g., Pin [Luk et al. 2005] or DynamoRIO [Bruening et al. 2003], can interpose library APIs without the explicit change of applications. However, such approaches typically impose more than $5\times$ performance overhead, making the precise time consumption analysis implausible. **`.got.plt` Interposition:** `DITool` [Serra et al. 2000] proposes to change the addresses stored in the `.got.plt` so that it can re-direct the flow to the user-defined functions, which also do not need the code change or recompilation. However, `DITool` requires users to provide the signatures of intercepting APIs, and it strongly couples with the implementation of `Irix` and `rld` (a specific version of tools). In contrast, `Scaler` overcomes these issues: it can interpose any APIs without the code change or knowing the signatures by employing Selective Binary Instrumentation and Universal Shadow Table.

7 CONCLUSION

Modern systems are complex, since there exist frequent interactions among all components of the whole system stack. This paper proposes a novel cross-flow analysis method (called `XFA`) that helps users understand the interactions of all components. We also implemented a profiler (named `Scaler`) that introduces multiple novel techniques that work together to reduce performance and memory/storage overhead, including Universal Shadow Table, and Relation-Aware Data Folding. Our comprehensive experiments confirm that `Scaler` could identify multiple performance issues that existing tools (e.g., `perf`) cannot detect. Therefore, `Scaler` will be a complement to existing profilers due to its unique property and reasonable overhead.

8 ACKNOWLEDGEMENTS

Many thanks to our shepherd and the anonymous reviewers for their insightful comments. This material is based in part upon work supported by the NSF grant CCF-2118745, CCF-2024253, CNS-2312396, DUE-2215193, and CNS-1750760.

REFERENCES

Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization

Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 298–313. https://doi.org/10.1145/3064176.3064186

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 72–81.

David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 550–560.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, 265–275.

Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-Fly False-Sharing Detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. 152–167. https://doi.org/10.1145/3178487.3178499

Intel corporation. 2022. User-Mode Sampling and Tracing Collection. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

Emery D. Curtsin ger, Charlie andBerger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 184–197. https://doi.org/10.1145/2815400.2815409

Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

Mel Gorman. 2015. malloc: Reduce worst-case behavior with madvise and refault overhead.

Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982a. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/800230.806987

Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 1982b. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*. 120–126.

ioviser. 2013. bpftrace — High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace.

John R Levine. 2001. *Linkers & loaders*. Morgan Kaufmann.

John Levon. 2021. OProfile. https://oprofile.sourceforge.io/about/.

Linux Community. 2013. ltrace(1) — Linux manual page. https://man7.org/linux/man-pages/man1/ltrace.1.html.

Linux Community. 2020. strace(1) — Linux manual page. https://man7.org/linux/man-pages/man1/strace.1.html.

Tongping Liu and Emery D. Berger. 2011. SHERIFF: precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048066.2048070

X. Liu, K. Sharma, and J. Mellor-Crummey. 2014. ArrayTool: A lightweight profiler to guide array regrouping. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 405–415. https://doi.org/10.1145/2628071.2628102

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* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

Linux manual page. 2022. perf-diff documentation. https://www.man7.org/linux/man-pages/man1/perf-diff.1.html.

Hussam Mousa, Chandra Krintz, Lamia Youseff, and Rich Wolski. 2007. VIProf: Vertically Integrated Full-System Performance Profiler. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–6. https://doi.org/10.1109/IPDPS.2007.370513

PeterDing. [n. d.]. TLS Model. https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1?topic=attributes-tls-model-attribute.

PeterDing. 2022. Aget - Asynchronous Downloader. https://github.com/PeterDing/aget.

Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, and Xu Liu. 2018. Lightweight Detection of Cache Conflicts. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/3168819

Alek sandar Milenk ovic. 2012. Perf Tool: Performance Analysis Tool for Linux. lacasa.uah.edu/images/Upload/tutorials/perf.tool/PerfTool.pdf.

Albert Serra, Nacho Navarro, and Toni Cortes. 2000. DITools: Application-Level Support for Dynamic Extension and Flexible Composition. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (San Diego, California) *(ATEC '00)*. USENIX Association, USA, 19.

Wikipedia. 2023. ptrace. https://en.wikipedia.org/wiki/Ptrace.

Mark Zaslavskiy, Edward Ryabikov, and Kirill Krinkin. 2013. Lightweight Linux dynamic libraries profiling technique for embedded systems. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*. 1–5.

Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. {Non-Intrusive} Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 603–618.

Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. WPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, 527–543.

Jin Zhou, Steven Tang, Hanmei Yang, and Tongping Liu. 2022. CachePerf: A Unified Cache Miss Classifier via Hybrid Hardware Sampling. In *SIGMETRICS '22: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*. ACM.