FlowProf: Profiling Multi-threaded Programs using Information-Flow

Ahamed Al Nahian
University of California
Irvine, USA
anahian@uci.edu

Brian Demsky
University of California
Irvine, USA
bdemsky@uci.edu

Abstract

Amdahl's law implies that even small sequential bottlenecks can seriously limit the scalability of multi-threaded programs. To achieve scalability, developers must painstakingly identify sequential bottlenecks in their program and eliminate these bottlenecks by either changing synchronization strategies or rearchitecting and rewriting any code with sequential bottlenecks. This can require significant effort by the developer to find and understand how to fix sequential bottlenecks. To address the issue, we bring a new tool, information flow, to the problem of understanding sequential bottlenecks. Information flow can help developers understand whether a bottleneck is fundamental to the computation, or merely an artifact of the implementation.

First, our strategy tracks memory access conflicts to find over-synchronized applications where redesigning the synchronization strategy on the existing implementation can improve performance. Then, information flow analysis finds optimization opportunities where changing the existing implementation can improve performance of applications that have bottlenecks due to unnecessary memory access conflicts. We implemented this in FlowProf. We have evaluated FlowProf on a set of multi-threaded Java applications where the generated optimization insights achieve performance gains of up to 58%.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Computing methodologies → Concurrent computing methodologies**.

Keywords: Synchronization, Optimization, Scalable Concurrency Control, Dynamic Program Analysis, Profiling

ACM Reference Format:
Ahamed Al Nahian and Brian Demsky. 2024. FlowProf: Profiling Multi-threaded Programs using Information-Flow. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3640537.3641577

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641577

1 Introduction

Developing high-performance multi-core code is difficult—just creating threads and adding locks to shared data structures often does not yield software that scales to more than a handful of cores. Experts have developed a toolbox of advanced techniques ranging from fine-grained locking to using low-level atomic operations to advanced techniques like combining trees that can help scale parallel code. The mainstream multi-core developer is typically not an expert in these techniques and thus is unlikely to even know when the application of these techniques might benefit their code.

Synchronization constructs used by concurrent data structures can sequentialize parts of the computation. Thus, it follows by Amdahl's law that these data structures can have large impacts on the performance of parallel computations [1, 13]. Intel now ships 56 core processors. On such a processor Amdahl's law implies that if an application contains just 2% sequential code, it runs at less than half of the speed of a fully parallel application on one of these processors.

Eliminating sequential bottlenecks is therefore an important aspect of improving the performance of parallel software. Often, the problem is that data structures use simple coarse-grained locking strategies that can introduce unnecessary waits between independent data structure operations. Removing these bottlenecks is a matter of redesigning the synchronization in a data structure to eliminate superfluous lock conflicts[1] by using finer-grained synchronization. However, sometimes sequential bottlenecks are due to actual conflicting memory accesses,[2] and just redesigning synchronization strategies is insufficient to eliminate the bottleneck.

Sequential bottlenecks due to conflicting field accesses can sometimes be eliminated by redesigning the data structure or replacing it with an alternative data structure. For example, a Java programmer might replace uses of `java.util.Hashtable` in which every put or remove operation updates a `count` field with `java.util.concurrent.ConcurrentHashMap`, which has been redesigned to eliminate several sequential bottlenecks.

[1] Lock conflict occurs when two or more threads try to acquire same lock.
[2] A conflicting access occurs when two threads access the same memory location and at least one thread performs a write.

In other cases, the memory access conflicts could be fundamental to the computation and cannot be eliminated. For example, each addition to a hash chain that is used to maintain a secure log fundamentally depends on all previous additions, and thus cannot be parallelized.

Thus, identifying and understanding sequential bottlenecks is an important component of eliminating sequential bottlenecks. Prior work used dynamic analysis to reason about conflicting memory accesses to discover opportunities to improve synchronization [39]. That work was applicable to critical sections protected by statically assigned locks and moreover, was limited to reasoning about changing synchronization strategies and could not discover opportunities to eliminate synchronization bottlenecks by removing unnecessary memory access conflicts using redesigned data structures.

1.1 Our Approach

We propose a new approach to automatically profiling concurrent code to help developers identify and understand sequential bottlenecks. **Our key insight is that understanding the flow of information through a concurrent data structure can help developers understand the dependencies between data structure operations.** In particular, if there is a flow of information in which an input value for operation A is used to compute the output value of operation B, there has to be a chain of memory access conflicts connecting operation A and B and therefore, there should exist some synchronization mechanism between A and B.[3] For the `java.util.HashTable` example, there is a flow of information between `put`, `get`, or `remove` calls on the same key. However, there is no flow of information between `put`, `get`, or `remove` calls on different keys which suggests that the conflicting memory accesses occurring for different keys are not fundamentally required and can be eliminated by changing the implementation.

This paper presents FlowProf, a profiler that uses dynamic analysis to track flow of information between critical sections to help developers optimize sequential bottlenecks. At each critical section invocation, FlowProf creates a new unique *taint value* and attaches this taint value to all of the input values[4] to the critical section. FlowProf maintains the mapping between taint values and critical section invocations and thus the taint values enable FlowProf to determine which critical section invocations provided a given value (or the values used to compute the given value). As the critical section computes on program values, FlowProf propagates the attached taint values along with the corresponding program value. If the program performs a computation that takes program values originating from two different critical section invocations (and thus have two different taint values) as input, FlowProf attaches both taint values to the result value. When a critical section invocation finishes, FlowProf examines the set of taints for the output values[5] of the critical section invocation to determine the set of prior critical section invocations that contributed values that were used to compute the output values. This information is useful because it provides an approximate lower bound[6] on the synchronization that is needed as information flows between threads have to be implemented using some conflicting memory accesses, which must be protected by synchronization.

Information flow analysis is powerful because it can differentiate between conflicting accesses that arise due to internal bookkeeping operations (such as `count` field updates) and conflicting accesses that propagate values that are necessary to implement the externally visible behavior of critical sections. While these bookkeeping operations can propagate information flows between memory accesses inside of data structure operations, they typically do not introduce an information flow between the input of one critical section and the output of another critical section because (1) the flows do not originate from an input value to a critical section and/or (2) the flows do not escape as an output value from a critical section.

Information flow analysis is not intended to replace analyses based on memory access conflicts. Information flow analysis can identify important optimization opportunities that other conflict-based analysis will miss. However, some optimization opportunities require changing the existing implementation which can often require more effort than changing only synchronization strategy on existing implementation. Therefore, FlowProf supports information-flow analysis along with conflict-based analysis as conflict-based analysis provides developers with optimization opportunities that require only optimizing synchronization strategy on existing implementation while information-flow analysis provides developers with additional opportunities by changing the existing implementation to improve performance beyond what conflict-based analysis can provide.

1.2 Contributions

This paper makes the following contributions:

- **Information-Flow Based Profiling:** It introduces the idea of using information-flow based profiling to help developers understand the potential for eliminating sequential bottlenecks in multi-threaded applications.

[3]We note the exception that relaxed atomics could be used to avoid synchronization in a few corner cases.

[4]The input to a critical section is the live variables at the start of the critical section.

[5]The output of a critical section is the live variables at the end of the critical section.

[6]The lower bound is approximate as the taint analysis used by FlowProf would report that the value `y` depends on `x` in the example `y=z-x+x`.

- **Schedule Impact Analysis:** This analysis uses profile information to compute the runtime of the same execution under the weaker dependencies that arise from either the field conflict analysis or the information flow analysis. The schedule impact analysis enables the developer to select the appropriate optimization type to eliminate sequential bottlenecks.
- **Evaluation:** It evaluates FlowProf on several open source multi-threaded benchmark applications. It evaluates both FlowProf's overhead and effectiveness at classifying sequential bottlenecks.

2 Example

The goal of FlowProf is to identify opportunities to improve the performance of multi-threaded code. This requires identifying code that both (1) significantly slows down the computation due to concurrency-related overheads and (2) can be optimized to greatly reduce these overheads. FlowProf uses information flow to help understand what concurrency control is fundamentally required by a computation and what concurrency control is merely required by the computation's current implementation.

```
1  synchronized void push(int value){
2     node n = new node(value);
3     n.next = top;
4     top = n;
5  }
6
7  synchronized int pop(){
8     node oldtop = top;
9     top = top.next;
10    return oldtop.value;
11 }
```

Figure 1. Example `Stack` Implementation

To illustrate, consider the straightforward implementation of a concurrent stack in Figure 1. Since there are conflicting memory accesses in both of the push and pop operations, to make the implementation thread safe the methods are declared as `synchronized` so that the operations are protected by an internal lock per `Stack` object. This lock-based stack implementation scales poorly as it only allows one thread at a time to update the stack.

An interesting observation is that a pop invocation only receives a value that originates from a single push invocation and thus it is conceptually possible for a pop invocation to only synchronize with one push invocation. Indeed, this observation has been leveraged to build highly scalable lock-free stacks that use elimination arrays [17]. Thus, surprisingly it turns out that the conflict in the example stack implementation is not fundamental, but merely an artifact of the standard implementation.

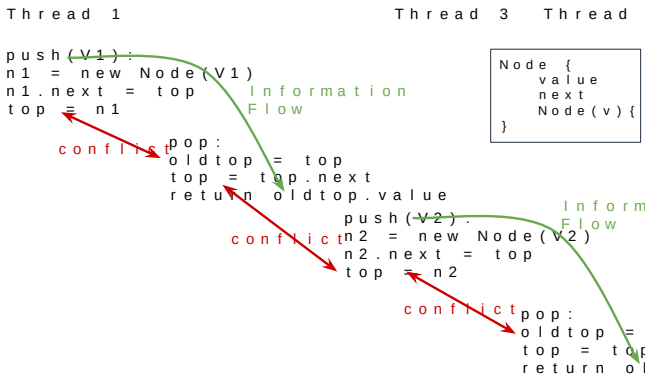

Figure 2. Information Flow vs. Conflicts

2.1 Information Flow in the Example

The stack example shows that analyzing concurrent code in terms of how values flow between critical sections can be a useful tool. It is clear from reading descriptions of concurrent algorithm designs [19] that algorithm designers sometimes think in terms of how parallel algorithms propagate information through data structures.

Information flow allows FlowProf to take a more global perspective of the role of memory accesses in a computation and not limit ourselves solely to local reasoning about the specific loads and stores that the current data structure implementation performs. In information flow analysis, we label each input to a critical section with a taint that is unique to the critical section and then check the taints of the outputs of critical sections. In Figure 1, the input and the output of a critical section are the argument and the return value of the synchronized methods respectively. The taints of the output provides us an approximate lower bound on the values that were necessary to compute the output and which critical section invocation provided those values. Figure 2 shows an example execution in which four threads access the stack. The red edges show that each push or pop operation has a conflicting memory access with the previous push or pop operation. The green edges show that information flows from a push to its paired pop that reads the pushed value. Since, the value of the top pointer, which has memory access conflicts, originated internally from a new memory allocation and did not originate as a value passed into the push method, it remains untainted. Finally, by the means of taint propagation the taint of the push operation would pass as the taint of the returned value of the pop operation creating a information flow from the push to its paired pop. Therefore, the information flow through the output is not impacted by access conflicts on the top field. Information flow provides information that can guide developers to only synchronize an operation with the operations that influenced its output. Hence a pop operation only needs to synchronize with the

push that provided its value. Thus, information flow identifies the opportunity to parallelize the stack computation such that synchronized push-pop pairs can run in parallel.

3 Optimization Insights

FlowProf tracks both memory access conflicts and information flow dependencies between critical sections to provide insights into whether changing a synchronization strategy suffices to improve performance or whether redesigning the code is necessary. Depending on the structure of the conflicts/dependencies, two different patterns can be found from the result of FlowProf's analysis. Each pattern requires different measures to overcome their bottlenecks.

Pattern 1: Over-synchronisation. Programs where a significant portion of the total waiting for synchronisation happens between critical sections with no associated memory access conflicts are over-synchronized. In such programs memory access conflicts rarely happen among overlapping[7] critical sections that compete for the same locks and thus eliminating unnecessary lock conflicts can improve performance. Depending on the state of the existing program implementation, over-synchronization can be addressed in the following ways:

1. **Finer-grained locking:** Implementations that use a coarse-grained locking strategy can benefit from replacing a lock with multiple locks each protecting a subset of the state.
2. **Reader-writer lock:** For read heavy access patterns with rare writes, developers can replace a regular lock with a reader-writer lock to reduce over-synchronization.
3. **Lock elision:** Sometimes a superfluous lock can be removed to eliminate over-synchronization.

Pattern 2: Non-fundamental access conflicts. When a significant portion of the total waiting for synchronisation happens between critical sections in which there are memory access conflicts but no corresponding information flow, we say that such bottlenecks are due to non-fundamental memory access conflicts. In such programs information flows rarely among overlapping critical sections that have conflicting memory accesses. The lack of information flow indicates that the memory access conflicts are not fundamentally required for the actual computation and thus can be eliminated in a different implementation. **Redesigning code or changing the implementation** can improve performance in this case. This process might, for example, replace a use of `java.util.Hashtable` with `java.util.concurrent.ConcurrentHashMap`.

Some applications show neither of the above two patterns. In that case synchronisation waiting is between overlapping critical sections that mediate information flow between threads. In such cases lock conflicts are mostly also associated with memory access conflicts and information flow dependencies. Such a case indicates that the operation's semantics restrain further performance improvement.

We present a schedule impact analysis in Section 6.1 that provides programmers with an estimate of the synchronisation waiting time that takes place because of unnecessary lock conflicts and waiting time that takes place because of unnecessary memory access conflicts. The result of the schedule impact analysis can be matched with the above two patterns to suggest corresponding optimizations to eliminate bottlenecks.

[7]Two critical sections overlap if their execution time intervals (from the start of waiting for locks up to release of locks) overlap.

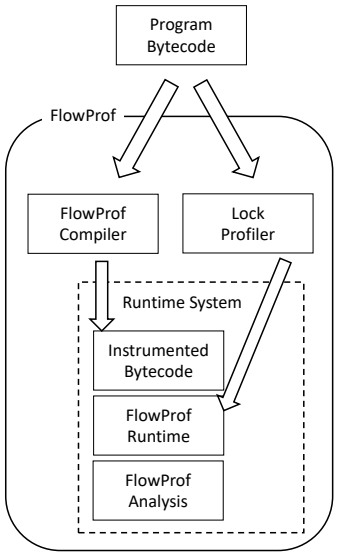

Figure 3. Architecture of FlowProf

4 FlowProf Overview

Figure 3 presents an overview of FlowProf. FlowProf's profiling can incur significant space and time overheads, and thus FlowProf targets its profiling at the class granularity towards code with significant contention. The uninstrumented program bytecode is fed to FlowProf's Lock profiler for an initial run that finds classes with significant lock contention. FlowProf's lock profiler identifies synchronization bottlenecks by monitoring the time spent waiting to acquire contended locks using Java Virtual Machine Tool Interface (JVMTI). The output of the lock profiler is a list of classes along with their total waiting time for synchronization.

The FlowProf compiler is implemented as a SOOT [25] compiler pass and instruments the bytecode of the application to be profiled. FlowProf's instrumentation supports running instrumented code only inside critical sections of contended class. The FlowProf runtime is fed with output of lock profiler which in turn enables profiling for the contended class. The FlowProf runtime efficiently records the

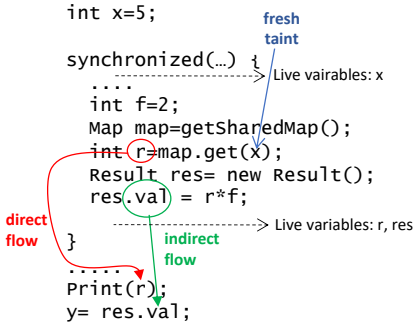

Figure 4. Information Flow Analysis

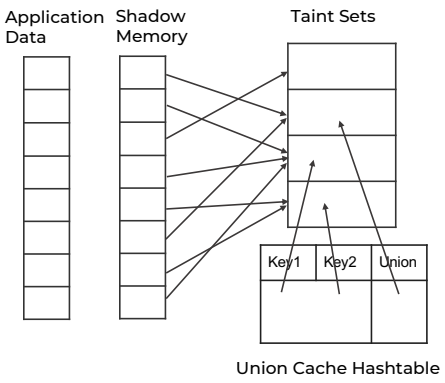

Figure 5. Shadow information used by FlowProf

execution of synchronized regions, records shared memory accesses, and tracks information flows.

Finally, FlowProf presents the extracted information with a conflict graph. FlowProf also performs schedule impact analysis to help developers understand the potential opportunity to eliminate bottlenecks.

5 Information-Flow Analysis

The goal of FlowProf's analysis is to determine what information is necessary to compute the outputs of a critical section. FlowProf uses information flow analysis to track this flow of information — FlowProf's approach requires that it can distinguish between inputs of different critical sections. Thus, FlowProf implements an information flow analysis that can track the flow of multiple distinct types of taints.

Figure 4 shows how FlowProf uses information flow analysis to monitor the flow of information in and out of a critical section, enabling FlowProf to track information flow between critical sections. FlowProf labels information flowing into a critical section with a freshly generated taint by tainting all variables that are live at the start of the critical section. In the example of Figure 4 only `x` gets fresh taint since it was live at the beginning of the critical section. Information can flow out of a critical section through either direct or indirect flows. Direct flows occur when a variable that is live out of a critical section is tainted. Indirect flows occur when a tainted field or array element is read outside of the critical section. FlowProf identifies variables that are live at the end of the CS, and records their set of taint types as direct flows. Tracking indirect flows out of the critical section through the heap is more challenging—FlowProf detects such flows when a tainted field is read from outside of the critical section. In Figure 4, variable `x` and reference `res` is live at the end so their taint sets are recorded to determine direct flows. However, reference `res` itself would not be tainted in this example but its pointed object's field `val` would be tainted. The taint of `val` would create indirect flow when it is read outside of the critical section.

5.1 Taint Sets & Shadow State

A value in a data structure may have been computed from the inputs of several different dynamically executed critical sections (CS). Thus, we must shadow each value in the computation with a set of taints. Since each CS execution gets a unique new taint, the total number of taints can be large. In our experiments, the number can be as large as several hundred thousand. State of the art techniques [8, 12] for handling multiple taints assign a bit vector of size equal to the total number of taints for each taint set where each bit indicates whether the particular taint is present in the set. Though such an approach is easy to implement and fast for merge operations, it is only feasible if the number of taints is at most in the range of hundreds. Since, in our case the number is way higher we have implemented taint tracking differently. We represent taint as an integer and a taint set with an array of integers. Consequently, we have implemented a number of optimizations in our approach to make our implementation of taint tracking efficient.

FlowProf represents each taint by a positive integer that indexes into a taint vector, which stores metadata associated with the corresponding critical section. FlowProf shadows each local variable, instance or static field with memory space to hold an integer that specifies the particular taint set. Shadow memory for arrays are integer arrays which are allocated on demand. Taint sets with two or more taints are represented as integer indices into a taint set vector. Each taint set record in the taint set vector contains the integers for the individual taints all stored in increasing order to support fast merge operations. We implement an optimization for singleton taint sets—singleton taint sets are represented as negative integers where the singleton taint set's integer is the negation of the integer for its single taint. The empty taint set is represented using the integer 0. Figure 5 presents FlowProf's implementation of taint sets & shadow state.

The most common operation in FlowProf's information flow analysis is merging taint sets. Naively merging taint

sets incurs overhead—it requires looking up two taint sets in the taint set vector, merging the underlying sets, and then looking up the merged taint set in the taint set to index map. FlowProf optimizes the merge operation using a fixed-size union cache hashtable that caches the result of merging two taint set indexes.

Array Shadows: FlowProf's runtime maintains a map from array objects to their shadow array objects. Each shadow array has the same length as the original array, and its elements store the integer of the taint set of the corresponding element in the original array. To speed up the process of looking up array shadows, FlowProf caches the most recently used mappings in a thread local cache.

Java represents multi-dimensional arrays as a tree of single-dimensional array objects. For a multi-dimensional array, FlowProf maintains a shadow array for each of its component one-dimensional arrays. Shadow arrays are created on demand to save space and time—arrays that are never written to inside of a critical section will not have shadow arrays. The absence of a shadow array implies that the taint sets for the array's elements are all empty.

5.2 Taint Propagation

We next briefly describe how FlowProf propagates taint for each operation that can appear in the intermediate representation of a method's control flow graph. Table 1 presents the taint propagation rules for samples of different operations. The notation $\mathcal{T}[\![x]\!]$ represents the taint set for x, where x can be a local variable, array element, instance field, or static field.

In general for assignments, the right hand side (RHS) will propagate its taints to the left hand side (LHS). If the RHS is a binary operation, then FlowProf computes the taint set of the LHS as the union of taint sets of the two operands. If it is unary or cast operation, then FlowProf just propagates the RHS taint set to the LHS. If it is an `instanceof` expression, then FlowProf propagates the taint set of the operand to the LHS. If it is a `new` or `new array` expression, then FlowProf assigns the LHS to the empty taint set.

Field and Array Accesses: Reads from static fields propagate the taint set of the static field's shadow to the LHS. Writes to static fields store the taint set of the RHS to the static field's shadow.

To efficiently taint the inputs of critical sections, FlowProf only taints the variables that are live into the critical section and read by the critical section. Thus, we need reads from instance fields to further propagate the taint set for the reference. Therefore, FlowProf computes the taint set of a read from an instance field as the union of the taint set of the object reference and the taint set from the field's shadow. Writes to instance fields only propagate the taint set of RHS to the destination field's shadow. Similarly, we make reading an array element propagate the union of the taint sets of the array reference and the array element. When writing to an array element, we propagate only the taint set of RHS to the destination field's shadow.

Table 1. Taint propagation rules for various operations

Object ob = new Object() $\mathcal{T}[\![ob]\!] = \{\}$	[new expression]
c = a+b $\mathcal{T}[\![c]\!] = \mathcal{T}[\![a]\!] \cup \mathcal{T}[\![b]\!]$	[binary operation]
d = (D)c $\mathcal{T}[\![d]\!] = \mathcal{T}[\![c]\!]$	[cast expression]
c = -b $\mathcal{T}[\![c]\!] = \mathcal{T}[\![b]\!]$	[unary expression]
int ar[] = new int[5] $\mathcal{T}[\![ar]\!] = \{\}$	[new Array expression]
bool b = c instanceof D $\mathcal{T}[\![b]\!] = \mathcal{T}[\![c]\!]$	[instanceof expression]
x = A.a $\mathcal{T}[\![x]\!] = \mathcal{T}[\![A.a]\!]$	[Static field read]
A.a = x $\mathcal{T}[\![A.a]\!] = \mathcal{T}[\![x]\!]$	[Static field write]
x = ob.a $\mathcal{T}[\![x]\!] = \mathcal{T}[\![ob]\!] \cup \mathcal{T}[\![ob.a]\!]$	[instance field read]
ob.a = x $\mathcal{T}[\![ob.a]\!] = \mathcal{T}[\![x]\!]$	[instance field write]
x = ar[i] $\mathcal{T}[\![x]\!] = \mathcal{T}[\![ar]\!] \cup \mathcal{T}[\![ar[i]]\!]$	[array read]
ar[i] = x $\mathcal{T}[\![ar[i]]\!] = \mathcal{T}[\![x]\!]$	[array write]

5.3 Instrumentation Strategy

FlowProf starts tracking information flow when the execution enters the outermost critical section and then stops tracking information flow when the execution exits that critical section. To facilitate executing uninstrumented code outside of the critical section FlowProf has two versions of every method. The first version is not instrumented and is called when FlowProf is not tracking information flow. The second version is instrumented to track information flow and is called from contexts in which FlowProf is tracking information flow.

FlowProf also facilitates executing instrumented code for contended classes only. Therefore, it inserts a check before the start of every critical section in the uninstrumented version of the method to determine whether the corresponding class has profiling turned on. In that case the execution switches to the instrumented code.

5.4 Execution Traces

For each critical section executed with instrumentation FlowProf records the lock it acquires, all the shared memory accesses and information flows into the critical section from other critical sections. FlowProf also records timestamps to measure the execution time of the critical section and waiting time to acquire the lock.

5.5 Sampling

Recording a trace for every critical section in a long running execution can require too much space. FlowProf uses a sampling strategy to limit the size of the trace that it must record. Randomly selecting individual critical sections is problematic

as FlowProf will likely miss dependencies between overlapping data structure operations. Such dependencies are the most important as they determine whether the operations can run in parallel. Instead FlowProf's sampling strategy selects windows in time. During these windows, the entire trace is recorded and outside of these windows FlowProf does not record a trace. This sampling strategy is useful because it can determine how much additional parallelism can be achieved in randomly selected windows of the execution. FlowProf's skipping strategy monitors memory usage to adaptively adjust the duration of the window during which profiling is disabled.

FlowProf disables profiling by continuing to run instrumented code, but assigns the critical section to the special taint 0. This disables recording time stamps and field and array accesses.

5.6 Indirect Taint Flows

FlowProf tracks direct information flows out of a critical section by taking the union of the taints of all live variables out of a critical section. But tainted values can also be passed out through object or class fields and array elements where the reference to the object or array is itself not tainted. To address the issue, we must instrument field and array element accesses outside of instrumented critical sections to check if the value is tainted. If a tainted value is read FlowProf records the time and taint.

Thus, FlowProf must also add minor instrumentation to code outside of instrumented critical sections to track information flows that escape a critical section via fields or array elements. FlowProf instruments all field or array reads to check if the field or element has a taint.

6 Analyzing Profile Data

From the execution trace FlowProf generates a static conflict graph that presents lock conflicts, memory access conflicts and information flows between executed critical sections along with their frequencies. Though the conflict graph helps to find problematic critical sections and provides a preliminary understanding of the bottleneck, FlowProf bases its analysis on schedule impact analysis.

6.1 Schedule Impact Analysis

Schedule impact analysis helps the developer understand how much different types of unnecessary conflicts contributes to the bottleneck. The schedule impact analysis allows the developer to explore what if scenarios. It simulates the recorded execution and estimates the waiting time in the bottleneck for an optimized implementation that does not require waiting for the dependencies that the developer plans to optimize away. Specifically, the developer can require the schedule impact analysis to report the time taken for the following scenarios: (1) the existing implementation in which we wait for all existing lock conflicts, (2) an optimized implementation that removes unnecessary lock conflicts such that we only have to wait when there are memory access conflicts, and (3) an optimized implementation that removes unnecessary memory access conflicts such that we only have to wait when there are information flow dependencies.

The schedule impact analysis takes as input one of the three conflict/dependency relations between critical section instances. The conflict/dependency relation maps a critical section to all of the prior critical section invocations on which it conflicts/depends, e.g., in the case of lock conflicts the critical sections that acquire the same lock. The analysis replays the recorded execution trace. At each lock acquire event, it computes a new time the event would have started if it only had to wait for the lock releases of the critical sections from the conflict/dependency relation that executed earlier in the original execution.

The schedule impact analysis enables the developer to match the structure of conflicts and information flow dependencies of a bottleneck to the patterns described in Section 3. A significant reduction in the percentage of total waiting time in the simulation that waits for only memory access conflicts stipulates a match to pattern 1. Similarly from memory access conflict simulation a significant reduction in the percentage of total waiting time in the simulation that waits for only information flow dependencies stipulates a match to pattern 2.

7 Evaluation

We evaluated FlowProf with the goal to understand how effectively it helps with understanding bottlenecks in applications and its efficiency for analyzing programs. We ran our experiments on an 3.6 GHz Intel(R) Core(TM) i9-10850K CPU with 10 physical cores, 2 hardware threads per core, and 128 GB of memory.

7.1 Benchmarks

We have searched open source Java projects and assembled a collection of benchmark applications to test FlowProf. We used the following criteria to include applications to our benchmark suite:

1. We selected benchmarks that have significant lock contention. In our case, we only considered those applications that spend at least 10% of their total thread execution time waiting on synchronization.
2. We eliminated benchmarks that were not compatible with the SOOT[25] compiler framework. The baseline SOOT compiler does not successfully generate working bytecode for programs that utilize some of the more advanced Java language features.
3. We only selected benchmarks that primarily use synchronized methods or synchronized blocks for concurrency control.

To find applications with significant lock contention we used FlowProf's lock profiler on other open source benchmark suites, projects from 50K-C repository [27] and on open source projects from GitHub. Most projects that have available test cases with sample inputs do not show significant lock contention. Next, the projects showing significant contention got filtered by the other 2 criteria. We found 3 compatible benchmarks, H2, Tradesoap, and Tradebeans, from the DaCapo benchmark suite [5]. Since all these 3 benchmarks show contentions in the same critical sections of the H2 database that all of them use, their analysis is summarized as H2 benchmark listed in our suite. We also tested FlowProf on the Renaissance suite [32] but only found one benchmark showing significant lock contention. Other benchmarks were collected from 50K-C project repository [27] and GitHub. The following applications are in our benchmark collection:

- **BigMap:** BigMap is a project from the 50K-C repository. It is a big, fast, persistent map based on memory mapped files.
- **H2:** H2 is a benchmark from the DaCapo suite. H2 is an SQL relational in-memory database engine written in Java.
- **UberZip:** UberZip [28] is a fast multi-threaded command-line application to extract zip files.
- **DB-shootout:** DB-shootout is a benchmark from the Renaissance suite. It executes a shootout test on an in-memory MVStore database.
- **Finmath:** The Finmath library [7] provides implementations of algorithms related to mathematical finance.
- **ConsistentHash:** ConsistentHash is a project from the 50K-C repository.

7.2 Analysis Results

We have run FlowProf on all the benchmarks analyzing the **most contended** classes reported by FlowProf's lock profiler. Table 2 summarizes the results of the schedule impact analysis for the benchmarks. As discussed in Section 6.1 schedule impact analysis reports waiting time for each of the 3 simulation considering either all lock conflicts, only memory access conflicts or only information flows. In Table 2 the total waiting time for simulation of lock conflicts is reported in absolute value computed by schedule impact analysis. For simulation of memory access conflicts and information flow the ratio of wait time with respect to waiting time of lock conflict is reported. As discussed in Section 3, pattern 1 occurs when there is a significant reduction in the percentage of waiting time in memory access conflict simulation. Pattern 2 occurs when the percentage of waiting time reduces significantly in information flow simulation from that of memory access conflict simulation.

In some applications, eliminating one bottleneck can reveal another new bottleneck. In such a case, we did another analysis cycle to understand the newly revealed bottleneck.

Table 2. Comparison of wait times for simulations of different conflicts in schedule impact analysis and their matched pattern. Lock simulation wait time is in the unit of 10^6 CPU cycles.

Benchmark	Lock simulation wait time	Wait time ratio with lock simulation		Matched pattern	
		Memory access	Information flow	1: Over-synchronization	2: Nonfundamental access conflicts
BigMap	51,407	99%	2%	-	✓
H2	129,025	100%	23%	-	✓
UberZip	17,415	0%	0%	✓	-
DB-shootout	444,800	100%	0%	-	✓
Finmath	13,469	0%	0%	✓	-
Finmath(v2)	34,422	0%	0%	✓	-
Consistent-Hash	1,550,119	24%	0%	✓	✓

This occurred in Finmath and we denoted the version after 1st bottleneck was eliminated as Finmath(v2). Table 3 shows that the performance gain from these optimizations ranges from 6% to 58%. To measure the performance gain we executed the benchmarks 10 times and used long runs when available to mitigate the effects of noise and other application startup overheads.

The optimization opportunities we found for H2 and DB-shootout have already been implemented in their later releases. We have reported the optimization opportunities in all other benchmarks to their respective developers. The developers of Finmath showed interest and merged a pull request to their repository with the optimization. The developer of UberZip expressed interest to incorporate the optimization if it is developed further in future. We did not hear back from the developers of BigMap and ConsistentHash.

We next briefly describe the results of our analysis, their optimization insights, and the optimizations we implemented for each of the benchmarks.

BigMap: The reported contention is in the `bigmap.page.MappedPageFactoryImpl` class where it uses a `java.util.HashMap` guarded by a lock to store values. FlowProf's schedule impact analysis as depicted in Table 2 shows similar wait times for the simulation of lock and memory access conflicts, indicating that there would not be any performance improvement by only changing the synchronization strategy. The high memory access conflicts relative to the lock conflict indicate that the current synchronization strategy is indeed necessary to protect conflicting memory accesses. But the analysis shows a significant 97% drop in wait time from the simulation of memory access conflicts to the simulation of information flow dependences. This is an example of pattern 2 and indicates that a different implementation of the synchronized region can improve the performance. We implemented

the optimization insight by replacing `java.util.HashMap` with `java.util.concurrent.ConcurrentHashMap`, which is a thread safe and a scalable redesign of `HashMap` where multiple threads can perform operations simultaneously. With this change the benchmark achieved a 55% performance improvement.

Table 3. Summary of optimizations implemented in the benchmarks according to the optimization insights and corresponding performance gains.

Benchmark	Optimization Pattern		Implemented Optimization	Performance gain	
	1	2			
BigMap		✓	Used scalable concurrent data-structure	55%	
H2		✓	Scalable redesign of the DB	22%	
UberZip	✓		Fine-grained locking	6%	
DB-shootout		✓	Scalable redesign of the MVStore	7%	
Finmath	✓		Fine-grained locking	17%	40%
Finmath (v2)	✓		Reader-writer lock	20%	
ConsistentHash	✓		Fine-grained locking	58%	

H2: The schedule impact analysis produced the same waiting time for lock and memory access conflicts, but showed a more than 70% reduction of waiting time for information flow dependences. The reduction of wait time for information flow dependencies suggests that changing the implementation of the database can improve performance. After examining the source code we found that H2 uses a single lock for the entire database. Each access causes H2 to modify a few common fields and as a result there is memory access conflict whenever it has lock conflict. Thus, it is necessary to redesign the database to allow different threads to access the database simultaneously. We have found there is a later version of the H2 database than the one used in DaCapo, and this version implements this optimization. We used that version to measure the performance benefit of the optimization insight.

UberZip: The initial run of the lock profiler indicates that the sequential bottleneck in this benchmark is in the `ZipFile.ZipFileInputStream` class from the `java.util.zip` package. The schedule impact analysis reports 100% reduction of waiting time for memory access conflicts relative to lock conflicts. This suggests that the current implementation is over-synchronized and eliminating over-synchronization is sufficient to parallelize the implementation. Java's `ZipFile` implementation uses a single lock for all `ZipFileInputStream` objects while reading different zip file entries from different input streams. To eliminate over-synchronization, we implemented fine-grained locking strategy where we assigned separate locks for separate `ZipFileInputStream` objects which essentially eliminates the contention.

DB-shootout: The contention happens in the in-memory MVStore database when all threads try to insert data into the MVStore. The results in Table 2 shows similar wait times for the simulation of lock and memory access conflicts and almost no waiting for the information flow simulation. That means current synchronization scheme is necessary for current implementation and a different implementation can eliminate the bottleneck. We checked the source code to find that the insert operation is guarded with a lock and thus all concurrent operations are serialized. The lock is necessary since it modifies common fields in every operation. A redesign of the MVStore is necessary to allow multiple threads to insert simultaneously. We have found that a later version of the MVStore implements the optimization and we replaced that version in the Renaissance suite to achieve the reported performance gain.

Finmath: The initial bottleneck is in the `curve.CurveInterpolation` class. When we ran our analysis on this class, the schedule impact analysis reported no waiting time for memory access conflicts. So it suggests that the current synchronization scheme is over-synchronized and optimizing the synchronization scheme can improve performance. The class used a single lock for all cloned objects of that class while working on separate sets of data. According to the optimization insight we assigned separate lock for each cloned objects which eliminates the bottleneck on that class.

Eliminating the first bottleneck revealed a new bottleneck on the `java.util.Vector` instance used in the `calibration.Solver` class. In our analysis of this version, Finmath(v2) also shows no wait time for memory access conflict simulation indicating that the bottleneck suffers from over-synchronization. The `java.util.Vector` class uses a single lock to synchronize all its operation. From the conflict graph we learned that the contention observed in the benchmark is in `get()` and `size()` operations for `Vector`. Since both operations only reads data from vector, we eliminate the over-synchronization by using a reader-write lock.

ConsistentHash: The benchmark shows a 76% simulation wait time reduction for the memory access conflict analysis relative to the lock conflict analysis and the rest of wait time is eliminated in information flow simulation. This indicates a strong match for pattern 1 where eliminating over-synchronization can reduce most of current bottleneck. The benchmark used a coarse lock to maintain counts of several of nodes in a map. To eliminate over-synchronization we used a fine-grained locking strategy where we assigned a separate lock for each node where threads increasing same node's count acquire the same lock but threads working on different node acquire different locks. This solution achieves a 58% performance improvement. As shown in Table 2 there is another 24% reduction in wait time in information flow

simulation which indicates that changing the current implementation can ideally show further improvement. To realize this optimization the counting on the same node needs to be parallelized as that is where the conflicting accesses are happening. This could potentially be parallelized by using a scalable shared counting implementation [18].

7.3 Efficiency

Table 4 reports the mean execution time over 5 runs for each benchmark. The benchmarks show a slowdown for instrumented runs between 2× to 31×. This compares well with the reported overheads of other dynamic performance profiling tools. SyncProf [39] reports a slowdown of 60× to 100×, JITProf [15] reported a 18× slowdown, and Toddler [31] reported a 15.9× slowdown. While most profilers improve efficiency by using sampling, FlowProf does not achieve the full benefit from sampling as it always runs the instrumented critical sections to propagate existing taints. FlowProf's primary performance optimization is that it only runs instrumented code for the critical sections that have shown significant contention.

Table 4. Different efficiency metrics for FlowProf observed on the benchmarks indicating standard deviation of measurements with ±.

Benchmark	Original runtime (s)	Slow down	Memory overhead	Analysis time (s)	Lock profiler overhead
BigMap	1.32 ± 0.12	4.34×	2.32×	0.78 ± 0.09	7.12%
H2	6.40 ± 0.01	18.53×	20.70 ×	28.63 ± 7.85	-0.16%
UberZip	1.43 ± 0.08	2.36×	1.48 ×	0.14 ± 0.01	-1.96%
DB-shootout	9.39 ± 0.37	30.91×	22.21 ×	6.25 ± 1.23	-1.36%
Finmath	0.37 ± 0.02	5.94×	4.95 ×	12.16 ± 1.77	-0.54%
Finmath(v2)	0.36 ± 0.03	6.47×	4.67 ×	0.34 ± 0.03	6.11%
ConsistentHash	4.17 ± 0.06	13.67×	13.05 ×	2.02 ± 0.33	21.44%

As discussed in Section 5.5, FlowProf 's memory overhead depends on sampling rate which is controlled in such a way that it does not exhaust available memory. Therefore, if it is run with more available memory it would have higher sampling rate resulting in higher memory overhead with higher precision. We ran our experiments on a machine with 128 GB of memory and the memory overhead observed in our experiments ranges from 1.5× to 22×. Running the experiments with higher/lower memory than this is supposed to impact the memory overhead accordingly.

The analysis time listed in Table 4 includes both static conflict graph generation time and schedule impact analysis time. The analysis time ranges from less than a second up to around half a minute. The JVMTI lock profiler which is run initially to find lock contention in benchmarks reports runtime overhead from -2% to 21%. The overhead is sometimes negative since the calls to the Java agent sometimes reduces contention on locks and/or the lock profiler sometimes influences the thread scheduling in a way that positively impacts the execution time.

7.4 Comparison with Existing Profiler

To the best of our knowledge, using information flow to profile concurrent application is novel and no previous profiler utilizes this approach. As a result no other existing profiler can find optimization opportunities in benchmarks that match pattern 2 in Table 2. In these cases the bottleneck is due to actual conflicting memory accesses and the existing synchronization is necessary to protect the memory accesses from data races. Other than that we only found one profiler, SyncProf [39], that suggests optimization insights for synchronization bottlenecks. SyncProf tracks memory accesses to suggest optimizations. Since SyncProf's implementation is not publicly available we could not test it on our benchmarks. However, SyncProf's approach to find optimization opportunities requires the application to use a static locking strategy where the same lock is assigned to protect a critical section in all of its executions. SyncProf cannot handle cases where a critical section acquires different locks depending on data it is going to access. This is serious limitation as this is the most common case in Java programs—for example, every non-static synchronized method acquires different locks depending on the receiver object it is invoked upon. The limitation of SyncProf stems from its approach where it tries to generate an optimization pattern using a static synchronization graph. FlowProf's schedule impact analysis preserves all dynamic conflict information and hence overcomes this limitation. Table 5 summarizes whether the technique of SyncProf could be used to discover the optimization opportunity in the respective benchmark.

Table 5. SyncProf's suitability in finding optimization opportunities in the benchmarks.

Benchmark	Suitable with SyncProf?	Reason
BigMap	No	Requires tracking information flow
H2	No	Requires tracking information flow
UberZip	No	Requires handling dynamic locking
DB-shootout	No	Requires tracking information flow
Finmath	No	Requires handling dynamic locking
ConsistentHash	No	Requires handling dynamic locking

7.5 Limitations

One limitation of this work is that it does not directly account for instrumentation distortion. FlowProf runs instrumented code inside critical sections of the most contended class and other parts are mostly uninstrumented. Therefore, the waiting time computed in schedule impact analysis would be higher because of the instrumentation overhead. However, the optimization insight does not depend on the absolute values of waiting times. The waiting times of simulation for three different types of conflicts are compared with each other to gain insights on optimization. Thus, the overall

impact of instrumentation is compensated in the final optimization insight the analysis produces. Another limitation is that this work tracks information flow through explicit taint propagation and does not track implicit flow where information flows through control dependency. Though it is possible to engineer cases where it is insufficient to only track explicit flow, we did not come across any such cases in the real world applications where FlowProf misses information flow because it does not handle implicit flow. In any case, existing approaches [8, 22] for handling implicit flow can be combined with FlowProf to make it more robust which we consider outside the scope of this paper.

Apart from that, the schedule impact analysis can be too conservative and can easily underestimate the potential benefits of optimizations. Schedule impact analysis computes execution time for an execution that respects the same dependency structure as the original execution. For example, if there is a queue of jobs, it will require that threads put jobs into the queue in the same order and that each thread waits to remove the same jobs as in the original execution. But in reality the optimized version that uses the queue of jobs can potentially run faster when threads add and remove jobs in different orders. However, FlowProf does not have enough information to simulate these alternate orders.

8 Related Work

Much work has been done on identifying parallelism bottlenecks. Some work has looked at parallelization bottlenecks in task parallel frameworks [33, 38]. Call graph profiling was introduced by gprof [16]. Sampling has been used to efficiently measure lock contention in multi-threaded programs [34]. Measurement overhead from performance profiling can distort results, but techniques have been developed to compensate for this distortion [26]. Techniques have been developed to support efficient flow and context sensitive profiling with hardware performance counters [2].

SyncProf is a profiling tool for concurrent programs [39]. It measures the time spent waiting to acquire a critical section or lock and monitors dependencies between critical sections. It then analyzes the critical section dependence graph to find opportunities to improve locking strategies, e.g., fine-grained locking or reader-writer locks. A major limitation of SyncProf is that it only handles static locking strategies where a critical section always acquires the same locks and does not consider data dependent locking strategies (such as associating a lock with each hash table) where a critical section acquires different locks depending on the data it accesses.

Kismet [21] and Kremlin [14] use critical path analysis [20] to estimate the parallelization potential of serial programs. These tools target helping developers parallelize serial programs while FlowProf targets optimizing already parallel programs. Data dependence profiling has been used to help parallelize sequential code [23]. These approaches use conflicting memory accesses to determine dependencies while FlowProf uses information flow analysis to determine which dependencies are fundamental to the computation. These approaches could potentially benefit from FlowProf—information flow analysis could potentially enable these tools to find more opportunities to parallelize code.

Causal profiling can simulate the performance benefits of optimizations to help developers identify the best optimization opportunities [9]. Causal profiling can measure the performance benefits of optimizing code blocks, but cannot measure the performance benefits of reducing contention by redesigning synchronization or data structure implementations.

HPCToolkit [36] supports profiling lock contention. This work explores ways of attributing the overhead of waiting on a lock—blaming the thread that waits on the lock or blaming the thread that holds the lock. This work does not examine whether the existing synchronization approach can be improved, but simply outputs how much execution time each code block is blamed for. Critical path analysis can determine which locks are important for performance [6]. Wait-for graphs can be used to reason about the potential performance impacts of reducing waiting time [40]. The FreeLunch profiler [10] measures the percentage of execution time spent waiting on locks. There has been work on lightweight call path tracing for HPC applications [35]. Tools built on Pin can identify lock contention and loops that can be parallelized [4]. Bottlegraphs help developers visualize performance in multi-threaded programs [11].

Much work has been done on taint analysis [8]. It has been widely used to analyze security properties [3, 24, 30, 37]. Selective propagation of taint along control flow edges also known as implicit flow has been implemented in the context of security [22].

9 Conclusion

This paper presents a new profiling approach that leverage information flow to reason about whether sequential bottlenecks in multi-threaded programs are fundamental to the computation or merely artifacts of the implementations. We implemented this approach in FlowProf. We evaluated FlowProf on a set of multi-threaded benchmarks with sequential bottlenecks.

Acknowledgments

We thank the anonymous reviewers for their thorough and insightful comments. We would also like to thank Xiafa Wu and Conan Truong for their contributions in implementing discrete event simulation. This work is supported by the National Science Foundation grants CNS-1703598, OAC-1740210, CCF-2006948, CCF-2220410, and CCF-2102940.

Availability. FlowProf artifact that includes source codes, the studied benchmarks and instructions are available on Zenodo [29].

References

[1] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. Association for Computing Machinery, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[2] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/258915.258924

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 259–269. https://doi.org/10.1145/2594291.2594299

[4] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. 2010. Analyzing Parallel Programs with Pin. *Computer* 43, 3 (March 2010), 34–41. http://dl.acm.org/citation.cfm?id=1749398.1749444

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN conference on Object-oriented Programming Systems, Languages, and Applications*. 169–190.

[6] Guancheng Chen and Per Stenstrom. 2012. Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 71, 11 pages.

[7] Christian Fries. 2020. Mathematical Finance Library: Algorithms and methodologies related to mathematical finance. https://github.com/finmath/finmath-lib [Online; accessed 7-June-2023].

[8] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. 196–206. https://doi.org/10.1145/1273463.1273490

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

[10] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 291–307. https://doi.org/10.1145/2660193.2660210

[11] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. 355–372. https://doi.org/10.1145/2509136.2509529

[12] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. 32, 2, Article 5 (jun 2014), 29 pages. https://doi.org/10.1145/2619091

[13] Stijn Eyerman and Lieven Eeckhout. 2010. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 362–370. https://doi.org/10.1145/1815961.1816011

[14] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and Rebooting Gprof for the Multicore Age. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 12 pages. https://doi.org/10.1145/1993498.1993553

[15] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly javascript code. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 357–368.

[16] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. 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

[17] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A Scalable Lock-free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 206–215. http://doi.acm.org/10.1145/1007912.1007944

[18] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. 1995. Scalable Concurrent Counting. 13, 4 (nov 1995), 343–364. https://doi.org/10.1145/210223.210225

[19] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Jeffrey K Hollingsworth. 1998. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems* 9, 10 (1998), 1029–1040.

[21] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. 2011. Kismet: Parallel Speedup Estimates for Serial Programs. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 18 pages. https://doi.org/10.1145/2048066.2048108

[22] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the NDSS Conference*.

[23] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. 535–546. https://doi.org/10.1109/MICRO.2010.49

[24] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. https://doi.org/10.1145/2614628.2614633

[25] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15.

[26] Allen D. Malony and Sameer S. Shende. 2004. Overhead Compensation in Performance Profiling. In *Proceedings of the 2004 European Conference on Parallel Processing*.

[27] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: A Dataset of Compilable, and Compiled, Java Projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 1–5.

[28] Matt Hicks. 2008. Multithreaded Unzip? https://matthicks.com/2008/01/14/multithreaded-unzip/ [Online; accessed 7-June-2023].

[29] Ahamed Al Nahian and Brian Demsky. 2024. Artifact of CC 2024 Paper 'FlowProf: Profiling Multi-threaded Programs using Information-Flow'. https://doi.org/10.5281/zenodo.10464417

[30] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software.. In *Proceedings of the Network and Distributed System Security Symposium*.

[31] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 562–571. http://dl.acm.org/citation.cfm?id=2486788.2486862

[32] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[33] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2018. Analyzing and Optimizing Task Granularity on the JVM. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 27–37. https://doi.org/10.1145/3168828

[34] Andreas Schörgenhumer, Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2017. Efficient Sampling-based Lock Contention Profiling for Java. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 331–334. https://doi.org/10.1145/3030207.3030234

[35] Nathan R. Tallent, John Mellor-Crummey, Michael Franco, Reed Landrum, and Laksono Adhianto. 2011. Scalable Fine-grained Call Path Tracing. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1995896.1995908

[36] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. 269–280. https://doi.org/10.1145/1693453.1693489

[37] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. 87–97. https://doi.org/10.1145/1542476.1542486

[38] Adarsh Yoga and Santosh Nagarakatte. 2017. A Fast Causal Profiler for Task Parallel Programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3106237.3106254

[39] Tingting Yu and Michael Pradel. 2016. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 389–400. https://doi.org/10.1145/2931037.2931070

[40] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 527–543. https://www.usenix.org/conference/osdi18/presentation/zhou

Received 10-NOV-2023; accepted 2023-12-23