Profile-based Detection of Layered Bottlenecks

Tatsushi Inagaki Yohei Ueda Takuya Nakaike Moriyoshi Ohara
IBM Research
Tokyo, Japan
{e29253,yohei,nakaike,ohara}@jp.ibm.com

ABSTRACT

Detection of software bottlenecks which hinder utilizing hardware resources is a classic but complex problem due to the layered structures of the software bottlenecks. However, model-based approaches require a performance model given, which is impractical to maintain under today's agile development environment, and profile-based approaches do not handle the layered structures of the software bottlenecks.

This paper proposes a novel approach of taking the best of both worlds which extracts a performance model from execution profiles of the target application to detect the layered bottlenecks. We collect a *wake-up profile* of threads, which samples an event that one thread wakes up another thread, and build a *thread dependency graph* to detect the layered bottlenecks.

We implement our approach of profile-based detection of layered bottlenecks in the Go programming language. We demonstrate that our method can detect software bottlenecks limiting scalability and throughput of state-of-the-art middleware such as a web application server and a permissioned blockchain network, with small amount of the runtime overhead for profile collection.

KEYWORDS

layered bottlenecks, wake-up profile, thread dependency graph

ACM Reference Format:
Tatsushi Inagaki Yohei Ueda Takuya Nakaike Moriyoshi Ohara. 2019. Profile-based Detection of Layered Bottlenecks. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3297663.3310296

1 INTRODUCTION

Software resources in a computer system, such as threads, mutual exclusion locks, and communication channels, can be software bottlenecks [25] of the system. That is, capacities of the software resources diminish the maximum performance of the system by preventing full utilization of hardware resources, often unexpectedly due to various reasons about design and configuration of software resources. For example, scalability of the Acme Air Go web application benchmark [45] in Figure 1 implies that some software bottlenecks exist in the system configured as shown in Figure 2.

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310296

The achieved throughput does not scale proportionally to the target throughput when the target throughput is 8000 transactions per second or larger, but none of the host processors are saturated. While not shown for simplicity, the percentage of the time to wait for input and output (I/O) operations is also at most 0.1% on any host.

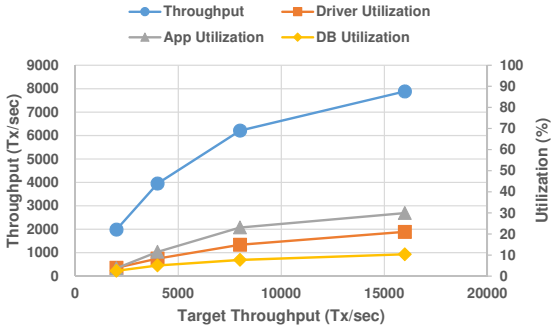

Figure 1: Throughput and processor utilization of the Acme Air Go web application benchmark.

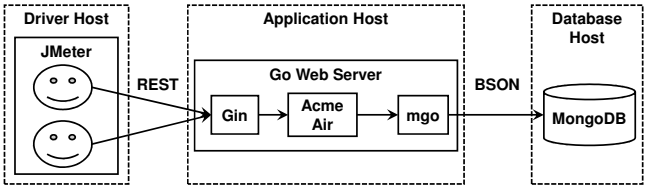

Figure 2: A configuration of the Acme Air Go benchmark.

Software bottlenecks are also called layered bottlenecks [42] since a service request to the system can simultaneously hold software resources from multiple layers of underlying services, in contrast to that a request holds hardware resources one by one. While the existence of layered bottlenecks itself has been understood from many decades ago [20], detecting and thus modeling them in a given computer system is not trivial at all. As far as we know, the layered queueing network (LQN) [11, 42] is one of the most simple but still flexible performance model which can be used to analyze bottlenecks in large-scale software systems such as a web application server [32].

One practical issue of the LQN and other model-based approaches is how to maintain performance models. Usually a performance analyst defines the hierarchical structure of the target system and specifies service demand manually, which requires the domain knowledge and complex performance measurement steps for each

component in the model [12, 46]. Today, this is more challenging than before since agile software development environments build software systems on top of a number of third parties' components. This is a dilemma, because a performance problem often occurs when we do *not* know the performance model.

Assuming that the performance of an existing target system can be modeled by an LQN, how can we *measure* the model and its layered bottlenecks from execution profiles of the target system? This paper answers that we can build a model which allows us to detect layered bottlenecks, by profiling threads and their *dependency* in the target system. More specifically, we can build a *thread dependency graph* to capture dependency among software resources and their average queue lengths in the measured execution, as described in Section 2. The graph is built from existing thread profiles and novel *wake-up profiles*, which extend existing profiles of blocked threads, as implemented in Section 3.

Note that our target is to *detect* layered bottlenecks in an existing software system, but is not to estimate benefits of eliminating the bottlenecks, or to suggest possible optimizations. The former requires modeling potential bottlenecks, which might be hidden in the current configuration. Instead, we can build a thread dependency graph only from execution profiles of an existing software system, in contrast to the traditional LQNs. The latter requires domain knowledge of the target application and generally is not provided by other profile-based approaches [1, 7, 9, 28, 48, 49].

Nevertheless, our approach can detect layered bottlenecks in complex software systems such as a web application server and a permissioned blockchain, as shown in Section 4. For example, a thread dependency graph of the Acme Air Go benchmark with the baseline configuration is shown in Figure 12. The graph represents threads, that is, software resources, as nodes labeled with the functions and average numbers of threads, and their dependency as directed links[1]. From the threads which handle incoming transaction requests, by hierarchically tracking dependencies with the largest average number of threads, we see most of the requests are blocked at allocating a new database connection. This observation leads to optimization to cache live database connections among transactions. Figure 14 shows a new thread dependency graph after the optimization. Now most of the requests are waiting for database driver threads, whose majority is waiting for I/O operations for the database. The optimization achieved performance improvement of the throughput up to +30%, as shown in Figure 13.

Compared to the related work discussed in Section 5, our contributions in this paper are as follows:

- We proposed a new method to detect layered bottlenecks which only relies on execution profiles of the target application, but does not rely on a manually built performance model. Our approach has a practical benefit that we can analyze modern applications built on top of various third parties' components from open source code repositories.
- We implemented the proposed method in the Go programming language [37], which provides highly scalable user-level multithreading. Our implementation is based on an extension to the built-in sampling profiler of the Go programming language, whose runtime overhead can be arbitrarily minimized by reducing the sampling frequency.
- We demonstrated that the proposed method can detect layered bottlenecks in complex and state-of-the-art middleware such as a web application server and a permissioned blockchain. Mitigating the detected bottlenecks improved performance and scalability of the target applications.

One of our conclusions in Section 6 is that a modern server application running on a distributed platform is a complex LQN built on modular components and systems. Our method will be useful to detect and optimize bottlenecks of such applications in environments of agile and continuous software development.

2 THREAD DEPENDENCY GRAPH

Assuming that a target application and its execution profile are available, we characterize performance of the target application as a *thread dependency graph*. A thread dependency graph is a directed graph which represents threads and their dependency in given execution profiles of the target application.

This section describes what is a thread dependency graph and how to detect layered bottlenecks from the graph. How to build a thread dependency graph is described in Section 3.

2.1 Node

A node of a thread dependency graph represents a set of threads which handle a class of requests. A node has an average *number* of threads in a given execution profile, and the *status* of the threads, which may be runnable or blocked.

Invoking Thread Node. An invoking thread node represents a set of threads which are:

- created by the same statement,
- executing the same root function, and
- blocked to wait for services from other nodes by calling functions.

In our graphical notation in Figure 3, an invoking thread node is represented as a square box labeled with the root function and the average number of threads.

Note that ordinary function call by one thread can be also considered as two different threads executing the caller and the callee functions, the caller thread creates the callee thread, and the caller thread is blocked until the callee thread completes. Thus, a thread dependency graph can uniformly represent both of a traditional call graph and dependency among threads.

Performance of a set of threads executing the same root function is suitable to be modeled as a flat queueing network with a single customer class, because each thread has a unique instruction pointer and uses the underlying services one by one, and each request is likely to have the same service demand.

Blocked Thread Node. A blocked thread node represents a set of threads blocked due to synchronizing operation of the underlying programming language, such as acquiring a mutual exclusion lock, sending or receiving a message via a communication channel, an I/O operation, sleeping at a timer, making an operating system call, and so on. For example, Table 1 shows the blocking status of threads

[1]What we call a "thread" here about this example is actually a *goroutine*, which is a user level thread managed by the Go language runtime library on top of the operating system level thread. For simplicity, we call goroutines as threads in this study, since they are equivalent from the perspective for performance models.

Table 1: Blocking status of threads in the Go language.

Status	Description
semacquire	Acquiring a built-in mutual exclusion lock
chan receive	Receiving a message from a channel
chan send	Sending a message into a channel
select	Waiting at a `select` statement
IO wait	Waiting for a network I/O operation
sleep	Sleeping
syscall	Requesting an operating system call

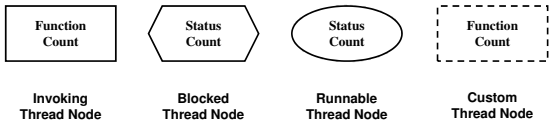

Figure 3: Nodes of a thread dependency graph.

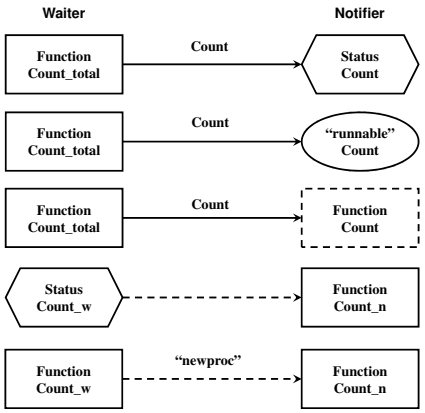

Figure 4: Links of a thread dependency graph.

in the Go language. In our graphical notation in Figure 3, a blocked thread node is represented as a hexagon labeled with the status and the average number of the threads.

Runnable Thread Node. A runnable thread node represents a set of threads which are runnable and can use underlying hardware resources. In a thread dependency graph, only processors are the hardware resources, since other hardware resources such as hard disks or network devices are used via the operating system and they are represented as blocked nodes. In our graphical notation in Figure 3, a runnable thread node is represented as an oval labeled with the status (i.e. `runnable`) and the average number of the threads.

Custom Thread Node. A custom thread node is an invoking thread node created by a user's annotation which specifies the root function and to refine an original invoking node to serve multiple customer classes into a set of underling invoking thread nodes with a single customer class for each. For example, the top level request handler is typically built on a generic message handling library for the underlying protocol, such as Hypertext Transfer Protocol (HTTP) [17], gRPC Remote Procedure Calls [38], and so on, which listens to incoming requests and creates or assigns a worker thread to handle the requests. When the target application serves multiple types of transactions, we can refine a thread dependency graph by separating custom thread nodes to handle each transaction from the invoking node to handle the generic incoming requests. Such kind of custom nodes can be specified base on the application programming interface (API) of the target application for the corresponding protocol. In our graphical notation in Figure 3, a custom thread node is represented as a dashed square box.

2.2 Link

A directed link of a thread dependency graph represents dependency among threads. The threads corresponding to the source node, that is, "waiter", threads wait for services from the threads corresponding the destination node, that is, "notifier" threads.

Synchronous Dependency Link. A synchronous dependency link represents dependency due to function calls, as described above in Section 2.1. The source node is either an invoking thread node or a custom thread node. The destination node is either a blocked thread node, runnable thread node, or custom thread node. A synchronous link is labeled with the average number of threads contributing the function call. Thus, the number of threads of the waiter is decomposed as a sum of the number of threads on the outgoing synchronous links. In our graphical notation in Figure 4, a synchronous dependency link is represented as a solid arrow from the waiter node to the notifier node.

Asynchronous Dependency Link. An asynchronous dependency link represents dependency due to synchronization among threads. That is, the waiter threads corresponding to the source node wait for the notifier threads corresponding to the destination node, via a mutual exclusion lock, communication channel, and so on. The source node is a blocked thread node and the destination node is either a thread node or custom thread node. In our graphical notation in Figure 4, an asynchronous dependency link is represented as a dashed arrow from the waiter node to the notifier node.

Creation of a thread is also represented as an asynchronous dependency link. The source node is the thread node for the child thread and the destination node is the thread node for the parent thread. In this case, the corresponding asynchronous dependency link is labeled as `newproc`.

2.3 Shorthand Notation

We introduce the following two shorthand notations to improve the readability of a thread dependency graph by omitting trivial nodes and links.

Figure 5 shows a shorthand notation when a single blocked thread node depends on a single invoking thread node or a custom thread node. In this case, we omit the blocked thread node and connect from the parent invoking thread node to the notifier node, via an asynchronous dependency link labeled with the average number of threads of the omitted blocked thread node.

Figure 6 shows another shorthand notation when an invoking thread node and a blocked thread node have cyclic dependency. This dependency typically occurs with a mutual exclusion lock, when one thread of the invoking thread node is blocked at the blocked

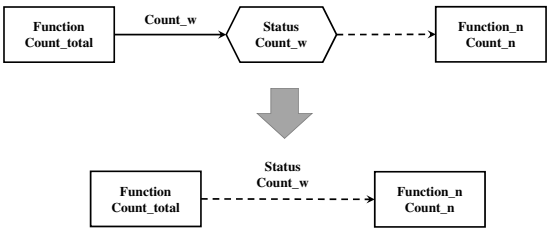

Figure 5: A shorthand notation of dependency from a single waiter to a single notifier.

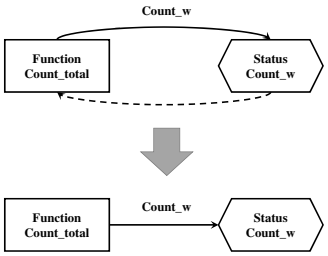

Figure 6: A shorthand notation of cyclic dependency.

thread node at acquiring the lock, and another thread of the same invoking thread node releases the lock. In this case, we simply omit the corresponding asynchronous dependency link, since existence of the backward link can be inferred from the semantics of the mutual exclusion lock.

2.4 Example

Let us think of a sample Go program shown in Figure 7. The main procedure spawns one sender thread at Line 6, spawns ten receiver threads at Line 8, and sleeps 5 minutes at Line 10. The sender thread repeats sleeping 1 milliseconds at Line 15, and sending a message into the channel c at Line 16. The receiver threads repeat receiving a message from the channel c at Line 22, and sleeping 1 nanoseconds at Line 23.

Figure 8 shows a corresponding thread dependency graph built from execution profiles of the program in Figure 7. The graph captures the performance characteristics of the sample program such that:

- there exist ten threads executing the function `receiver`,
- there exists one thread to execute the function `sender`,
- the receiver threads are waiting for the sender thread in 99% of their execution time, and
- the sender thread is almost always sleeping.

2.5 Bottleneck Detection

Given a thread dependency graph, we detect the layered bottlenecks by hierarchically traversing the path having the largest number of threads, starting from the custom thread node which is annotated to handle the target transaction. More specifically, for a given node in a thread dependency graph, the *bottleneck nodes* of the node are:

```
1  package main
2  import "time"
3  var c chan bool = make(chan bool)
4
5  func main() {
6      go sender()
7      for i := 0; i < 10; i++ {
8          go receiver()
9      }
10     time.Sleep(time.Duration(5) * time.Minute)
11 }
12
13 func sender() {
14     for true {
15         time.Sleep(time.Duration(1) * time.Millisecond)
16         c <- true
17     }
18 }
19
20 func receiver() {
21     for true {
22         _ = <-c
23         time.Sleep(time.Duration(1) * time.Nanosecond)
24     }
25 }
```

Figure 7: A sample Go program.

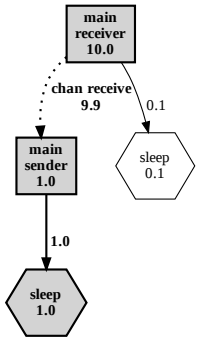

Figure 8: A thread dependency graph.

- among destination nodes connected via synchronous dependency links, the destination node with the largest number of the threads of the source node, and
- all destination nodes connected via asynchronous dependency links.

The rationale why we use the number of threads, that is, the queue length, instead of the utilization to detect bottlenecks is that the queue length is directly measurable in execution profiles as the number of threads, but the utilization is not, because the capacity of each software resource is generally not available in the profiles.

However, this simple algorithm practically works well, as we will see in Section 4, since when we increase the arrival rate of the target transaction, the queue length of the bottleneck resource will asymptotically diverge.

3 PROFILING

This section describes how to build a thread dependency graph from execution profiles. While we explain our implementation in the Go programming language, the discussions below are also applicable

```
goroutine 19 [sleep]:
time.Sleep(0xf4240)
        /opt/go/src/runtime/time.go:65 +0x130
main.sender()
        /main.go:15 +0x20
created by main.main
        /main.go:6 +0x51
goroutine 20 [chan receive]:
main.receiver()
        /main.go:22 +0x20
created by main.main
        /main.go:8 +0x72
```

Figure 9: A thread profile.

```
1175794700 503 @ 0x405bdc 0x405a18 0x405383 0x6e2bc6
# Waiter
#    runtime.gopark+0x12b          /opt/go/src/runtime/proc.go:287
#    runtime.goparkunlock+0x5d     /opt/go/src/runtime/proc.go:293
#    runtime.chanrecv+0x303        /opt/go/src/runtime/chan.go:506
#    runtime.chanrecv1+0x2a        /opt/go/src/runtime/chan.go:388
#    main.receiver+0x1f            /main.go:22
# created by
#    main.main+0x71                /main.go:8
# Notifier
#    runtime.send+0x8b             /opt/go/src/runtime/chan.go:280
#    runtime.chansend+0x687        /opt/go/src/runtime/chan.go:179
#    runtime.chansend1+0x42        /opt/go/src/runtime/chan.go:113
#    main.sender+0x1f              /main.go:16
# created by
#    main.main+0x50                /main.go:6
```

Figure 10: A wake-up profile.

to other programming languages and platforms such as the user level threads and the kernel threads.

3.1 Thread Profile

We sample a *thread profile*, a snapshot of stack traces of the all threads in the target application, including both of runnable and blocked threads, to capture the status and the number of threads. They are used to create thread nodes and the average number of threads. The sampling is done periodically based on a timer, so that the number of sampled threads becomes proportional to the cumulative time spent by the threads.

In the Go programming language, a thread profile is provided as goroutine profile by the pprof profiling library [37]. Figure 9 shows an example of a thread profile of the sample program in Figure 7. A thread profile takes a snapshot of the status, stack trace, and creation site of all threads.

For the user level operating system threads, which are typically used to implement threads in other programming languages, a system call such as ptrace [13] can be used to capture a snapshot of all threads in the target process. For the kernel threads, operating systems such as Linux [40] provide a pseudo file system to capture a snapshot of stack traces of a kernel thread [5].

3.2 Wake-up Profile

To profile asynchronous dependency among threads, we need to profile which thread a blocked thread is waiting for. This is captured by our novel *wake-up* profile. The wake-up profile samples an event which one thread makes another blocked thread runnable. In the Go programming language, this can be implemented as an extension to the existing block profile of the pprof runtime library. The block profile records blocked cycles, the number of events, and the stack trace of a blocked thread, by sampling an event which a blocked thread becomes runnable, with a given sampling rate. To implement the wake-up profile, we also record the stack trace of the notifier thread into an event record. Figure 10 shows a wake-up profile of the sample program in Figure 7.

For user level and kernel level operating system threads, trace tools such as Ftrace [29], SystemTap [30], and LTTng [41] support profiling blocked threads. It will be possible to implement a wake-up profile for the operating system threads as an extension of these tools.

3.3 Synchronized Calling Context Tree

To generate a thread dependency graph, we merge thread profiles and wake-up profiles into an intermediate data structure, *synchronized calling context tree*, which is an extension to the traditional calling context tree [2].

First, we build a calling context tree by merging thread profiles. We calculate the average number of threads for each stack frame, by dividing the number of threads having the stack frame by the number of total samples.

Next, we add wake-up profiles into the calling context tree, and merge the stack top frames of the waiter thread and the notifier thread into a super node. Note that the wake-up profile does not contribute to the number of threads calculated from the thread profiles, since the wake-up profile is sampled base on synchronization events, and is not based on a timer. Figure 11 shows an example of a synchronized calling context tree, which was used to generate the thread dependency graph in Figure 8. Nodes and links with bold boundaries are from the thread profile. Nodes and links with thin boundaries are from the wake-up profile. An enclosing node is a super node which merges a waiter node and a notifier node.

Finally, we generate a thread dependency graph by merging nodes of the synchronized calling context tree which belong to the same thread into a corresponding invoking thread node of the thread dependency graph. A super node in the synchronized calling context tree becomes a blocked thread node, and asynchronous dependency links are added from the blocked thread node to the notifiers. For each root function annotated, a custom thread node is created and the all sub tree nodes are merged into it. The leaf nodes of the synchronized calling context tree are merged into a runnable thread node or a blocked thread node, depending on the thread status.

4 EXPERIMENTS

This section demonstrates that our bottleneck detection using a thread dependency graph can identify layered bottlenecks, and mitigating the detected bottlenecks improves the throughput and scalability. Note that estimating the *amount* of performance improvement is out of the scope, as explained in Section 1. Rather, the experiments in this section confirm that:

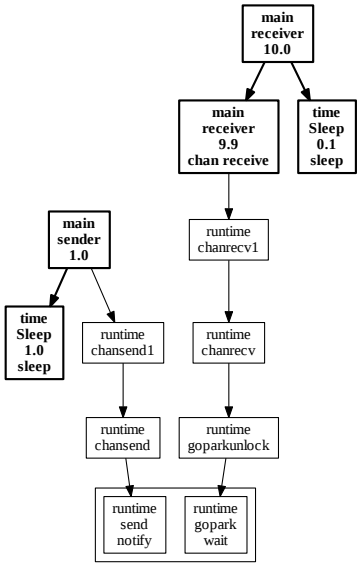

Figure 11: A synchronized calling context tree.

(1) bottleneck detection using a thread dependency graph can identify layered bottlenecks (which can be already known) of a given target application,
(2) performance improvement can be observed by optimizing one of the detected bottlenecks, and
(3) the optimized bottleneck is eliminated in a new thread dependency graph.

4.1 Environment

We used a measurement environment which consists of 3 virtual machines with 32 x86_64 processors each running at 2.0 gigahertz clock frequency, 64 gigabytes of memory, and 125 gigabytes of virtual block devices. All of the virtual machines are deployed in the same data center of IBM Cloud [16]. The operating system is Ubuntu 16.10.

We extended the Go language version 1.10.2 to implement the wake-up profile, described in Section 3, and compiled the target applications.

For profiled run, we sampled one wake-up event per every 1000 wake-up events, into the on-memory event buffer of the pprof runtime library, which is linked with the target application. Then we collected a thread profile and a wake-up profile per every 10 seconds, by using the default HTTP server of the pprof profiling library.

We used the Distributed Testing environment of the Apache JMeter [36] test tool to vary the target throughput of the benchmark. A master JMeter process controls multiple JMeter slave processes which run the same benchmark script. Each JMeter slave issues transactions to the target application with a constant arrival rate, by using the Constant Throughput Timer. We varied the target throughput by varying the number of the slave processes.

4.2 Acme Air Go

Scenario. We used the Acme Air Go benchmark [45], which is a Go language version of the original Acme Air benchmark [14], as an example of web application servers. The application implements a fictitious online airline service where users can login, logout, query and book flights, and cancel bookings. The configuration of the benchmark is shown in Figure 2.

The application server provides a Representational State Transfer (REST) [10] style API for the transactions above, which receives a transaction request as an HTTP request, and returns the result as an HTTP response containing a JavaScript Object Notation (JSON) [8] format document. Persistent data are stored as JSON documents in a backend MongoDB [24] database. The Go language version uses Gin Web Framework [22] to handle incoming HTTP requests, and mgo MongoDB driver [26] to manage the database on MongoDB.

We used the Acme Air workload driver [15] to drive the benchmark. The driver uses the Apache JMeter test tool to emulate multiple users as a JMeter thread group. Each thread group issues transactions with a given transaction mix and an injection rate. One JMeter slave spawns 64 client threads and targets to issue 2000 transactions per second.

Analysis. The custom thread node for the top level request handler is the node to execute the function `(*conn).server`, because this is a generic HTTP request handler of the standard library of the Go language, and the performance is measured by the throughput to handle the REST API requests. To refine the thread dependency graph for each transaction type, we also specified custom thread nodes for the REST API request handlers of the Acme Air benchmark, that is, the functions `Login`, `QueryFlights`, `BookFlights`, and `BookingByUser`. During the bottleneck analysis below, we identified bottlenecks in the data access layer of the Acme Air benchmark. For better readability, we also specified custom thread nodes for these functions, that is, the functions `(*Mongo).New` and `(*Mongo).Close`.

Figure 12 shows a thread dependency graph of the Acme Air Go web application server, when the target throughput is 16000 transactions per second. The numbered path starting from the invoking thread node labeled `(*conn).serve` to a blocked thread node labeled with `semacquire` represents the detected layered bottlenecks. The hierarchical structure of the layered bottlenecks suggests that:

- $369.9 \div 428.7 \approx 86\%$ of the cycles for all the REST API requests are blocked to allocate a new database connection to the backend MongoDB, and
- $369.4 \div 369.9 \approx 100\%$ of them, that is, $369.4 \div 428.7 \approx 86\%$ of the all requests, are blocked at acquiring a mutual exclusion lock to allocate a new database connection.

Further source code level analysis revealed that the last mutual exclusion lock is actually acquired by the `mgo` MongoDB driver [27]. This function `(*Session).New()` copies an existing database session with the same authentication information. At that time, it acquires a mutual exclusion lock to copy structured data such as the authentication information and underlying connection pools. This is an example that layered bottlenecks may be hidden in third parties' libraries, as explained in Section 1.

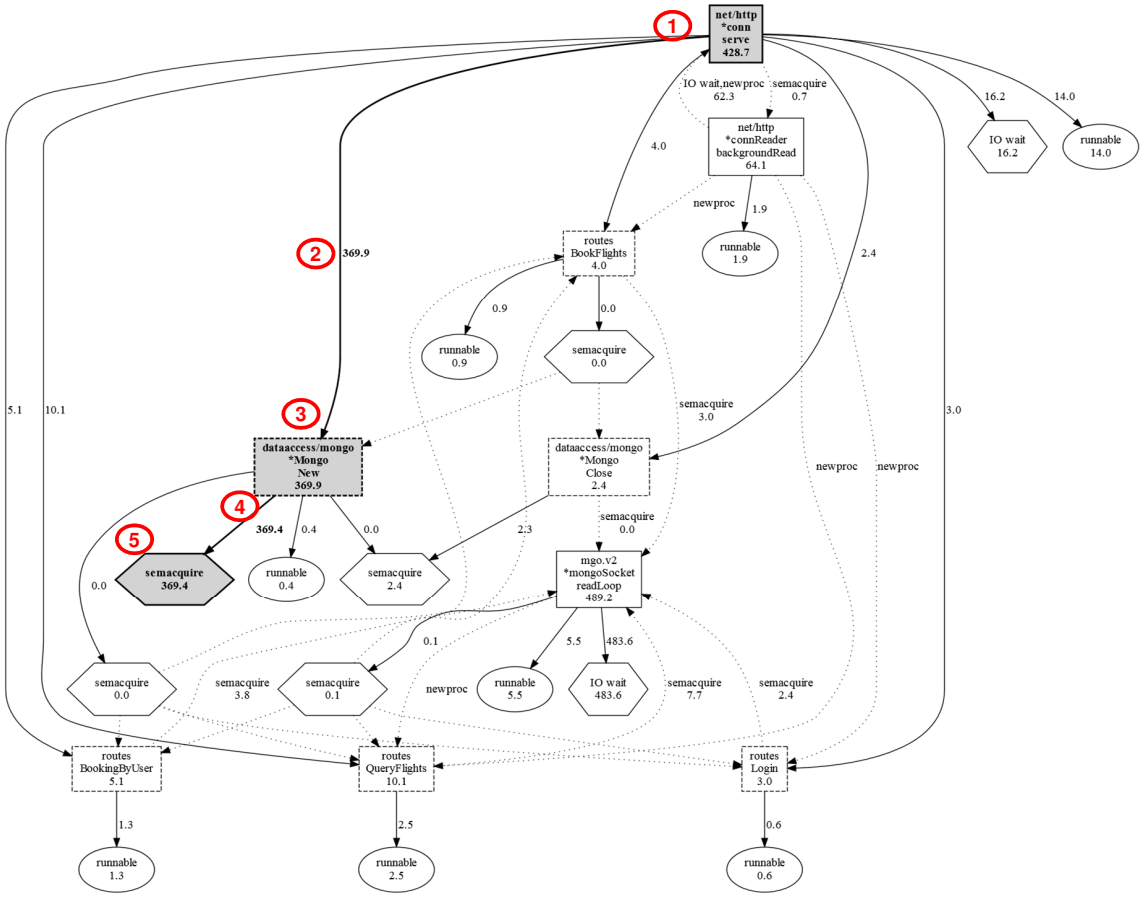

Figure 12: A thread dependency graph of the Acme Air Go web application benchmark.

Optimization. The source code level analysis above suggests an optimization to reuse a database connection at the application level without invoking the functions `New`, instead of the MongoDB driver level, since the Acme Air benchmark reuses a dedicated single MongoDB user for all database transactions. Figure 13 shows the performance improvement by this optimization, which improved the throughput up to +30% of the baseline configuration. Thus, the blocked thread node above is likely to be a layered bottleneck in the baseline configuration.

Figure 14 shows a new thread dependency graph of the Acme Air Go web application server after the optimization above, when the target throughput is 16000 transactions per second. The new layered bottlenecks are the numbered path starting from the invoking thread node labeled `(*conn).serve` to a blocked thread node labeled with `IO wait`. It suggests that:

- The dominating transaction is `QueryFlights` which consumes 125.5 ÷ 413.2 ≈ 30.4% of the cycles for all the REST API requests,
- 94.7 ÷ 125.5 ≈ 75.5% of them, that is, 94.7 ÷ 413.2 ≈ 22.9% of the REST API requests are waiting for the socket reader of the MongoDB driver, and

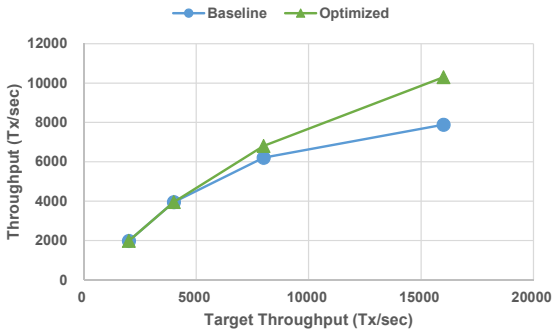

Figure 13: Performance improvement of the Acme Air Go web application benchmark by the optimization to cache live database connections.

- 318.4 ÷ 484.1 ≈ 65.8% of the socket reader are waiting for I/O operations with the backend MongoDB.

Thus, we can see the bottleneck optimized above has been eliminated from the new thread dependency graph.

Figure 14: A thread dependency graph of the Acme Air Go web application benchmark after the optimization.

4.3 Hyperledger Fabric

Scenario. As a more complicated and distributed target application, we used Hyperledger Fabric [39], which is an open source platform to implement a permissioned blockchain network. Hyperledger Fabric acts like an application server for a blockchain application, whose transactions are ordered by consensus of fault-tolerant orderers and transaction results are stored into a distributed "ledger", as an immutable chain of blocks verifiable by cryptographic hash codes. Participants, such as peers, orderers, consortium, organizations, and users are required to be cryptographically authenticated to enable highly secure and scalable blockchain networks [2].

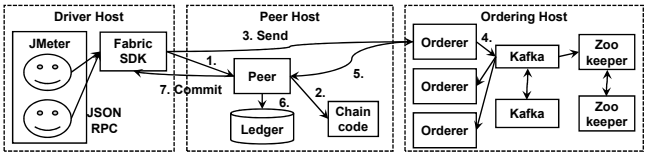

Figure 15: A configuration of the Hyperledger Fabric.

The configuration of the benchmark is shown in Figure 15. The transaction flow of Hyperledger Fabric is asynchronous due to its distributed nature. When a transaction to update the ledger is executed, what happens in a blockchain network are:

(1) a client sends a signed transaction proposal to peers,
(2) each peer simulates execution of the specified transaction in a container running a "smart contract", which is a blockchain application, and also called as "chaincode" for Hyperledger Fabric,
(3) the simulated results are sent back from the chaincode container to the peer, signed by the peer, sent back from the peer to the client, and the client sends the endorsements as a transaction request to an orderer,
(4) the orderer signs and sends the transaction request to the backend ordering service to (in this configuration, they are an Apache Kafka [35] cluster managed by an Apache Zookeeper [34] ensemble) determine the total order of the transactions,
(5) a block of the ordered transactions is delivered back from the ordering service to an orderer, and from the orderer to peers,
(6) each peer validates the block and updates the local ledger with the valid transactions, and

(7) the peer sends a block commit event to the client.

Messages between clients, peers, and orderers are exchanged via the gRPC protocol. We used Fabric Go Software Development Kit (SDK) [31] as a client for a blockchain network.

The version of Hyperledger Fabric is 1.2.0, which was the latest release when we started the experiments. We used a blockchain network consisting of one peer, three orderers, four Kafka servers, and three Zookeeper servers. The ledger database is Go LevelDB [43], which is embedded in the peer. An orderer creates a block of transactions and send it to the ordering service when the number of the sent transactions reaches 1000, the block size reaches 2 megabytes, or 1 seconds of the batch window timer expires. One JMeter slave spawns 128 client threads and targets to issue 400 transactions per second. Each slave submits transactions to a dedicated Fabric client process via the JSON-RPC protocol [19]. A client thread submits a transaction to endorsing peers and then to an ordering peer at Step 3 above, and then does not wait for completion of the transaction. Another dedicated client thread listens to commitment of blocks at Step 7 above, and calculates the throughput of committing transactions.

The blockchain application we used in this experiment executes daily post-trade netting for a stock exchange market [4]. For benchmarking, we used the most simple transaction which adds a trade record into a blockchain network. The transactions do not conflict each other. Figure 16 shows the scalability of adding trade records into a blockchain network configured as Figure 15. While the throughput of sending transactions to an orderer is mostly proportional to the target throughput, the throughput of committing transactions into the ledger degrades when the target throughput is equal to or greater than 1600 transactions per second.

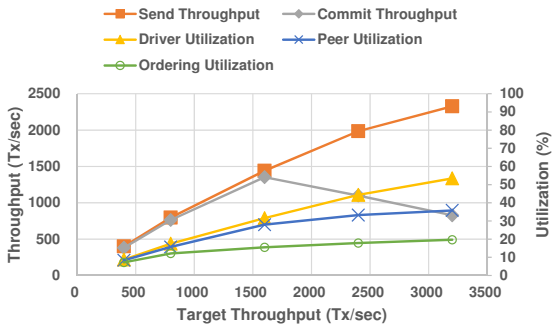

Figure 16: Throughput and resource utilization of the Hyperledger Fabric.

Analysis. We specified the custom thread node for the top level request handler for the function `_Deliver_DeliverFiltered_Handler`. This is a gRPC request handler to send a block commit event to a client at Step 7 of the transaction sequence. To refine the thread dependency graph, we also specified custom thread nodes for other gRPC request handlers. They are functions `_Endorser_ProcessProposal_Handler` and `_ChaincodeSupport_Register_Handler`.

Figure 17 shows a part of a thread dependency graph of the peer, when the target throughput is 3200 transactions per second. The whole graph is not shown here for better readability, since it is much larger than the one for the Acme Air. The numbered path starting from the invoking thread node labeled `_Deliver_DeliverFiltered_Handler` suggests that:

- in $7.2 \div 8.0 = 90\%$ of the total cycles, the block event listeners wait for the deliver block handler, which is labeled with `deliverBlocks.func1`,
- in $6.9 \div 7.2 \approx 96\%$ of the total cycles, the deliver block handler waits for the block committer, labeled with `deliverPayloads`,
- in $0.4 \div 1.0 = 40\%$ of the total cycles, the block committer waits for the transaction validators, labeled with `Validate.func1.1`, and
- in $8.3 \div 13.1 \approx 63.4\%$ of the total cycles, the validators are blocked at acquiring a mutual exclusion lock, which is also acquired by the endorser.

The mutual exclusion lock at the last step is acquired to update the eviction order in the Membership Service Provider (MSP) identity cache [47]. This feature was introduced to eliminate a previous bottleneck at verifying the same certificates multiple times [33]. While the cache greatly saved processor cycles, the cache itself can be a new bottleneck with a higher injection rate.

Optimization. We submitted a patch [44] to the Hyperledger Fabric to alleviate the bottleneck above. The proposed optimization is to modify the replacement algorithm from the least recently used algorithm to the second chance algorithm, in order to eliminate the lock contention. Figure 18 shows the performance improvement in the throughput of committing transactions by the proposed optimization, which applies the originally submitted patch set. We can see that the performance degradation at the target throughput of 2400 transactions per second has been resolved.

Figure 19 shows a new thread dependency graph of the peer after the optimization has been applied, when the target throughput is 3200 transactions per second. We can see that the lock contention at the MSP identity cache has been eliminated, and the new bottleneck observed is processor busy cycles spent by the committer, which is also mentioned in [3]. Thus, bottleneck detection using a thread dependency graph successfully identified layered bottlenecks in the Hyperledger Fabric.

4.4 Runtime Overhead

We evaluated the runtime overhead of collecting thread profiles and wake-up profiles, by measuring increases in transaction latency and busy cycles when the profiling is enabled. We compared a profiled version compiled by our customized Go language compiler against the original version compiled by the standard Go language compiler.

Both of the impacts on lattices and busy cycles were as small as up to 1.1%. For the application server process of the Acme Air Go web application benchmark, the increases in lances and busy cycles are up to 1.1% and 1.0%, respectively. For the peer process of the Hyperledger Fabric, the increases in lateness and busy cycles are up to 0.1% and 0.8%, respectively.

The majority of the overhead in busy cycles comes from the package `runtime`, which is the runtime library embedded into the target application. This is reasonable since the pprof runtime library and our implementation of the wake-up profile belong to this package.

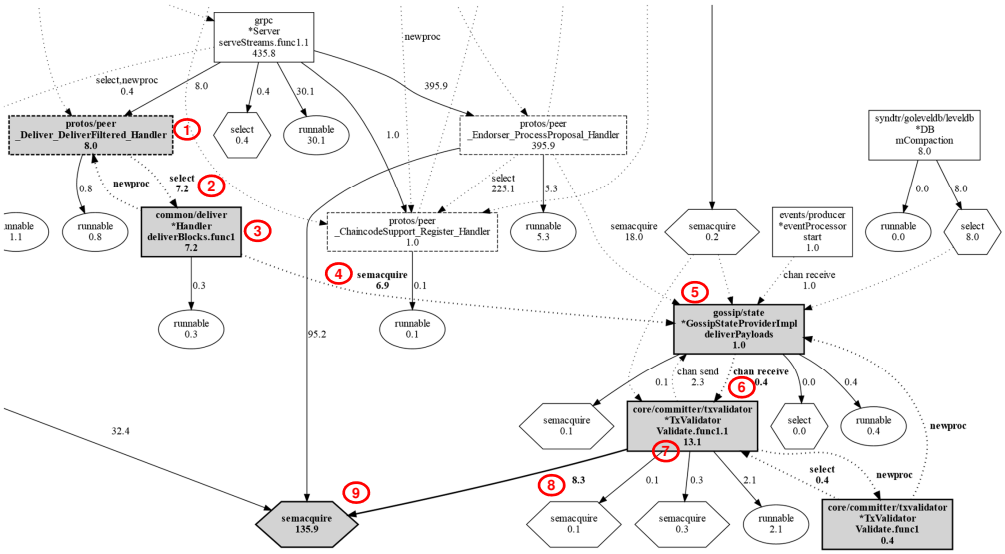

Figure 17: A part of a thread dependency graph of the Hyperledger Fabric peer.

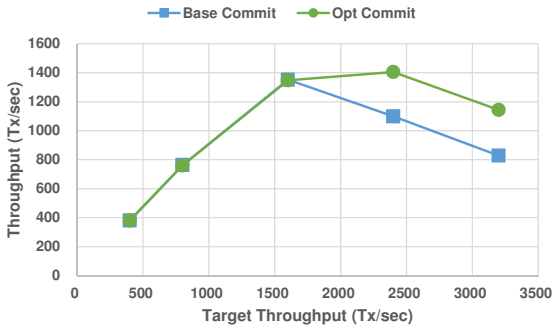

Figure 18: Performance improvement of the Hyperledger Fabric.

Figure 20 shows the functions with the top five largest profiling overhead in the Acme Air Go web application server. The primary overhead originates from allocating extra objects to export profiles, since the top three functions are for object allocation and garbage collection in the Go runtime library.

Figure 21 shows the functions with the top file largest profiling overhead in the Hyperledger Fabric peer. The primary overhead originates from sampling stack traces of the threads, since the top four functions are used by the pprof runtime library to collect profiles. The function `wakeupevent` is our custom function to sample wake-up profiles.

5 RELATED WORK

5.1 Model-based Approaches

The LQN [11, 42] is an extension to the traditional flat queueing network, and it is the most popular approach to analyze software bottlenecks due to its simplicity but broad coverage compared to other model-based approaches [21, 23]. Development and maintenance of performance model is an open challenge because today's software development is agile and modular, but still manual. We addressed this issue by integrating profile-based approaches.

Our approach can be considered to approximate the resource dependency graph [42] by the thread dependency graph extracted from execution profiles of the target application and to measure the queue lengths for the resources. A few differences of the thread dependency graph from the resource dependency graph are, the former:

- only models layered bottlenecks which are observed in execution profiles of the existing configuration,
- can be cyclic, because our target is just bottleneck detection, but is not synthetic performance analysis, and
- does not correlate a critical section to a corresponding mutual exclusion lock, because this is generally not available in execution profiles.

By limiting the scope of a thread dependency graph, we can implement our bottleneck detection by an extension to the existing profiling infrastructure, as described in Section 3, and with the small runtime overhead, as shown in Section 4.4.

5.2 Profile-based Approaches

Profile-based bottleneck analysis is a quite hot topic in recent studies. However, our approach is still unique because we analyze dependency of software bottlenecks based on execution profiles sampled by the profiling library of the underlying programming language. In contrast, most of the existing approaches either:

- only focus on hardware bottlenecks, that is, processor busy cycles [9],
- do not consider structures and dependency among software bottlenecks [1, 7],

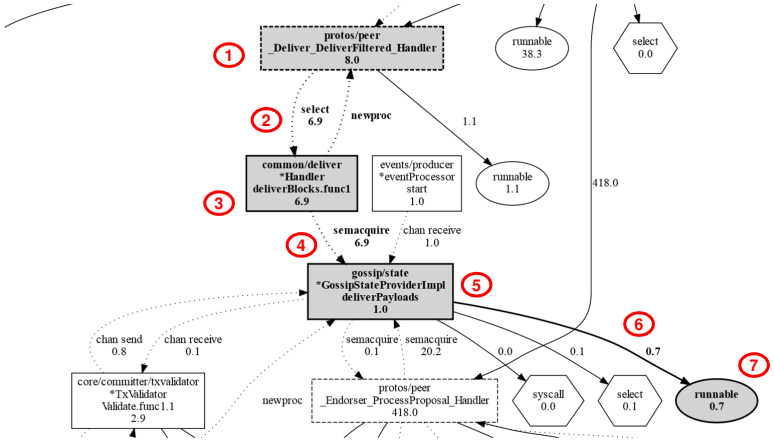

Figure 19: A part of a thread dependency graph of the Hyperledger Fabric peer after the optimization.

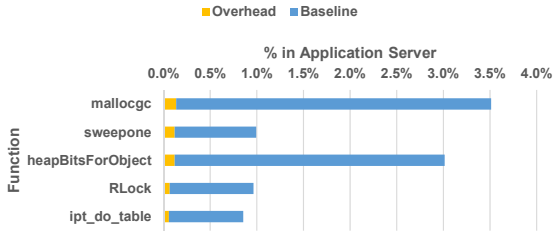

Figure 20: Functions with the top five largest profiling overhead in the Acme Air Go.

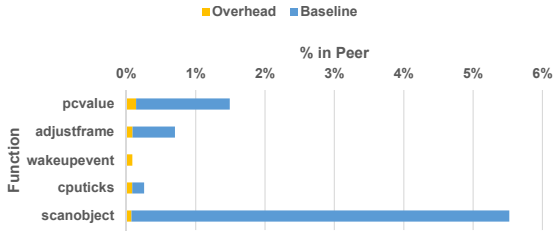

Figure 21: Functions with the top five largest profiling overhead in the Hyperledger Fabric.

- targets a specific execution model or programming framework [6, 9, 18, 48], or
- relies on tools such as traces, which incur larger runtime overhead [28, 48].

Recently, Zhou et al. [49] also proposed a novel profile-based bottleneck detection method which analyzes dependency among threads. The following differences exist between the two approaches:

- They rely on operating-system level tracing of every blocking and wake-up events, while we combine timer-based thread profiles and event-based wake-up profiles. This implementation allows us to minimize the profiling overhead, and to handle both blocked and busy cycles uniformly.
- They primarily focus on identifying a cyclic dependency among threads, while we identify bottlenecks as a sub graph of a thread dependency graph. This representation allows us to show the hierarchical structure of layered bottlenecks in a thread dependency graph.

6 CONCLUSION

We proposed a novel approach for detection of layered bottlenecks by combining model-based approaches and profile-based approaches, using a thread dependency graph built from thread profiles and wake-up profiles of the target application. As we have seen in Section 4, today's middleware is a quite complex LQN by itself, and we need a tool to analyze their layered bottlenecks on demand.

We believe our approach is complementary to the existing model-based bottleneck analyses, because profiling which is aware of the layered bottlenecks will be useful to develop a performance model for LQN, and to determine service demand of each component of the model.

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their insightful comments to improve early versions of this paper. In particular, Review #14C of the Middleware 2018 conference gave many constructive suggestions to organize the experiments and the discussions.

This work has been initiated by our joint study about the performance of Hyperledger Fabric together with the IBM Center for Blockchain Innovation. We would like to thank Fabian Yu Chin Lim, Venkatraman Ramakrishna, and Chun Hui Suen, for their contributions.

REFERENCES

[1] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance Analysis of Idle Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 739–753. https://doi.org/10.1145/1869459.1869519

[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 (PLDI '97). ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/258915.258924

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[4] Atsushi Santo, Ikuo Minowa, Go Hosaka, Satoshi Hayakawa, Masafumi Kondo, Shingo Ichiki, and Yuki Kaneko. 2016. *Applicability of Distributed Ledger Technology to Capital Market Infrastructure*. JPX Working Paper 15. Japan Exchange Group. http://www.jpx.co.jp/english/corporate/research-study/working-paper/b5b4pj000000i468-att/E_JPX_working_paper_No15.pdf.

[5] Terrehon Bowden, Bodo Bauer, Jorge Nerin, Shen Feng, and Stefani Seibold. 2009. The /proc Filesystem. https://www.kernel.org/doc/Documentation/filesystems/proc.txt.

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

[7] 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)*. ACM, New York, NY, USA, 355–372. https://doi.org/10.1145/2509136.2509529

[8] Ecma International. 2017. Standard ECMA-404 The JSON Data Interchange Syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[9] S. Eyerman, K. Du Bois, and L. Eeckhout. 2012. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 145–155. https://doi.org/10.1109/ISPASS.2012.6189221

[10] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. AAI9980887.

[11] Greg Franks, Shikharesh Majumdar, Neilson, Dorina Petriu, Jerome Rolia, and Murray Woodside. 1996. Performance Analysis of Distributed Server Systems. In *In Proceedings of the 6th International Conference on Software Quality*. 15–26.

[12] Shadi Ghaith, Miao Wang, Philip Perry, and Liam Murphy. 2014. Software Contention Aware Queueing Network Model of Three-tier Web Systems. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE '14)*. ACM, New York, NY, USA, 273–276. https://doi.org/10.1145/2568088.2576760

[13] Michael Haardt, Mike Coleman, Denys Vlasenko, and Michael Kerrisk. 2016. ptrace - process trace. http://man7.org/linux/man-pages/man2/ptrace.2.html.

[14] IBM Corporation. 2004. Acme Air Sample and Benchmark. https://github.com/acmeair/acmeair.

[15] IBM Corporation. 2010. Acme Air Workload driver. https://github.com/acmeair/acmeair-driver.

[16] IBM Corporation. 2018. IBM Cloud. https://www.ibm.com/cloud/.

[17] Internet Engineering Task Force. 2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. https://tools.ietf.org/html/rfc7230.

[18] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. 2006. Operating System Profiling via Latency Analysis. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. 89–102. http://www.usenix.org/events/osdi06/tech/joukov.html

[19] JSON-RPC Working Group. 2013. JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification.

[20] W. C. Lynch. 1972. Operating System Performance. *Commun. ACM* 15, 7 (July 1972), 579–585. https://doi.org/10.1145/361454.361476

[21] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. 1998. Modelling with Generalized Stochastic Petri Nets. *SIGMETRICS Perform. Eval. Rev.* 26, 2 (Aug. 1998), 2–. https://doi.org/10.1145/288197.581193

[22] Manuel Martínez-Almeida. 2014. Gin Web Framework. https://github.com/gin-gonic/gin.

[23] D. A. Menasce. 2002. Two-level iterative queuing modeling of software contention. In *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*. 267–276. https://doi.org/10.1109/MASCOT.2002.1167086

[24] MongoDB, Inc. 2018. mongoDB. https://www.mongodb.com/.

[25] J. E. Neilson, C. M. Woodside, D. C. Petriu, and S. Majumdar. 1995. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering* 21, 9 (Sep 1995), 776–782. https://doi.org/10.1109/32.464543

[26] Gustavo Niemeyer. 2010. The MongoDB driver for Go. https://github.com/go-mgo/mgo/tree/v2.

[27] Gustavo Niemeyer. 2011. Line 1579 of mgo/session.go. https://github.com/go-mgo/mgo/blob/9856a29383ce1c59f308dd1cf0363a79b5bef6b5/session.go#L1579.

[28] Tony Ohmann, Kevin Thai, Ivan Beschastnikh, and Yuriy Brun. 2014. Mining Precise Performance-aware Behavioral Models from Existing Instrumentation. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 484–487. https://doi.org/10.1145/2591062.2591107

[29] Red Hat Inc. 2017. ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[30] Red Hat Inc., IBM Corp., and Intel Corporation. 2013. SystemTap. https://sourceware.org/systemtap/wiki.

[31] SecureKey Technologies Inc. and IBM Corporation. 2018. Hyperledger Fabric Client SDK for Go. https://github.com/hyperledger/fabric-sdk-go.

[32] Yasir Shoaib and Olivia Das. 2011. Web Application Performance Modeling Using Layered Queueing Networks. *Electronic Notes in Theoretical Computer Science* 275 (2011), 123 – 142. https://doi.org/10.1016/j.entcs.2011.09.009 Fifth International Workshop on the Practical Application of Stochastic Modelling (PASM).

[33] P. Thakkar, S. Nathan, and B. Vishwanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. *ArXiv e-prints* (May 2018). arXiv:cs.DC/1805.11390

[34] The Apache Software Foundation. 2010. Apache ZooKeeper. https://zookeeper.apache.org/.

[35] The Apache Software Foundation. 2017. Apache Kafka – A distributed streaming platform. https://kafka.apache.org/.

[36] The Apache Software Foundation. 2018. Apache JMeter. https://jmeter.apache.org/.

[37] The Go Authors. 2009. The Go Programming Language. https://golang.org.

[38] The gRPC Authors. 2018. gRPC. https://grpc.io.

[39] The Linux Foundation. 2018. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric.

[40] The Linux Kernel Organization, Inc. 2014. The Linux Kernel Archives. https://www.kernel.org.

[41] The LTTng Project. 2018. LTTng: an open source tracing framework for Linux. https://lttng.org.

[42] P. Tregunno, Jing Xu, M. Woodside, D. Petriu, and G. Franks. 2006. Layered Bottlenecks and Their Mitigation. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*. 103–114. https://doi.org/10.1109/QEST.2006.23

[43] Suryandaru Triandana. 2012. LevelDB key/value database in Go. https://github.com/syndtr/goleveldb.

[44] Yohei Ueda. 2018. FAB-11321: Alleviating lock contention of MSP cache. https://jira.hyperledger.org/browse/FAB-11321.

[45] Y. Ueda and M. Ohara. 2017. Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 13–22. https://doi.org/10.1109/ISPASS.2017.7975266

[46] Murray Woodside, Dorina C. Petriu, José Merseguer, Dorin B. Petriu, and Mohammad Alhaj. 2014. Transformation challenges: from software models to performance models. *Software & Systems Modeling* 13, 4 (01 Oct 2014), 1529–1552. https://doi.org/10.1007/s10270-013-0385-x

[47] yacovm. 2017. Line 75 of msp/cache/cache.go. https://github.com/hyperledger/fabric/blob/v1.2.0/msp/cache/cache.go#L75.

[48] 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 2016, Savannah, GA, USA, November 2-4, 2016*. 603–618. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao

[49] 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

TRADEMARKS