Profile-based Detection of Layered Bottlenecks

Tatsushi Inagaki, Yohei Ueda, Takuya Nakaike, Moriyoshi Ohara
IBM Research – Tokyo
Software bottlenecks can diminish the maximum performance of a computer system.

- Capacities of software resources can prevent full utilization of hardware resources.
- Examples:
  - Insufficient number of pooled threads
  - Contended mutual exclusion locks
  - Blocking communication channels
- Also called *layered bottlenecks*, since a service request can hold software resources simultaneously from multiple layers of services.
Example – Acme Air Go web application

Throughput starts saturating around 6000 Tx/sec.

Processors are not fully utilized.
Where are the software bottlenecks?
Layered queueing network can analyze software bottlenecks, if a performance model is given.

• Models software bottlenecks as layers of queueing networks.
  • A request can use a hardware resource or a service from an underlying layer.

• Outputs:
  • Throughput
  • Utilization
  • Response time
  • Queue length

• But a performance problem often occurs when we do not know the performance model!
Our approach: estimating a layered performance model from execution profiles

• We build a thread dependency graph from given execution profiles to capture synchronization dependency among threads and mean thread counts.

• Top down graph traversal along the largest thread counts allows us to detect layered bottlenecks.

• We can build the graph with a small runtime overhead by extending existing profiling libraries in the Go language.
A thread dependency graph shows thread counts and synchronization dependency.

April 10, 2019

April 10, 2019

April 10, 2019
We build the graph from thread profiles and novel wake-up profiles.

Thread profiles are sampled by timer to reflect mean thread counts.

Wake-up profiles are sampled at synchronization events to detect dependency among threads.

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
The profiles are merged as a calling context tree, which is reduced into a thread dependency graph.

Call stacks
Captured in a wake-up profile

Converted to a dependency link in a thread dependency graph
Iterative steps of bottleneck detection and optimization

1. Compile the target application by the Go compiler with wake-up profiles enabled.
2. Run the target workload to periodically collect thread profiles and wake-up profiles.
3. Annotate to the function which handles the target transaction.
4. Post-process the profiles to generate a calling context tree and a thread dependency graph.
5. Identify the layered bottlenecks for the target transaction.
6. Design optimizations to mitigate the bottlenecks.
7. Apply the optimizations to the application and/or the workload.
8. Repeat from Step 1.
Top down traversal of the largest thread counts shows layered bottlenecks in Acme Air Go.

- Allocating a new DB connector
- A lock in mgo package at copying authentication information
- HTTP request handler

April 10, 2019
We can improve the scalability of Acme Air Go by pooling authenticated connections.
The profiling overhead was as small as 1% of the busy cycles.

Accumulated overhead was 1% of the total busy cycles.
Another example – Hyperledger Fabric, a permissioned blockchain network
Lock contention at an identity cache can be a bottleneck with Hyperledger Fabric v1.2.
After the contention is removed, the committer thread becomes the next bottleneck.
Related work

• Model-based approaches
  • Requires a performance model given.
  • A thread dependency graph approximates a resource dependency graph of the LQN labeled with measured queue lengths.

• Profile-based approaches
  • Do not handle dependency among threads.
  • Recently Zhou et al. also proposed trace-based bottleneck detection which focuses on cyclic dependencies among threads.
Conclusions

• We proposed a novel approach for detecting layered bottlenecks by combining model-based and profile-based approaches.

• Our approach can be implemented by extending profiling libraries of the Go language and works with a small runtime overhead.

• Today’s middleware is a complex LQN and our approach is useful to analyze its layered bottlenecks on demand.