wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events

Fang Zhou, Yifan Gan, Sixiang Ma, Yang Wang
The Ohio State University
Optimizing bottleneck is critical to throughput

• Bottleneck: factors that limit the throughput of application.

• Question: where is the bottleneck?
Where is the bottleneck?

• Both execution and waiting can create the bottlenecks.

<table>
<thead>
<tr>
<th>PID</th>
<th>%CPU</th>
<th>%MEM</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>930</td>
<td>20.0%</td>
<td>0.0%</td>
<td>test</td>
</tr>
<tr>
<td>931</td>
<td>50.0%</td>
<td>0.0%</td>
<td>test</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device</th>
<th>tps</th>
<th>kB_read/s</th>
<th>kB_wrttn/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>sda</td>
<td>7.37</td>
<td>0</td>
<td>1778.27</td>
</tr>
</tbody>
</table>
On-CPU & Off-CPU analysis

• On-CPU analysis
  • What execution events are creating the bottleneck?
  • Quite well studied: Recording execution time (perf, oprofile, etc.), Critical Path Analysis, Causal Profiler (Coz SOSP’15), etc.

• Off-CPU analysis
  • What waiting events are creating the bottleneck?
  • Common waiting events: lock contention, condition variable, I/O waiting, etc.
  • Lock-based (e.g., SyncPerf EuroSys’17, etc.) solutions are incomplete.
  • Length-based (e.g., Off-CPU flamegraph, etc.) solutions are inaccurate.
Key challenge of off-CPU analysis

Local impact vs global impact

• Local impact: impact on threads directly waiting for the event
• Global impact: impact on the whole application
• Large local impact does not mean large global impact
Overview of wPerf

• Goal: identify bottlenecks caused by all kinds of waiting events.
  • (Note: how to optimize bottlenecks requires the users’ efforts)

• To compute global impact
  • Generate a holistic view (wait-for graph) of the application
  • Theorem: knot in a wait-for graph must contain a bottleneck

• Results:
  • Up to 4.83x improvement in seven open source applications
Concrete example

Queue is a producer-consumer queue with max size $k$. Assume $k = 1$ for simplicity. Thread A (enqueue) blocks if queue size is 1. Thread B (dequeue) blocks if queue size is 0.
Concrete example

NIC
Thread A
Queue
Thread B
Disk

FunA
FunB
Sync
Waiting Event
R_i
Queue
NIC

Time
Concrete example

NIC
Thread A
Queue
Thread B
Disk

Time
Concrete example

- FunA
- FunB
- Sync
- Waiting Event
- $R_i$
- Queue
- NIC

NIC
- $R_1$
- $R_2$
- $R_3$

Thread A
- $R_1$
- $R_2$

Queue
- $R_2$

Thread B
- $R_1$

Disk
- $R_1$

Time
Concrete example

NIC

Thread A

Queue

Thread B

Disk

Time
Concrete example
Concrete example
Concrete example

- **FunA**
- **FunB**
- **Sync**
- **Waiting Event**
- **R_i**
- **Queue**
- **NIC**

**NIC**

**Thread A**

**Queue**

**Thread B**

**Disk**

Time
Concrete example

- FunA
- FunB
- Sync
- Waiting Event
- $R_i$
- Queue
- NIC

NIC

Thread A

Queue

Thread B

Disk

Time
Observation: waiting is important

Thread A

Thread B

Disk

Observations:
Waiting can have a large impact on throughput.
Longer waiting events may not be more important.
Contention is not the only waiting event that matters.
Observation: waiting is important

Observations:
- Waiting can have a large impact on throughput.
- Longer waiting events may not be more important.
- Contention is not the only waiting event that matters.
Observation: long waiting may not be important

Observations:

Waiting can have a large impact on throughput.

- Longer waiting events may not be more important.
- Large local impact does not mean large global impact.
- Contention is not the only waiting event that matters.
Observation: contention is not everything

Observations:
Waiting can have a large impact on throughput.
Longer waiting events may not be more important.
Contention is not the only waiting event that matters.
Key insights of wPerf

• Insight 1: to improve the throughput, we need to improve all the threads involved in request processing (worker threads).
  • Worker threads: request handling, disk flushing, garbage collection, etc.
  • Background threads: heartbeat processing, deadlock checking, etc.
  • See formal definition in the paper.

• Implication:
  • Bottleneck is an event whose optimization can improve all worker threads
Insight 1: a bottleneck is an event whose optimization can improve all worker threads.
Key insights of wPerf

Before optimization:
Thread A
Thread B
Disk

After optimization:
Thread A
Thread B
Disk

Optimizing sync can double the throughputs of all worker threads, so sync is a bottleneck.
Key insights of wPerf

• Insight 1: a bottleneck is an event whose optimization can improve all worker threads

• Insight 2: if thread B never waits for A, either directly or indirectly, then optimizing A’s event will not help B.
  • Implication: A’s event is not a bottleneck, if B is a worker thread.
Insight 2: if thread B never waits for A, either directly or indirectly, then optimizing A’s event will not help B.
Key idea of wPerf

• Insight 1: a bottleneck is an event whose optimization can improve all worker threads

• Insight 2: if thread B never waits for A, either directly or indirectly, then optimizing A’s event will not help B.
  • Implication: A’s event is not a bottleneck, if B is a worker thread.

• Key idea: narrow down the search space by excluding non-bottlenecks
Key idea of wPerf

- Construct a holistic view of the application using wait-for graph:
  - Each thread is a vertex.
  - A directed edge (A->B) means thread A sometimes is waiting for thread B.

- Theorem: Each knot with at least one worker contains a bottleneck.
  - A knot is a strongly connected component with no outgoing edges.
  - Optimizing events outside of knot cannot improve worker in the knot.

The wait-for graph of the example
Theory vs Practice

A → B ⇐ Disk

Theory

Practice
Solution: trim unimportant edges

• wPerf trims edges with little impact on throughput.
  • However, computing global impact is a challenging problem in the first place.

• Solution: use the waiting time spent on an edge to estimate the upper bound of the benefit of optimizing the edge.

• Challenge: nested waiting
An example of nested waiting

- **Time**
  - $t_0$
  - $t_1$
  - $t_2$

- **Processes**
  - A: Running
  - B: Running
  - C: Running

- **States**
  - Running
  - Waiting

- **Events**
  - Wake up

Diagram shows processes A, B, and C transitioning between running and waiting states over time $t_0$, $t_1$, and $t_2$.
Naïve approach to compute waiting time

Naïve approach:
A waits for B from t0 to t2, add (t2-t0) to A→B.
B waits for C from t0 to t1, add (t1-t0) to B→C.

Problem: underestimate B→C
wPerf’s solution

Detailed algorithm: cascaded re-distribution

Wait-for graph

$(t_2-t_0)$

$2(t_1-t_0)$
wPerf’s overall algorithm

1. Build the wait-for graph with weights.
2. Identify knot.
3. If the knot is smaller than a threshold, terminate.
4. Otherwise remove the edge with the lowest weight.
5. Go to 2.

Termination condition: smallest weight in the knot is larger than a threshold
- Threshold value depends on how much improvement the user expects.
Overall procedure of using wPerf

1. Annotation if necessary
2. Run the application with wPerf
3. Run wPerf analyzer
4. Investigate the source code of bottleneck
5. Optimize

This step requires user’s effort
Evaluation

• Case studies: Can wPerf identify bottlenecks in real applications?
  • We apply wPerf to seven open-source applications.
  • To confirm wPerf’s accuracy, we tried to investigate and optimize the bottlenecks reported by wPerf.

• Overhead:
  • How much does recording slow down the application?
  • Required user’s effort?
## Summary of case studies

<table>
<thead>
<tr>
<th>Application</th>
<th>Problem</th>
<th>Speedup after Optimization</th>
<th>Recording Overhead</th>
<th>Known fixes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase 0.92</td>
<td>Blocking write</td>
<td>2.74x</td>
<td>3.37%</td>
<td>Yes</td>
</tr>
<tr>
<td>ZooKeeper 3.4.11</td>
<td>Blocking write</td>
<td>4.83x</td>
<td>2.84%</td>
<td>No</td>
</tr>
<tr>
<td>HDFS 2.70</td>
<td>Blocking write</td>
<td>2.56x</td>
<td>3.40%</td>
<td>Yes</td>
</tr>
<tr>
<td>grep over NFS</td>
<td>Blocking read</td>
<td>3.9x</td>
<td>0.77%</td>
<td>No</td>
</tr>
<tr>
<td>BlockGrace</td>
<td>Load imbalance</td>
<td>1.44x</td>
<td>8.04%</td>
<td>No</td>
</tr>
<tr>
<td>Memcached</td>
<td>Lock contention</td>
<td>1.64x</td>
<td>2.43%</td>
<td>Partially</td>
</tr>
<tr>
<td>MySQL</td>
<td>Lock contention</td>
<td>1.42x</td>
<td>14.64%</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Case study: HBase

Workload: write workload with 1KB KV pairs.

Our solution: reducing blocking between Handler and RespProc

HBase uses parallel flushing to alleviate this problem, but the default setting of 10 handler threads is not enough.
Case study: HBase

Workload: write workload with 1KB KV pairs.

Our solution: reducing blocking between Handler and RespProc

HBase uses parallel flushing to alleviate this problem, but the default setting of 10 handler threads is not enough.

Use fast networks

Wait-for graph of original RegionServer
Case study: HBase

Workload: write workload with 1KB KV pairs.

Our solution: reducing blocking between Handler and RespProc

HBase uses parallel flushing to alleviate this problem, but the default setting of 10 handler threads is not enough.
Case study: HBase

Increasing handler count to 60 can improve throughput by 41%.

Comparing to the previous one, the weight of Handler->RespProc is much smaller (87.42 -> 16.54).

Optimize Handlers can further improve throughput.
Users’ efforts when using wPerf

- Annotation if necessary
  - 7 LOC for HBase
  - 12 LOC for MySQL

- Run the application with wPerf

- Run wPerf analyzer

- Investigate the source code of bottleneck
  - Usually a few hours

- Optimize
  - A few minutes to a week

This step requires user’s effort
Summary and future work

- wPerf identifies events with large impacts on all worker threads.

- wPerf can find bottlenecks others cannot find.

- In the future, we plan to extend wPerf to distributed systems.

- You can find the source code of wPerf in github. https://github.com/OSUSysLab/wPerf

- Poster number: 12