wPerf: Identifying Critical Waiting in Multi-threaded Applications

Fang Zhou
Ohio State University
zhou.1250@osu.edu

Yang Wang *
Ohio State University
wang.7564@osu.edu

1. Introduction

In a multi-threaded program, a thread could be executing some tasks or waiting for events. Therefore, when optimizing a multi-threaded program, a developer has two options: one is to identify critical tasks and optimize their code; the other is to identify and reduce critical waiting. Optimizing tasks can indirectly reduce waiting time in many cases, but waiting time may also be directly reduced by techniques like fine-grained locking or non-blocking APIs [1].

To illustrate the problem, we show an example of a badly designed multi-threaded program in Figure 1. Since thread B needs to wait for a response from thread C each time (Figure 1a), B and C actually cannot process requests in parallel at runtime (Figure 1b). As a result, B and C combined take 10ms to process a request, which is longer than the 8ms spent in A. Therefore, in this example, the combination of B and C becomes the bottleneck. Techniques like critical path analysis (CPA) [4, 6–10, 12, 13] and causal profiling [3] can identify correctly that *funB* and *funC* are critical tasks, but optimizing *funB* or *funC* is not the only option: another option is to make thread B non-blocking.

While many existing works have investigated how to find critical pieces of code that are worth optimizing, we find no tools can provide insights about which waiting relationships are important. Currently, identifying such problems relies on developers expertise and will become more challenging when the applications become more complex.

The goal of our paper is to develop a systematic method to identify critical waiting relationship whose optimization can lead to significant performance improvement. This paper makes the following contributions:

1. It presents a graph-based model to identify critical waiting in multi-threaded applications.
2. It proposes wPerf, a tool that can build the model for unmodified applications in Linux with 3% overhead on average.

2. Design

Our approach is based on a simple observation that if thread B never waits for A (directly or indirectly), then reducing

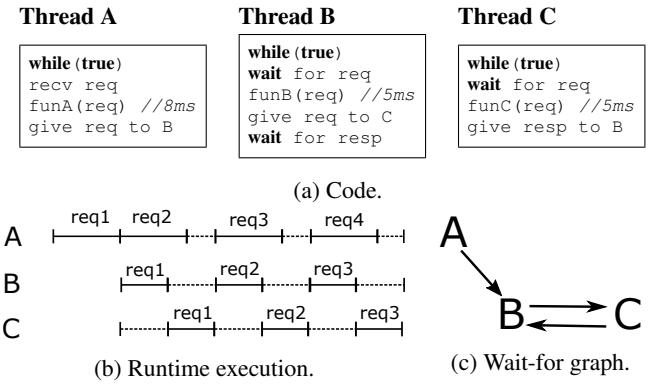
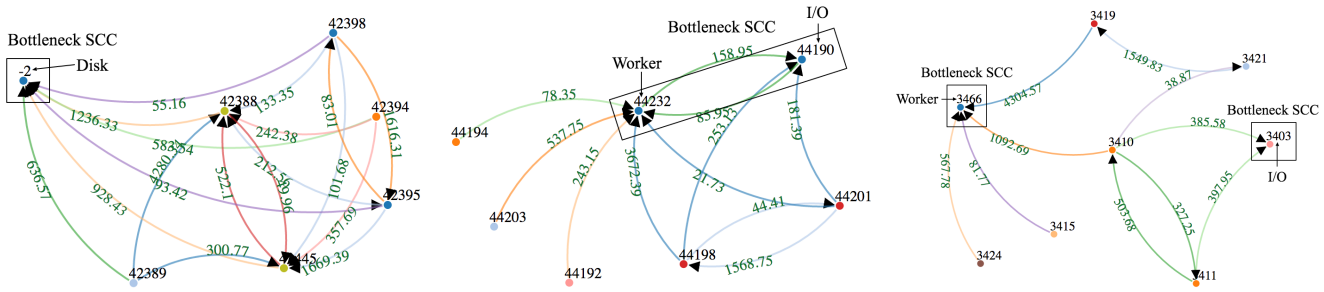


Figure 1: A badly designed multi-threaded program.

A's waiting time will not improve B, because neither B's execution speed nor B's waiting time is affected. Based on this observation, our approach models the execution of a multi-threaded program as a *wait-for graph*, in which each thread is a vertex and an edge from A to B means thread A sometimes waits for B. We can prove that in such a graph, bottleneck must include at least one vertex from each strongly connected component (SCC) with no outgoing edges. Intuitively, this conclusion is a generalization of our observation: if a group of threads never wait for other threads, then reducing other threads' waiting time will not improve this group of threads. Therefore, waiting among this group of threads is worth developers' efforts for further optimization.

Bottleneck SCCs with a single vertex are not much different from a bottleneck in a single-threaded program. On the other hand, bottleneck SCCs with more vertices indicate there exists cyclic wait-for relationships among its threads which could create a bottleneck in the system even when none of the threads on the cycle are saturated: such bottleneck is unique to multi-threaded programs. In practice, such cycles can be caused by various reasons including lock contention, load imbalance, and inefficient parallelism. Still taking Figure 1 as an example, since thread B needs to wait for the response from C and thread C needs to wait for requests from B, it creates a cyclic wait-for relationship between B and C (Figure 1c), which becomes the bottleneck. Therefore, in this example, B's waiting on C and C's waiting on B are critical, while A's waiting on B is not.

* This work is supervised by the author.



(a) MySQL on hard drive (default buffer) (b) MySQL on RAM-disk (default buffer) (c) MySQL on RAM-disk (8GB buffer)

Figure 2: Wait-for graphs of MySQL experiments. Vertices and edges are generated automatically by wPerf, and we add the text "bottleneck SCC" and thread names. Edges with weights less than 1% of experiment time have been removed.

If a bottleneck SCC is complicated, our approach further refines the SCC by removing edges that are less likely to be important. In principle, an edge is less important if reducing or even eliminating time spent on this edge has little impact on the overall. To identify importance of edges, we use waiting time spent on each edge as a heuristic. However, using waiting time directly as the weight of the edge may be misleading. For example, if thread A spends 30ms waiting for B and in the meanwhile, B spends 29ms waiting for C, we will conclude that these two edges are almost equally important if we look at waiting time directly, but this is not true: A's time waiting on B is mainly caused by B's waiting on C. Therefore, our approach adjusts weights if there are nested waiting relationship [2, 5]. Given such weights, our approach contiguously removes edges with small weights. At some point, the SCC may become disconnected and be separated into multiple sub-SCCs: our approach applies the above idea again to drop sub-SCCs with outgoing edges.

To build wait-for graphs for applications, we have developed wPerf. It is composed of an online recorder and an offline analyzer. The online recorder uses jprobe to trace all related function calls in the kernel. In particular, wPerf traces the *enqueue_futex* function and the *unqueue_futex* function to record information for futex related synchronization events. wPerf traces *__switch_to* and *try_to_wakeup* functions for disk I/O events. wPerf's recorder uses double buffering to store events and avoid blocking, and flushes synchronization events to the disk periodically. wPerf's analyzer uses a parallel algorithm to build the graph and applies the SCC to it.

3. Evaluation

To understand the effectiveness of wPerf, we have applied it to a number of server applications to identify whether there exists synchronization bottlenecks.

We run all experiments in a private cluster. We only show the MySQL results here because of the limited pages. We ran the TPC-C benchmark [14], which simulates an online transaction processing system.

We start by running experiments using the default setting of MySQL and storing data on a hard drive. We can gain a maximum throughput of 66.133 transactions/sec. As shown in Figure 2a, wPerf finds that in the wait-for graph, the bottleneck SCC contains a single vertex, which is the hard drive, and our measurement shows disk is close to be saturated. This is as expected, since hard-drive is well-known to cause I/O bottlenecks.

Next we create a RAM-disk and set MySQL to store all data in the RAM-disk. This time we can gain a maximum throughput of 2681.815 transactions/sec. In this experiment, wPerf finds all vertices in the wait-for graph is in a single SCC, so it starts to remove edges with low weights (less than 1% test time). As shown in Figure 2b, wPerf finds that in the wait-for graph, the bottleneck SCC contains two vertices: 44232 is the worker thread to process user's requests, and 44190 is the I/O thread to operate pages from storage. There exists a cyclic wait-for relationship between these two threads and the corresponding function stacks recorded by wPerf show that such wait-for relationship is caused by the lock of flushing slot. MySQL only provides 8 slots for single page flushing. When the slots are not enough, worker threads have to wait for the empty flushing slot on slot lock. The solution is to increase memory buffer size or increase the number of flushing slots. This bug has been reported in MySQL's developer forum [11].

We take a simple approach to move forward: we increase MySQL's buffer size to 8GB so that all data can be buffered in memory. Of course this may not be feasible in practice if the dataset is larger than memory size, and we use this simple solution to testify the problem reported by wPerf. This time we can gain a throughput of 3806.075 transactions/sec. As shown in Figure 2c, this time the wait-for graph contains two bottleneck SCCs: one contains the worker thread and one contains the I/O thread. The function stacks show that the worker thread now mainly waits for other worker threads because of contention on rows. Previous works [15, 16] have studied how to optimize the concurrency control of MySQL so we decide not to further optimize.

At runtime, wPerf needs to record synchronization-related events. wPerf first records logs in memory and when memory buffer becomes full, wPerf flushes data to disk in the background. The overhead of wPerf is about 3% on average. For the offline analyzer, when processing 10GB information recorded for MySQL, with the help of the parallel algorithm, it only takes 5 minutes on a 32-core machines .

References

[1] ALAM, M. M. U., LIU, T., ZENG, G., AND MUZAHID, A. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 298–313.

[2] ANDERSON, T. E., AND LAZOWSKA, E. D. Quartz: A tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1990), SIGMETRICS '90, ACM, pp. 115–125.

[3] CURTSINGER, C., AND BERGER, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 184–197.

[4] GARCIA, S., JEON, D., LOUIE, C. M., AND TAYLOR, M. B. Kremlin: Rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 458–469.

[5] HALL, R. J. Cpprofj: Aspect-capable call path profiling of multi-threaded java applications. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering* (Washington, DC, USA, 2002), ASE '02, IEEE Computer Society, pp. 107–.

[6] HE, Y., LEISERSON, C. E., AND LEISERSON, W. M. The cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2010), SPAA '10, ACM, pp. 145–156.

[7] HILL, J. M. D., JARVIS, S. A., SINIOLAKIS, C. J., AND VASILEV, V. P. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on* (Jan 1998), pp. 286–294.

[8] HOLLINGSWORTH, J. K., AND MILLER, B. P. Slack: A new performance metric for parallel programs. Tech. rep., 1994.

[9] MILLER, B. P., CLARK, M., HOLLINGSWORTH, J., KIERSTEAD, S., LIM, S. S., AND TORZEWSKI, T. Ips-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (Apr 1990), 206–217.

[10] MILLER, B. P., AND YANG, C.-Q. Ips: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS* (1987).

[11] Mysql bug no. 81376. https://bugs.mysql.com/bug.php?id=81376.

[12] OYAMA, Y., TAURA, K., AND YONEZAWA, A. Online computation of critical paths for multithreaded languages. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, UK, 2000), IPDPS '00, Springer-Verlag, pp. 301–313.

[13] SZEBENYI, Z., WOLF, F., AND WYLIE, B. J. N. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Nov 2009), pp. 1–12.

[14] TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-W home page. http://www.tpc.org/tpcw.

[15] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 495–509.

[16] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 279–294.