Causal Profiling

(Full Presentation)

Charlie Curtsinger Emery D. Berger

School of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

Problem

Developers use profilers to identify code that should be optimized. For each block of code, developers would like to know *"How much will optimizing this code improve performance?"* This is a question of causality, which traditional profilers cannot answer. Traditional profilers only observe execution. They measure the time spent in each function and the number of times each function was called. For serial programs, these measurements are sufficient to infer causality; however, concurrent and asynchronous programs present a problem. Threads in a parallel program can interfere, and not all code contributes to a program's execution time.

Causal Profiling

We introduce **Causal Profiling**, a novel profiling technique that relies on controlled experimentation rather than passive observation. Profilers that simply observe a program's execution require performance models to present usable information to developers. These models must explicitly handle all asynchronous operations, thread communication mechanisms, and all sources of contention. With causal profiling, *the program is the performance model*. There is no need to predict the effect of a performance change because a causal profiler measures it directly.

A causal profiler conducts a series of performance experiments to determine which blocks, if optimized, will cause the greatest improvement in a program's performance. For each experiment, the profiler makes a small change to the performance of a single block of code. Any effect in whole-program performance must be caused by this change. Ideally, the profiler would automatically modify the subject block to run faster, but this is generally not possible. Instead, a causal profiler tests a progression of criteria for each block to eliminate blocks that do not impact performance.

Finding Blocks for Experiments First, the profiler focuses on blocks that are frequently executed. It is reasonable to assume code that is never executed does not impact program performance, and thus optimizing it will not have any effect. A causal profiler selects blocks for performance experiments using the empirical distribution over code. Frequently executed blocks are more likely to be selected than infrequently executed blocks.

Traditional profilers would typically stop at this point, but frequently executed blocks are not necessarily good targets for optimization. One example of this is a simple server: on each request, the server performs some useful work to generate output for the connected client. Meanwhile, the server sends `keep-alive` messages to the client. Regardless of the number of `keep-alive` messages sent, optimizing the code to produce these messages will not decrease the server's request latency.

Slowdown Experiments Given a subject block of code, we would like to know if it has any effect on performance. A causal profiler establishes this with a slowdown experiment. During the experiment, the profiler inserts a small delay in the subject block each time it is executed. If the program's performance is unchanged, the block must be part of the `keep-alive` thread in our example server. The larger the degradation in the program's performance (relative to the size of the delay), the more important the block may be for optimization.

Somewhat surprisingly, slowing a block of code may actually improve program performance. If the subject block executes just before access to some contended resource, contention on this resource will be decreased. Locks, condition variables, monitors, cache lines, and the memory bus can all be sources of contention. Previous profiling techniques must intercept accesses to these resources to identify contention, but a causal profiler can identify contention regardless of the mechanism responsible. If improving a block's performance will ultimately increase contention and degrade performance, a causal profiler will not include it as a potential target for optimization.

"Speedup" Experiments While slowing a subject block will degrade performance, this does not guarantee that optimizing the block will improve performance. If our example server program performs its useful work in two threads that take equal time, slowing either thread will degrade performance but performance can only be improved by optimizing *both* threads. A causal profiler uses "speedup" experiments to eliminate these cases of perfectly balanced parallelism from its output. Speedup experiments emulate the effect of optimizing the subject block by "stealing time" from other threads. When the subject block is executed the profiler briefly pauses all other threads. The total pause time is later subtracted from the program's wall running time to compute the effective execution time. The resulting change in performance (measured using the effective time) tells us the expected effect of an optimization.

Online Performance Measurement

For each performance experiment, a causal profiler must compare the program's performance with and without a change to the subject block. Timing a program's whole execution is a common approach to measuring performance. However, causal profiling would be prohibitively slow if each performance experiment required a full run of the program. Instead, we ask developers to indicate which parts of

the program should run faster. This allows a causal profiler to conduct many performance experiments during a single run.

Measuring Throughput Many programs process a number of work items, then exit. A causal profiler can measure throughput if developers insert a marker (the `CAUSAL_PROGRESS` C macro) at the point where a single item is completed. For example, in the `matrix_multiply` program from Phoenix MapReduce, we inserted a progress point after the program computes a cell in the output matrix [7].

Measuring Latency While progress points allow a causal profiler to measure throughput over a short window of execution, developers are often concerned with request latency as well. To measure latency, developers simply insert the `CAUSAL_REQUEST_START` and `CAUSAL_REQUEST_FINISH` macros at request boundaries. There is no need specify which request is starting or finishing. Instead, we rely on Little's Law to compute average latency. Little's law states that $L = \lambda W$, where L is the number of active requests, λ is the arrival rate, and W is the wait time [4]. Latency (W) can be computed using the known arrival rate and the number of active requests.

Other Metrics Other online performance measurements can be collected by interposing on library functions. File input and output throughput can be measured by intercepting read and write operations, and latency for simple network servers can be measured by intercepting socket open and close operations.

Past Approaches

Previous efforts to generate meaningful profiles from concurrent programs generally fall into two groups: time attribution profilers, which attach "blame" to running code whenever threads are blocked; and critical path profilers, which trace program activity to construct an explicit graph of all dependent events.

Time Attribution Time attribution profilers use the states of other threads as a heuristic for dependence [1, 2, 9]. Code that executes while other threads are blocked is not necessarily responsible for the other threads' blocked state. These profilers may overstate the important of code, leading to wasted developer time. Unlike causal profiling, these techniques cannot identify contention when it does not affect threads' scheduler state.

Critical Path Profilers Critical path profilers trace a program's execution to collect all "happens-before" edges. An offline analysis can use this graph to identify the program's critical path and predict changes to the critical path. This approach has been applied in message-passing and distributed parallel systems, and for languages with first-class thread communication primitives [3, 5, 6, 8, 10]. Unlike causal profiling, tracing-based approaches must intercept all interaction between threads of control. For shared memory parallelism, this requires prohibitively expensive instrumentation.

Conclusion

Causal profiling is a novel technique to measure optimization potential. This measurement matches developers' assumptions about profilers: that optimizing highly-ranked code will have the greatest impact on performance. Causal profiling measures optimization potential for serial, parallel, and asynchronous programs without instrumentation of special handling for library calls and concurrency primitives. Instead, a causal profiler uses performance experiments to predict the effect of optimizations. This allows the profiler to establish causality: "optimizing function X will have effect Y," exactly the measurement developers had assumed they were getting all along.

Our prototype causal profiler, which runs on Linux and Mac OSX, is available for download at `http://github.com/plasma-umass/causal`.

References

[1] E. R. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, pp. 739–753. ACM, 2010.

[2] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *SIGMETRICS*, pp. 115–125, 1990.

[3] J. M. D. Hill, S. A. Jarvis, C. J. Siniolakis, and V. P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *PDP*, pp. 286–294, 1998.

[4] J. D. Little. OR FORUM: Little's Law as Viewed on Its 50th Anniversary. *Operations Research*, 59(3):536–549, 2011.

[5] B. P. Miller and C.-Q. Yang. Ips: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS*, pp. 482–489, 1987.

[6] Y. Oyama, K. Taura, and A. Yonezawa. Online computation of critical paths for multithreaded languages. In *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pp. 301–313. Springer, 2000.

[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pp. 13–24. IEEE Computer Society, 2007.

[8] Z. Szebenyi, F. Wolf, and B. J. N. Wylie. Space-efficient time-series call-path profiling of parallel applications. In *SC*. ACM, 2009.

[9] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPOPP*, pp. 229–240. ACM, 2009.

[10] C.-Q. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *ICDCS*, pp. 366–373, 1988.