**SCORE: Scalability-Oriented Optimization**

**Pls:** Emery D. Berger (University of Massachusetts Amherst) and Stephen N. Freund (Williams College)

**Participants:** Charlie Curtsinger (University of Massachusetts Amherst), Zander Majercik (Williams), Yitong Tse (Williams)

---

**SCORE Objectives**

Concurrent programming is notoriously difficult; programmers must take extreme care to orchestrate synchronization operations to avoid races, atomicity violations, and deadlocks. At the same time, programmers must eliminate synchronization whenever possible to maximize performance.

Resolving the tension between the two goals of safety and performance can be out of reach for all but expert programmers.

Scalability-oriented optimization (SCORE) maximizes multithreaded application performance while maintaining correctness.

**SCORE** will automatically locate scalability bottlenecks and transform the code or execution environment to eliminate them.

By handling architectural and synchronization optimizations without programmer involvement, **SCORE** lets programmers deliver applications that safely and effectively harness concurrency.

---

**Software Profilers**

Developers use profilers to select code for manual performance tuning. Profilers help developers identify code that:

- runs for a long time or
- runs many times

Profiling works well for serial programs because optimizing any part of the program will reduce total runtime. The more of that runtime a piece of code is responsible for, the larger the effect of optimizing that code.

What about parallel programs?

Not all code in a parallel program contributes to its total runtime.

Even though A ran just as long, optimizing A will not reduce runtime.

**Causal Profiling**

Causal profiling makes it possible to predict the effect of a hypothetical optimization without magic. This technique relies on two key ideas: virtual speedups and progress points.

**Virtual Speedups**

We can’t magically make code faster, but we can create this effect by slowing everything else down.

Each time A runs, we pause all other running threads.

After accounting for the size of the delay we inserted, we can measure the expected program speedup.

Are virtual speedups accurate?

This graph compares the real effect of optimizing a piece of code (y-axis) to the “predicted” effect measured by virtually speeding up that same code (x-axis).

The effect of a virtual speedup is highly correlated with the effect of a real speedup.

Virtual speedups even accurately predict cases where optimizing code will hurt performance.

**Progress Points**

End-to-end runtime is not a useful measurement of performance for many applications. Causal profiling uses progress points to measure both throughput and latency, even for programs that run forever.

**Cauz A Causal Profiler for Linux**

Both virtual speedups and progress points are implemented in cauZ, a prototype causal profiler for Linux.

CauZ uses sampling to approximate the virtual speedup mechanism described above.

CauZ relies on debug information to identify source lines in unmodified Linux executables.

**Using CauZ to Optimize a Program**

We demonstrate cauZ to find important code in the Ferret PARSEC application, manually optimize this code, and compare the results to cauZ’s predictions.

**Ferret: Image Similarity Search**

Ferret uses a pipeline model to process image similarity queries in parallel. There are four main stages, each assigned a fixed set of threads:

- Segmentation
- Feature Extraction
- Indexing
- Ranking

The causal profile for ferret shows important lines in three stages:

- Ranking
- Indexing
- Segmentation

We can “optimize” a pipeline stage by reassigning threads to that stage. Moving six threads from extraction to ranking yielded a 21% speedup.

**Get CauZ**

github.com/plasma-umass/coz

Supported by NSF Awards 1439008 and 1439042