Review of

Quantifying Behavioral Differences Between C and C++ Programs by Brad Calder, Dirk Grunwald, and Benjamin Zorn

Michelle Strout

January 19, 2006

What problem did the paper address?

Big picture problem

- How can we make C++ programs execute faster using hardware technology and compiler optimizations?

Why the problem is hard

- C++ is a new language (1995) and perceived as different than previous languages
- Computer architecture and optimizing compiler design has been driven by the behavior in C benchmarks

Specific Problem

- How do C++ programs behave in comparison to C programs?

Why should we care?

In 1995 C++ was becoming the standard programming language in industry

Similar program behavior studies in C and Fortran

- guide computer architecture design
- guide optimizing compiler design directly and indirectly through computer arch design

No other study had compared the behavior between to closely related languages

Benchmark selection

- C programs from SPEC benchmarks, which are still used extensively in the computer architecture community: gcc, tex, etc.
- C++ programs are academia and industry applications with many users

What is the approach used to solve this problem?

Empirical study of the behavior of C and C++ programs

- hypothesis: "...C++ programs behave quite differently from C programs and that these differences may have a significant impact on performance"
- select C and C++ benchmarks
  - that many people use
  - select some C++ benchmarks that have similar goals to C benchmarks
- collect static and dynamic statistics about the program using the ATOM tool, execute programs on DEC Alpha architecture
- also use cache simulation based on dynamic memory reference stats

Static statistics

- # of instructions, # of functions, #instrs/function, etc.

Dynamic statistics

- # of instructions, # func calls, # of indirect func calls, instrs per call, branching behavior, memory reference behavior, etc.

How does the paper support the conclusions it reaches?

Some of their conclusions (most based directly on empirical results)

- DHRYSTONE benchmark does not capture the behavior of C or C++ programs
- C++ programs have "shorter procedures that are often reached via indirect function calls", therefore need procedure inlining
- C++ programs need different branch prediction architectures
- C++ programs may have ILP problems
- C++ program performance improves with a customized memory allocator
  - actually did a study to show this
- link-time optimizations will be important

Future Research Questions

How do specific optimizations and reasonable optimization combinations affect performance differently in C and C++ programs?

- constant propagation could help virtual method resolution
- how many loops are parallelizable?

What affect does alias/pointer analysis precision have on the program optimizations that are possible and their affect on performance?

How much ILP is available in C++ programs? Similar to the Wall study.

Is their a relationship between conditional branch directions and indirect function call targets?

Critique

Hypothesis

- have one, which is great
- the hypothesis is weak and the experiments can only validate the first part

"Truth in advertising"

- points out in multiple places that the results rely heavily on the set of benchmarks
- which compiler you use and which version of the operating system are important

Empirical results

- in each section they describe why the measurements they are taking are important

Benchmarks

- DHRYSTONE benchmarks were meant to model "average system behavior", they don't tell us what programs Weicker used to derive this
- They say that input doesn't have an effect. That is definitely not the case for performance in irregular applications, but they are studying #function calls, etc.

Relation to CS653

Shows how to define a program performance problem

- carefully select a set of benchmarks
- collect statistics and calculate metrics about program behavior
- determine how these statistics and metrics affect execution time
  - they used previous knowledge about this

The next step

- develop program optimizations (including the analyses that support them) to change those statistics and metrics in a way that improves execution time

Related possible project

- profiling benchmarks with Tau
- redo the profiling after an optimizing compiler has “optimized” the benchmarks