Is
Code Optimization
Research Relevant?

Bill Pugh

Univ. of Maryland

Motivation

- A Polemic by Rob Pike
- Proebsting's Law
- *Impact of Economics on Compiler Optimization* by Arch Robison
- Some of my own musings

Systems Software Research is Irrelevant

- A Polemic by Rob Pike
- An interesting read
- I’m not going to try to repeat it
  - get it yourself and read

Impact of Compiler Economics on Program Optimization

- Talk given by KAI's Arch Robison
- *Compile-time program optimizations are similar to poetry: more are written than actually published in commercial compilers. Hard economic reality is that many interesting optimizations have too narrow an audience to justify their cost in a general-purpose compiler and custom compilers are too expensive to write.*

Proebsting's Law

- Moore's law
  - chip density doubles every 18 months
  - often reflected in CPU power doubling every 18 months
- Proebsting's Law
  - compiler technology doubles CPU power every 18 years

Todd's justification

- Difference between optimizing and non-optimizing compiler about 4x.
- Assume compiler technology represents 36 years of progress
  - compiler technology doubles CPU power every 18 years
  - less than 4% a year

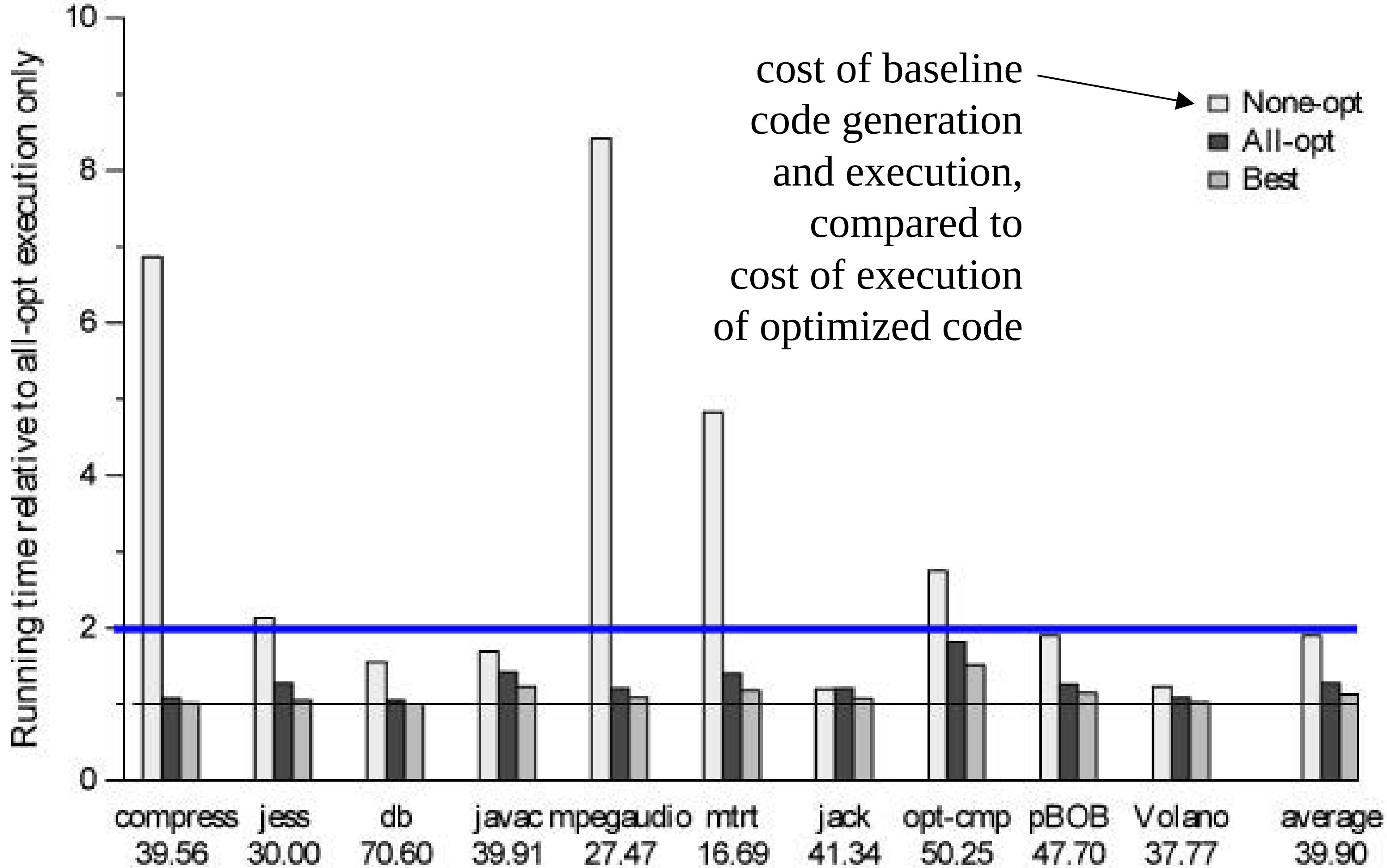
Let’s check Todd’s numbers

- Benefits from compiler optimization
- Very few cases with more than a factor of 2 difference
- 1.2 to 1.5 not uncommon
  - gcc ratio tends to be low
    - because unoptimized version is still pretty good
- Some exceptions
  - Matrix matrix multiplication

Jalepeño comparison

- Jalepeño has two compilers
  - Baseline compiler
    - Simple to implement, does little optimization
  - optimizing compiler
    - aggressive optimizing compiler
- Use result from another paper
  - compare cost to compile and execute using baseline compiler
  - vs. execution time only using opt. compiler

Results (from Arnold et al., 2000)



Benefits from optimization

- 4x is a reasonable estimate, perhaps generous
- 36 years is arbitrary, designed to get the magic 18 years
- where will we be 18 years from now?

18 years from now

- If we pull a Pentium III out of the deep freeze, apply our future compiler technology to SPECINT2000, and get an additional 2x speed improvement
  - I will be impressed/amazed

Irrelevant is OK

- Some of my best friends work on structural complexity theory

- But if we want to be more relevant,
  - what, if anything, should we be doing differently?

Code optimization is relevant

- Nobody is going to turn off their optimization and discard a factor of 2x
  - unless they don’t trust their optimizer
- But we already have code optimization
  - How much better can we make it?
  - A lot of us teach compilers from a 15 year old textbook
  - What can further research contribute?

Importance of Performance

- In many situations,
  - time to market
  - reliability
  - safety
- are much more important than 5-15% performance gains

Code optimization can help

- Human reality is, people tweak their code for performance
  - get that extra 5-15%
  - result is often hard to understand and maintain
  - “manual optimization” may even introduce errors
- Or use C or C++ rather than Java

Optimization of high level code

- Remove performance penalty for
  - using higher level constructs
  - safety checks (e.g., array bounds checks)
  - writing clean, simple code
    - no benefit to applying loop unrolling by hand
  - Encourage ADT's that are as efficient as primitive types
- Benefit: cleaner, higher level code gets written

How would we know?

- Many benchmark programs
  - have been hand-tuned to near death
  - use such bad programming style I wouldn’t allow undergraduates to see them
  - have been converted from Fortran
    - or written by people with a Fortran mindset

An example

- In work with a student, generated C++ code to perform sparse matrix computations
  - assumed the C++ compiler would optimize it well
  - Dec C++ compiler passed
  - GCC and Sun compiler failed horribly
    - factor of 3x slowdown
  - nothing fancy; gcc was just brain dead

We need high level benchmarks

- Benchmarks should be code that is
  - easy to understand
  - easy to reuse, composed from libraries
  - as close as possible to how you would describe the algorithm
- Languages should have performance requirements
  - e.g., tail recursion is efficient

Where is the performance?

- Most all compiler optimizations are micro-level benchmarks
  - Optimizing statements, expressions, etc
- The big performance wins are at a different level

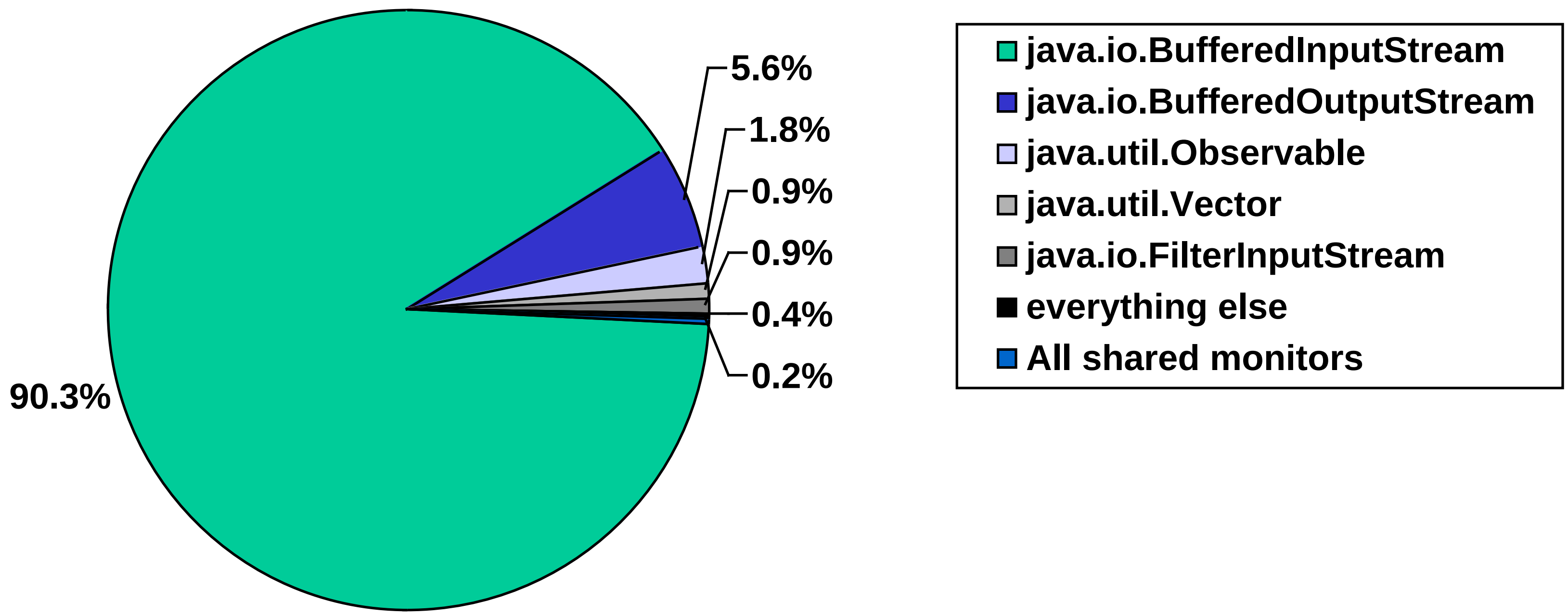
An Example

- In Java, synchronization on thread local objects is “useless”
- Allows classes to be designed to be thread safe
  - without regard to their use
- Lots of recent papers on removing “useless” synchronization
  - how much can it help

Cost of Synchronization

- Few good public multithreaded benchmarks
- Volano Benchmark
  - Most widely used server benchmark
  - Multithreaded chat room server
  - Client performs 4.8M synchronizations
    - 8K useful (0.2%)
  - Server 43M synchronizations
    - 1.7M useful (4%)

Synchronization in VolanoMark Client



7,684 synchronizations on shared monitors
4,828,130 thread local synchronizations

Cost of Synchronization in VolanoMark

- Removed synchronization of
  - java.io.BufferedInputStream
  - java.io.BufferedOutputStream
- Performance (2 processor Ultra 60)
  - HotSpot (1.3 beta)
    - Original: 4788
    - Altered: 4923 (+3%)
  - Exact VM (1.2.2)
    - Original: 6649
    - Altered: 6874 (+3%)

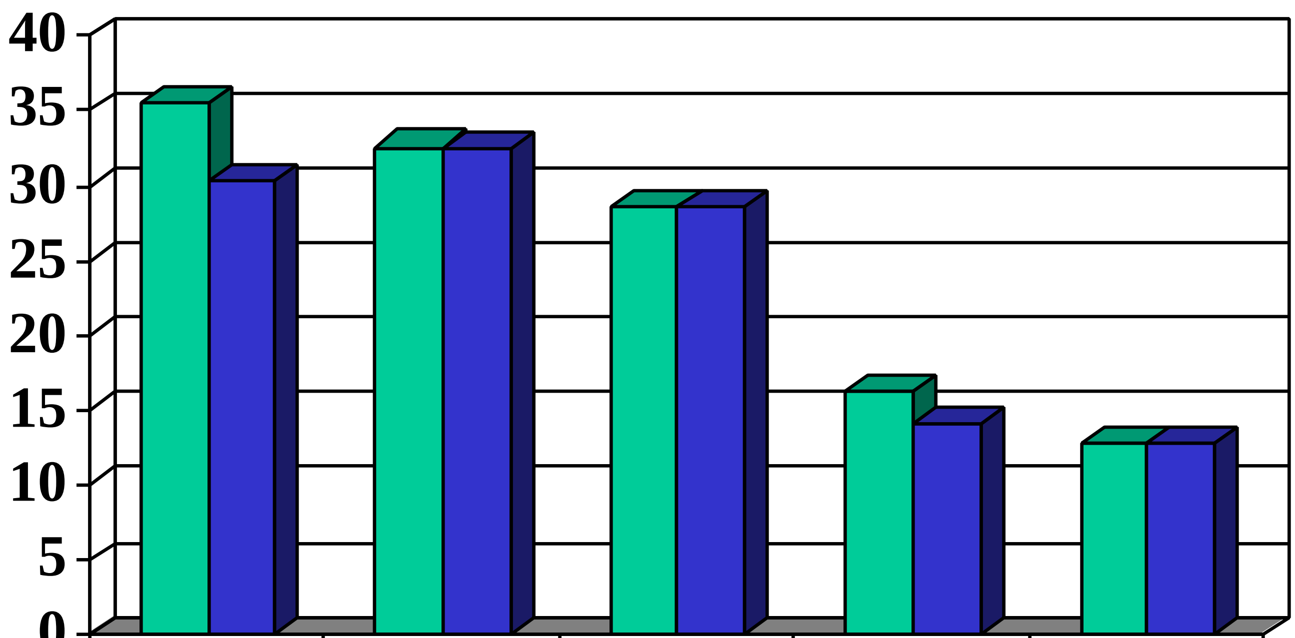
Some observations

- Not a big win (3%)
- Which JVM used more of an issue
  - Exact JVM does a better job of interfacing with Solaris networking libraries?
- Library design is important
  - BufferedInputStream should never have been designed as a synchronized class

Cost of Synchronization in SpecJVM DB Benchmark

- Program in the Spec JVM benchmark
- Does lots of synchronization
  - > 53,000,000 syncs
    - 99.9% comes from use of Vector
  - Benchmark is single threaded, all of it is useless
- Tried
  - Remove synchronizations
  - Switching to ArrayList
  - Improving the algorithm

Execution Time of Spec JVM _209_db, Hotspot Server



	Original	Use ArrayList	Use ArrayList and other minor	Change Shell Sort to Merge Sort	All
Original	35.5	32.6	28.5	16.2	12.8
Without Syncs	30.3	32.5	28.5	14.0	12.8

Lessons

- Synchronization cost can be substantial
  - 10-20% for DB benchmark
  - Better library design, recoding or better compiler opts would help
- But the real problem was the algorithm
  - Cost of stupidity higher than cost of synchronization
  - Used built-in merge sort rather than hand-coded shell sort

Small Research Idea

- Develop a tools that analyzes a program
  - Searches for quadratic sorting algorithms
- Don’t try to automatically update algorithm, or guarantee 100% accuracy
- Lots of stories about programs that contained a quadratic sort
  - not noticed until it was run on large inputs

Need Performance Tools

- gprof is pretty bad
- quantify and similar tools are better
  - still hard to isolate performance problems
  - particularly in libraries

Java Performance

- Non-graphical Java applications are pretty fast
- Swing performance is poor to fair
  - compiler optimizations aren't going to help
  - What needs to be changed?
    - Do we need to junk Swing and use a different API, or redesign the implementation?
  - How can tools help?

The cost of errors

- The cost incurred by buffer overruns
  - crashes and attacks
- is far greater than the cost of even naïve bounds checks
- Others
  - general crashes, freezes, blue screen of death
  - viruses

OK, what should we do?

- A lot of steps have already been taken:
  - Java is type-safe, has GC, does bounds checks, never forgets to release a lock
- But the lesson hasn't taken hold
  - C# allows unsafe code that does raw pointer smashing
    - so does Java through JNI
      - a transition mechanism only (I hope)
  - C# allows you to forget to release a lock

More to do

- Add whatever static checking we can
  - use generic polymorphism, rather than Java's generic containers
- Extended Static Checking for Java

Low hanging fruit

- Found a dozen or two bugs in Sun’s JDK
- hashCode() and equals(Object) not being in sync
- Defining equals(A) in class A, rather than equals(Object)
- Reading fields in constructor before they are written
- Use of Double-Checked Locking idiom

Low handing fruit (continued)

- Very, very simple implementation
- False negatives, false positives
- Required looking over code to determine if an error actually exists
  - About a 50% hit rate on errors

Data structure invariants

- Most useful kinds of invariants
- For example
  - this is a doubly linked list
  - n is the length of the list reachable from p
- Naïve checking is expensive
  - can we do efficiently?
  - good research problem

Data race detection

- Finding errors and performance bottlenecks in multithreaded programs is going to be a big issue
- Tools exist for dynamic data race detection
  - papers say 10-30x slowdown
  - commercial tools have a 100-9000x slowdown
  - lots of room for improvement

Where do we go from here?

As if People Programmed

- A lot of this comes back to:
- Doing compiler research, as though programs were written by people
  - who are still around and care about getting their program written correctly and quickly
  - and who also care about the performance
    - are willing to fix/improve algorithms
  - would happily interact with compiler/tools
    - if it was useful

If you want to get it published

- Compile dusty benchmarks
  - run them on their one data set
- All programs are “correct”
  - any deviations from official output is unacceptable
  - DB benchmark uses unstable shell sort
    - can’t replace it with stable merge sort
- No human involvement is allowed

Understandable

- Easy to measure the improvement a paper provides
  - what is the improvement in the SPECINT numbers?
- Much harder to objectively measure the things that matter

Consider

- A paper allows higher level constructs to be compiled efficiently
  - since they couldn’t be compiled efficiently before, no benchmarks use them
  - author provides his own benchmarks, show substantial improvement on benchmarks he wrote
  - one person’s high level construct is another’s contrived example

Human experiments ☹

- To determine if some tool can help people find errors or performance bottlenecks more effectively
  - need to do human experiments
  - probably with students
    - what do these results say about professional programmers?
  - Very, very hard
    - Done in Software Eng.

Some things to think about

- Most of the SPECINT benchmarks are done
  - no new research is going to get enough additional performance out of SPECINT
  - to warrant folding it into an industrial strength compiler
  - unless you come up with something very simple to implement

Encourage use of high-level constructs

- Reduce performance penalty for good coding style
- Eliminate motivation and reward for low level programming
- Example problems:
  - remove implicit down casts performed by GJ
  - compile a MATLAB-like language

New ways to evaluate papers

- We need well-written benchmarks
- We need new ways to evaluate papers
  - that take programmers into account

The big question

- What are we doing that is going to change
  – the way people use/experience computers,
  – or the way people write software
- five, ten or twenty years down the road?

- Software is hard…
  – improving the way software is written is harder