Compiler Optimization Research

(Based on talk by Prof. Bill Pugh, UMD)

1) Most Progress is in Hardware

- Progress by computer industry (hardware) is exciting
- Software research is not

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Operating Systems</th>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1979</td>
<td>1989</td>
<td>1999</td>
</tr>
<tr>
<td>8 MHz B086 4 Mb DRAM</td>
<td>33 MHz x86 32 Mb DRAM 10 Mb Ethernet Internet</td>
<td>1 GHz Pentium3 256 Mb DRAM 100 Mb Ethernet WWW</td>
</tr>
<tr>
<td>Unix</td>
<td>Linux</td>
<td>Unix</td>
</tr>
<tr>
<td>DOS</td>
<td>Windows 3.1</td>
<td>Windows NT</td>
</tr>
<tr>
<td>Cobol</td>
<td>C</td>
<td>C++</td>
</tr>
<tr>
<td>Fortran</td>
<td>Java</td>
<td>Perl, Python, PHP</td>
</tr>
<tr>
<td>LISP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2) Impact of Economics on Compiler Optimizations

- Assertion
  → Few new optimizations implemented in commercial compilers
- Commercial compilers
  → Expensive to build & maintain
- Compiler optimizations
  → Many interesting
  → Most narrowly applicable
  → General purpose compilers
  → Cannot justify expense
  → Custom compilers
  → Too expensive to write

3) 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

- Corollary
  → 1 year of code optimization research = 1 month of hardware improvement
  → No further need for compiler optimization research
  → Just wait a few months...

Todd’s Justification for Proebsting’s Law

- Assumptions
  → 4x performance improvement from optimizations
  → Compiler technology represents 36 years of progress

- Results in
  → Compiler technology doubles CPU power every 18 years
  → Improvement ≤ 4% a year
Checking Justification for Proebsting's Law

- Measure actual benefits from compiler optimization
- SPEC 95 benchmarks [Scott 2001]
  - Numeric Fortran code
  - DEC SPEC results (optimized) vs GNU f77 -O0 (unoptimized)
  - Integer C code
  - DEC SPEC results (optimized) vs DEC cc -O0 (unoptimized)
- Java benchmarks [Arnold 2000]
  - Jalapeno (optimized) vs Jalapeno (unoptimized)

Optimizations for SPECfp Benchmarks

Optimizations for SPECint Benchmarks

Optimizations for Java Benchmarks

Benefits from Compiler Optimization

- Average improvements from optimization
  - 8.1x for numeric codes (DEC SPEC results vs GNU f77 -O0)
  - 3.3x for integer codes (DEC SPEC results vs DEC cc -O0)
  - 2.0x for Java benchmarks (Jalapeno -O vs Jalapeno)
- SPEC comparisons exaggerate improvements, since compiler optimizations are carefully tuned and targeted
- 2-4x is a reasonable estimate for applying compiler optimizations to real programs, probably generous

Where Do We Go From Here?

- Current compiler optimizations
  - 2-4x improvements from optimization
- Past work on compiler optimization is relevant
  - Nobody is going to turn off optimization and discard a factor of 2x improvement
- What about the next 18 years?
  - Can we achieve another 2x improvement?
  - Is it worth the effort? (Wait 18 months for faster processors)
Compiler Optimization Research

• What won’t work
  → Take existing C / Fortran benchmarks (e.g., SPEC 95)
  → Apply complex, expensive program analyses / transformations
    → Automatic parallelization for multithreaded processors
    → Using interprocedural context-sensitive whole-path alias analysis of complex pointer-based data structures
  → Targeting existing RISC / x86 microprocessors

• Optimizations reaching point of diminishing returns
  → For current languages / applications / architectures
  → Too much work, not enough improvement

• So what is left?

Importance of Performance

• For general software, many issues dominate
  → Time to market
  → Maintainability
  → Reliability
  → Safety / security

• Much more important than another 4% / year speedup

Compiler Optimization Research

• So what compiler optimization research is relevant?

• Some suggestions
  → Targeting high-performance computing (HPC) applications
  → Exploiting new architectural features
  → Improving programmer productivity

• But only if performance improvement is significant
  → I.e., closer to 4% / month (processor) than 4% / year (compiler)

Overview

• Motivation
• High-performance computing (HPC)
• Exploiting new architectural features
• Improving programmer productivity

1) Targeting HPC Applications

• High performance computing applications
  → Computational science
  → Simulation using numerical models (molecules to galaxies)
  → Precision depends on computation power

• Unlike general applications, performance is important

• Compiler optimization research is worthwhile

• Caveat
  → Techniques may not be economical for general compiler
  → May produce programming tool instead of compiler

2) Exploiting New Architectural Features

• Moore’s law
  → Chip density doubles every 18 months

• Chip density improves performance
  → Smaller gate size = faster switching speed
  → Smaller chip = less wire delay

• But performance does not automatically double
  → 2x chip density ≠ 2x clock speed increase
  → 2x clock speed increase ≠ 2x performance improvement
Exploiting New Architectural Features

- Source of additional improvement
  - Extra transistors → more processor features

- Uses for extra transistors
  - Larger on-chip caches
  - Vector operations
  - Long instruction words (VLIW)
  - Out-of-order execution
  - Branch prediction
  - Value prediction
  - Predicated instructions
  - Multithreading
  - Speculative threads
  - Prefetching
  - Multi-core

- Many features require compiler optimizations
  - On-chip caches → locality optimizations
  - Vector operations → automatic vectorization
  - Long instruction words (VLIW) → instruction scheduling
  - Out-of-order execution → instruction scheduling
  - Predicated instructions → control dependence analysis
  - Multithreading, multi-core → automatic parallelization
  - Speculative threads → dependence analysis
  - Prefetching → software prefetching

- Otherwise limited benefit from new features

Exploiting New Architectural Features

- Compiler research can thus focus on new features
  - Vector units
  - Long instruction words
  - Predicated instructions
  - Large on-chip caches
  - Multithreading
  - Multi-core

- Improvements can be much larger than 4% / year

- Key
  - Pick architectural features responsible for largest gains
  - Balance improvement against compiler implementation effort
  - Avoid falling back into 4% improvement / year range

Architectural Features - Pentium 3 vs 4 Binaries

Linpack, 1.8 Ghz Intel Xeon

- 72%
- 94%

Other improvements – 3% SPECint 2000, 8% SPECfp 2000

[Mehta+ 2002]

Exploiting New Architectural Features

- Locality
  - Processors faster than memory, network
  - In cache → avoid memory latency
  - On processor → avoid network latency

- Growing processor – memory gap
  - Performance impact of locality increasing
  - Prime candidate for compiler optimizations

Architectural Features - SSE2 Vector Instructions

ATLAS, 2.2 Ghz Intel Xeon

- 90%

[Mehta+ 2002]
3) Improving Programmer Productivity

- Improving programmer productivity is probably most important problem facing computer science today
  - How can compiler optimization research help?

- Areas
  - Discourage manual optimizations
  - Encourage high-level languages
  - Reduce cost of
    - High-level language constructs
    - Error-checking/security
  - Provide/exploit user feedback

- Goal is higher productivity

Improving Productivity - High-Level Languages

- People use low-level languages for performance
  - Use assembly code instead of C
  - Use C instead of C++
  - Use C++ instead of Java
  - Use MPI instead of HPF

- Problem
  - Low-level programming generally less productive
  - May even introduce errors
    - Malloc/free vs. garbage collection
    - Arbitrary pointer arithmetic vs. multidimensional arrays
    - Arbitrary type casting vs. safe types
    - Message deadlock in message-passing programs

Improving Productivity - Reduce Manual Optns.

- People tweak their code for performance
  - "Register" variable declarations
  - Write compact, dense code
  - Unroll loops by hand

- Problem
  - Code hard to understand and maintain
  - More difficult to optimize
  - May even introduce errors

- Compiler optimizations help
  - Handle simple cases, remove temptation

- Result → less hand-optimized code, easier to maintain

Improving Productivity - High-Level Languages

- Compiler optimizations help
  - Reduce penalty for high-level language constructs
    - Type safety
    - Objects
    - Inheritance
    - Abstract data types
    - Parametric polymorphism
    - Many of these features are already in Java compilers

- Result → cleaner, high-level code
Summary

- Compiler optimization research can be relevant
  - But not by doing the same thing for the next 18 years

- Some relevant research areas
  - High performance computing applications
  - Exploiting new processor architectural features
  - Improving programmer productivity

- Caveats
  - Only care about performance if improvement = 4% / year
  - If narrowly applicable, may produce programming tool instead of compiler