Is Compiler Optimization Research Still Relevant?

Chau-Wen Tseng

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

Department of Computer Science

University of Maryland, College Park
Compiler Optimizations

◆ Question
  - Is compiler optimization (research) still relevant?

◆ Reasons
  - Hardware improving at much faster rate
  - Compiler optimizations are not implemented in commercial compilers, anyway

◆ Better examine allegations in more detail…
Some Advocates For Irrelevancy

1. **Rob Pike, Bell Labs**
   - “Systems Software Research Is Irrelevant”

2. **Arch Robison, KAI**
   - “Impact of Economics on Compiler Optimization”

3. **Todd Proebsting, Microsoft**
   - Proebsting's Law
1) Systems Software Research is Irrelevant (?)

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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>1979</th>
<th>1989</th>
<th>1999</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Systems</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming Languages</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 Mhz 8088</td>
<td>33 Mhz x386</td>
<td>1 Ghz Pentium3</td>
</tr>
<tr>
<td></td>
<td>4 Mb DRAM</td>
<td>32 Mb DRAM</td>
<td>256 Mb DRAM</td>
</tr>
<tr>
<td></td>
<td>Modem</td>
<td>10 Mb Ethernet</td>
<td>100 Mb Ethernet</td>
</tr>
<tr>
<td></td>
<td>USENET</td>
<td>Internet</td>
<td>WWW</td>
</tr>
<tr>
<td></td>
<td>Unix</td>
<td>Unix</td>
<td>Unix</td>
</tr>
<tr>
<td></td>
<td>DOS</td>
<td>Windows 3.1</td>
<td>Windows NT</td>
</tr>
<tr>
<td></td>
<td>USENET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 Mhz 8088</td>
<td>33 Mhz x386</td>
<td>1 Ghz Pentium3</td>
</tr>
<tr>
<td></td>
<td>4 Mb DRAM</td>
<td>32 Mb DRAM</td>
<td>256 Mb DRAM</td>
</tr>
<tr>
<td></td>
<td>Modem</td>
<td>10 Mb Ethernet</td>
<td>100 Mb Ethernet</td>
</tr>
<tr>
<td></td>
<td>USENET</td>
<td>Internet</td>
<td>WWW</td>
</tr>
<tr>
<td></td>
<td>Unix</td>
<td>Unix</td>
<td>Unix</td>
</tr>
<tr>
<td></td>
<td>DOS</td>
<td>Windows 3.1</td>
<td>Windows NT</td>
</tr>
<tr>
<td></td>
<td>USENET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 Mhz 8088</td>
<td>33 Mhz x386</td>
<td>1 Ghz Pentium3</td>
</tr>
<tr>
<td></td>
<td>4 Mb DRAM</td>
<td>32 Mb DRAM</td>
<td>256 Mb DRAM</td>
</tr>
<tr>
<td></td>
<td>Modem</td>
<td>10 Mb Ethernet</td>
<td>100 Mb Ethernet</td>
</tr>
<tr>
<td></td>
<td>USENET</td>
<td>Internet</td>
<td>WWW</td>
</tr>
<tr>
<td></td>
<td>Unix</td>
<td>Unix</td>
<td>Unix</td>
</tr>
<tr>
<td></td>
<td>DOS</td>
<td>Windows 3.1</td>
<td>Windows NT</td>
</tr>
<tr>
<td></td>
<td>USENET</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Systems Software Research is Irrelevant (?)

- **Decline of systems software research**
  - Standardization has taken over
    - UNIX, Windows, TCP/IP…
  - Innovations coming from industry
    - UNIX, WWW, Perl, Java…

- **Current systems software research**
  - Incremental
  - Just experimental comparisons
  - Few new ideas
2) Impact of Economics on Compiler Opti.

◆ **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

![Bar chart showing speedup for various SPECfp benchmarks. The chart includes labels for 'su2cor', 'hydro2d', 'tomcatv', 'fppp', 'swim', 'apsi', 'turb3d', 'mgrid', 'applu', and 'Avg'. The average speedup is highlighted as 8.1.]
Optimizations for SPECint Benchmarks

- li
- perl
- gcc
- jpeg
- vortex
- go
- compress
- m88ksim
- Avg

Average Speedup: 3.3

Scott 2001
Optimizations for Java Benchmarks

\[
\text{Ratio} = \frac{\text{compilation + execution time}}{\text{execution time of optimized code}}
\]

Arnold+ 2000

Unoptimized code
All optimizations
Best optimized
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 their optimization and discard a factor of 2x improvement

- What about the next 18 years?
  1. Can we achieve another 2x improvement?
  2. Is it worth the effort? (Wait 18 months for faster processors)
18 Years From Now

Bill Pugh, UMD

“If we pull a Pentium III out of the deep freeze, apply our future compiler technology to SPECint 2000, and get an additional 2x speed improvement, I will be impressed / amazed”
Compiler Optimization Research

◆ What won’t work
  - Take existing C / Fortran benchmarks (e.g., SPEC 95)
  - Apply complex, expensive program analyses / transformations
    - Automatic parallelization 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

- So should we all
  - Give up compiler optimization research, and
  - Become software engineers?
Compiler Optimization Research

So what compiler optimization research is relevant?

My suggestions

1. Targeting high-performance computing (HPC) applications
2. Exploiting new architectural features
3. 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)
  - Parallelism

◆ Exploiting new architectural features
  - Locality

◆ Improving programmer productivity
  - High-level languages
  - Error-checking
  - Benchmarks
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
Targeting HPC Applications – Parallelism

- Parallel computing
  - Custom supercomputers too expensive
  - Combine power of multiple processors

- Many compiler optimization research areas
  - Identify parallelism (NOT!)
  - Program transformations
  - Load balancing
  - Generate message-passing code
  - Supporting high-level parallel programming
    - Not everyone wants to program forever using MPI
Message-Passing Compilers (Fortran D)

- **Message-passing computers**
  - Require explicit communication (send/recv)
  - High overhead for messages

- **Fortran D compiler**
  - User/tool specifies data distribution
  - Compiler applies “owner computes” rule
  - Identifies non-local accesses
    - distribution function $\mu(i) = (\delta(i), \alpha(i)) = (p,j)$
    - local iteration sets $\text{iter}(X) = g^{-1}(\text{image}(X)) \cap [l:m]$  
    - non-local index sets $\text{index}(Y) = h(\text{iter}(Y) - \text{iter}(X))$
    - Distribute $x,y::(\text{block})$
      
      ```
      DO k = l to m
      X[g(k)] = Y[h(k)]
      ```
      
      ```
      -   
      ```

CTHPC 2003
Communication Analysis

- Represent data using linear inequalities

```
RX[N,N]
DO j = m-1,1,-1
  DO i= 1,n
    RX1(i,j) =
    ... RX2(i,j+1)
```

(Array intersection Loops intersection Comp Data intersection LHS intersection RHS intersection (p ≠ p'))

Data: $b \times np = n$

$\mathbf{b} \times p \leq \mathbf{rx1} \leq \mathbf{b} \times (p+1)$

(Kernel from Tomcatv)
Communication Optimizations

- **Sources of improvement**
  - Reduce communication overhead
  - Overlap communication with computation
  - Exploit pipelining, reductions

- **Message vectorization**
  - Combines messages in loop
  - Place at deepest loop-carried dependence

\[
\delta_i \quad \text{DO } j = m-1,1,-1 \quad \text{comm } (RX[1:n,j]) \\
\quad \text{DO } i = 1,n \quad RX(i,j) = \quad \text{DO } i = 1,n \quad RX(i,j) = \\
\quad \text{DO } i = 1,n \quad RX(i,j) = \quad \text{DO } i = 1,n \quad RX(i,j) = \\
\quad \text{DO } i = 1,n \quad RX(i,j+1) \quad \text{DO } i = 1,n \quad RX(i,j+1)
\]
Impact of Fortran D Compiler

◆ Prototype Fortran D Compiler (1993)
  - Tested on suite of kernels & programs
  - Within 5-50% of hand-coded MPI programs
  - 4-20x faster than CM Fortran compiler available at time
  - Demonstrated message-passing compiler feasible

◆ High Performance Fortran (HPF)
  - Consortium of academia, labs, industry
  - Inspired by Fortran D & similar projects
  - HPF compilers on almost all supercomputers
  - Recently used for Earth Simulator to win Bell prize at SC02
Shared-Memory Compilers (SUIF)

- **Shared-memory**
  - Simple programming model
  - Hardware support (SGI Origin, Sun Starfire)
  - Software support (TreadMarks, CVM)

- **Synchronization**
  - Guard accesses to shared data
  - Prevents data races between processors
  - Enforce using barriers, locks

- **Idle time**
  - Sequential wait - waiting for next parallel task
  - Load imbalance - waiting for barrier completion
Execution Time Breakdown

- **Average idle time = 37%**

- **Solution - eliminate synchronization at compile-time**
Synchronization Optimizations

- **Key insight**
  - Synchronization *not* needed if no communication

- **Algorithm**
  - Apply compile-time computation partition
  - Enlarge parallel regions (merge)
  - Use communication analysis to eliminate barriers
Impact on Idle Time

- Avg idle time decreased 37%→12% (68% reduction)

![Chart showing impact on idle time with a decrease of 37% to 12%, indicating a 68% reduction.](chart.png)

CTHPC 2003
Transformations for Irregular Codes

- **Reorder data & computation for cache**

  ```
  do i = 1,size
  edge[left[i]] += …
  edge[right[i]] += …
  ```

- **Distribute data & computation to processors**
Parallel Irregular Reductions

- Replicate local buffers (ReplicateBufs)
  - local replicated buffer - entire array replicated
  - work for unused elements in local buffer
  - synchronization overhead

```
do time_step
  init local buffer
  compute on local buf
  mutex_accumulate()
```

\[ \text{X: unused} \]
Parallelizing Reductions: LocalWrite

- Compute and update only for locally-owned global data
  - no synchronization needed

- Inspector - select local edges and cut edges

- Cut-edges - replicate computation

```
inspect(local_edge, cut_edges)
do time_step
  compute for local_edges
  compute for cut_edges
```
Evaluation on 4-Processor DEC Alpha

Average Speedups

<table>
<thead>
<tr>
<th></th>
<th>ORIG</th>
<th>LocalWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIG</td>
<td>1.18</td>
<td>1.48</td>
</tr>
<tr>
<td>GPART</td>
<td>2.63</td>
<td>3.10</td>
</tr>
</tbody>
</table>
Overview

- Motivation
- High-performance computing (HPC)
  - Parallelism
- Exploiting new architectural features
  - Locality
- Improving programmer productivity
  - High-level languages
  - Error-checking
  - Benchmarks
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 $\neq$ 2x clock speed increase
  - 2x clock speed increase $\neq$ 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
Exploiting New Architectural Features

- 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 → 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
  - Caches
  - Vector units
  - Long instruction words
  - Multithreading
  - Predicated instructions

- 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

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

[Mehis+ 2002]
Architectural Features – SSE2 Vector Instructions

ATLAS, 2.2 Ghz Intel Xeon

90%

[Mejis+ 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

◆ **My personal research area**
Exploiting Architectural Features – Locality

- **Locality**
  - Temporal (reuse data)
  - Spatial (reuse nearby data)

- **Caches**
  - Long cache lines
  - Set associativity

- **Conflict misses:**
  - Caused by limited associativity
  - 50% of cache misses [McKinley & Temam: ASPLOS’96]
  - Severe (recurring) misses between references
Data Layout Transformations

- **Padding**
  - Inter-variable - change of base address
    \[
    \text{real } A(N), \text{ B}(N) \rightarrow \text{real } A(N), \text{ DUM(PAD)}, \text{ B}(N)
    \]
  - Intra-variable - change of array dimension
    \[
    \text{real } A(N,N) \rightarrow \text{real } A(N+\text{PAD}, N)
    \]

- **PAD algorithm**
  - analyze program, predict severe misses
  - find references with consistent access patterns
    \[
    \text{distance}(A[i, j-1], B[i+1, j+1]) = |BaseA - BaseB + \delta|
    \]
  - pad variables if distance in cache < 1 cache line
Impact of PAD on Execution Time

- Fewer misses → execution time improvements

![Graph showing speedup for different programs on Alpha 21064, UltraSparc2, and Pentium2.]

% Speedup

adi32  |  chol256  |  dgefa256  |  dot512  |  erle64  |  expl512  |  jacobi512  |  shal512  |  appsp  |  swim  |  tomcatv

- Alpha 21064
- UltraSparc2
- Pentium2

60%
Tiling Optimizations

- Move reuses closer in time
- Example: matrix multiplication

![Code Snippet]

- Tile data should fit in cache
Avoiding Conflict Misses

- Need to spread out tile columns on cache

![](image1.png)

- Targets
  - 2D linear algebra
  - 3D stencil codes

```
  do k=kk,kk+T
    do j=jj,jj+T
      do i=ii,ii+T
        A[i,j,k] = B[i±1,j±1,k±1]
  
```
Impact of Tiling on Linear Algebra Codes

- Tile size selection improves performance
- Padding avoids pathological problem sizes

![Graph showing performance improvement with tiling]

- 75% performance improvement

CTHPC 2003
Impact of Tiling 3D Stencils

- Tile size selection improves performance
- Padding improves stability

![Graph showing impact of tiling on performance](image)

- **Orig**
- **Tile**
- **Pad**

25% increase in performance
Overview

◆ Motivation

◆ High-performance computing (HPC)
  - Parallelism

◆ Exploiting new architectural features
  - Locality

◆ Improving programmer productivity
  - High-level languages
  - Error-checking
  - Benchmarks
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

- **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

◆ 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 – High-Level Languages

- **Compiler optimizations help**
  - Reduce penalty for high-level language constructs
    - Type safety
    - Objects
    - Inheritance
    - Abstract data types
    - Parametric polymorphism
    - Exceptions
    - Tagged unions
    - Garbage collection
    - Higher-order functions
    - Parameterized typedefs
  - Many of these features are already in Java compilers

- **Result → cleaner, high-level code**
High-Level Languages – Intel cc vs GNU gcc

SPECint 2000, 2.2 Ghz Intel Xeon

C++
272%

Avg
29%

*252.eon is C++ source code

[Mehis+ 2002]
Improving Productivity – Safety Checks

◆ People avoid built-in safety checks for performance
  - Examples
    ● Array bounds / buffer overrun check
    ● Pointer range, dangling pointer check
    ● Run-time type check
    ● Garbage collection

◆ Problem if error encountered
  - Incorrect results, crashes / freezes (blue screen of death)
  - Viruses, worms (for network software)

◆ Compiler optimizations help
  - Reduce penalty for error checking / security

◆ Result → safer, more stable code
Safety Checks – Case Study: Cyclone

- **Cyclone**
  - Safe C-based programming language
  - Rules out programs that have buffer overflows, dangling pointers, format string attacks, etc…

- **Provides safety while remaining compatible with C**
  - Enforces type safety (only safe casts)
  - Same data representation & calling conventions
  - Region-based, manual memory management
  - Using a combination of type information and run-time checks to prevent array-bound violations
  - Wrapping the C standard library with appropriate run-time checks as necessary (e.g., has a FILE already been closed)
Safety Checks – Cyclone Overhead vs. C

![Bar graph showing slowdown percentages for various applications. The graph indicates that grobner has a slowdown of 185% compared to C, while tile has a slowdown of 38%.]
Safety Checks – Cyclone

- **Cyclone**
  - Overhead ranges from 0 to 3x slowdown
  - Compiler optimizations improve performance
  - In some cases can be critical

- **Fat pointers**
  - Pointer represented as 3 words
    - Base address / bound address / current pointer
    - Allows run-time safety check on pointers
  - Pointer copy
    - gcc → block move with movsl
    - gcc –march=i686 → register copies
  - On Pentium III, 17x slowdown for copy, 2x slowdown overall
Improving Productivity – User Feedback

◆ People write programs with poor style
  - Examples
    ● Pointer aliases / branches limit instruction-level parallelism
    ● Control /data dependences restrict task / loop parallelism
    ● Data allocation / access patterns reduce locality

◆ Problem
  - Significantly reduced program performance (in some cases)

◆ Compiler analysis can identify source of problem
  - Provide user feedback
  - Train people to code in “better” (i.e., compiler-analyzable) style
  - Allow users to interactively provide additional information

◆ Result → cleaner, more efficient code
Benchmarking Performance

◆ **Current benchmarks**
  - Designed to measure best processor + compiler performance
  - Do not measure impact of compiler optimizations on improving programmer productivity

◆ **Most benchmark programs**
  - Use low-level programming languages (Fortran, C)
  - Perform no error-checking / security checks
  - Hand-tuned to death
  - Use poor programming style (dusty deck)
  - Disallow user modifications (based on compiler feedback)

◆ **Example**
  - SPEC95, Perfect, Linpack, etc…
Benchmarking Performance

◆ Current benchmark design is understandable
  - Easy to measure performance improvements
  - Useful for comparing processor architectures

◆ Much harder to objectively measure optimizations designed to improve productivity

◆ Some movement to high-level language benchmarks
  - NAS 1.0 (paper & pencil)
  - SPEC CPU 2000 (1 C++, 4 Fortran90 out of 26 codes)
  - SPEC JVM 98 / JBB 2000 (Java client / server codes)

◆ Can we design better benchmarks?
Next Generation Benchmarks

◆ To measure the impact of compilers on productivity
  - Benchmarks should be code that is
    ● Easy to understand
    ● Easy to reuse, composed from libraries
    ● Close to natural description of high-level algorithm
    ● Written in high-level, easy to maintain style
  - Benchmarks should also
    ● Test support of high-level programming language features
    ● Require safety / security checks to be turned on
    ● Allow possibility of improvement through user feedback
◆ Unfortunately, probably not anytime soon…
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
As if People Programmed

- Do compiler research, assuming programmers
  - Care about writing their program correctly and quickly
  - Also care about the performance
    - And are willing to fix / improve algorithms
  - Would happily interact with compiler / tools
    - If it was useful
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?

- Writing software is hard…
  - Improving the way software is written is harder

- But beats 4% performance improvement / year!