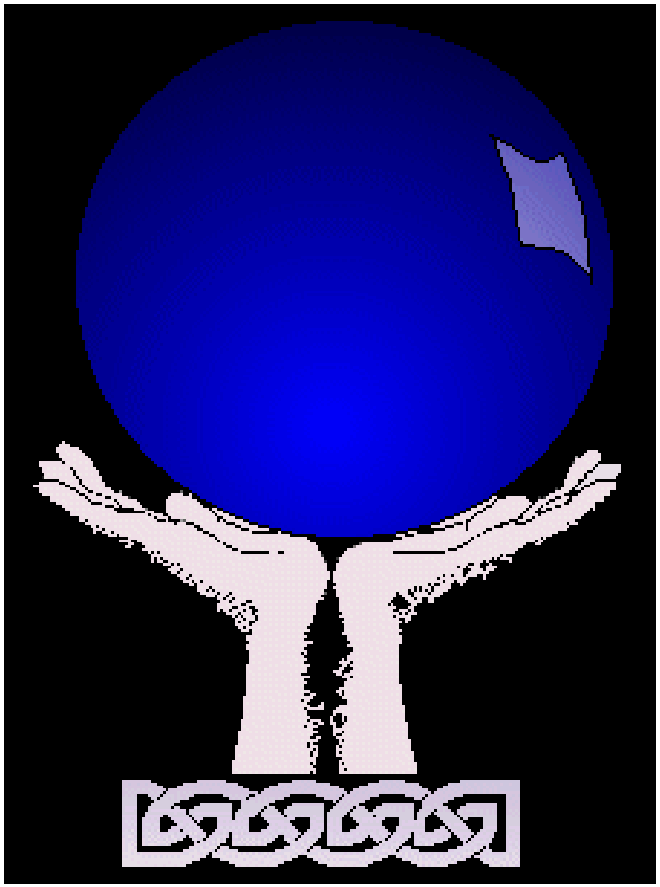
6.189 IAP 2007

Lecture 18

The Future

Predicting the Future is Always Risky

- "I think there is a world market for maybe five computers.“
  - Thomas Watson, chairman of IBM, 1949
- "There is no reason in the world anyone would want a computer in their home. No reason.”
  - Ken Olsen, Chairman, DEC, 1977
- "640K of RAM ought to be enough for anybody.”
  - Bill Gates, 1981



Future = Evolution + Revolution

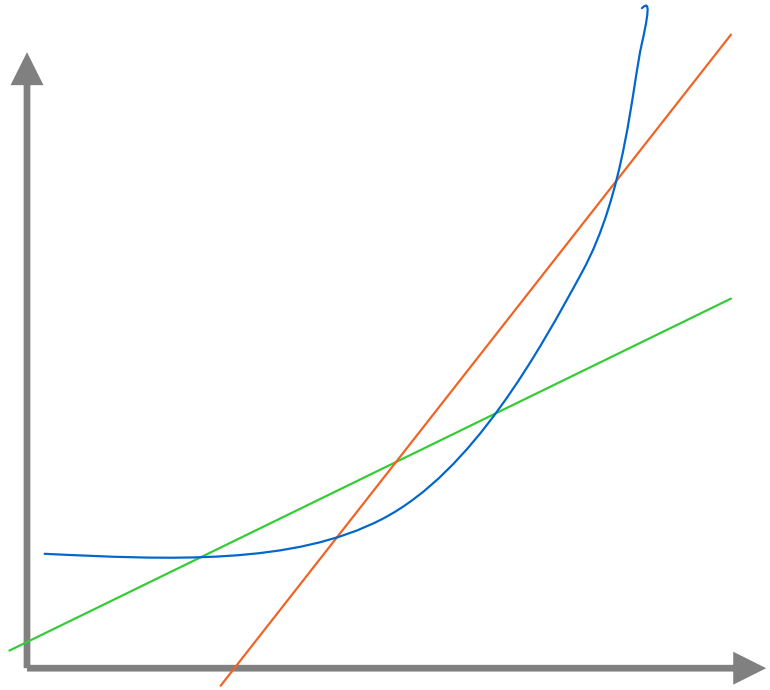
- Evolution
  - Relatively easy to predict
  - Extrapolate the trends
- Revolution
  - A completely new technology or solution
  - Hard to Predict
- Paradigm Shifts can occur in both

Outline

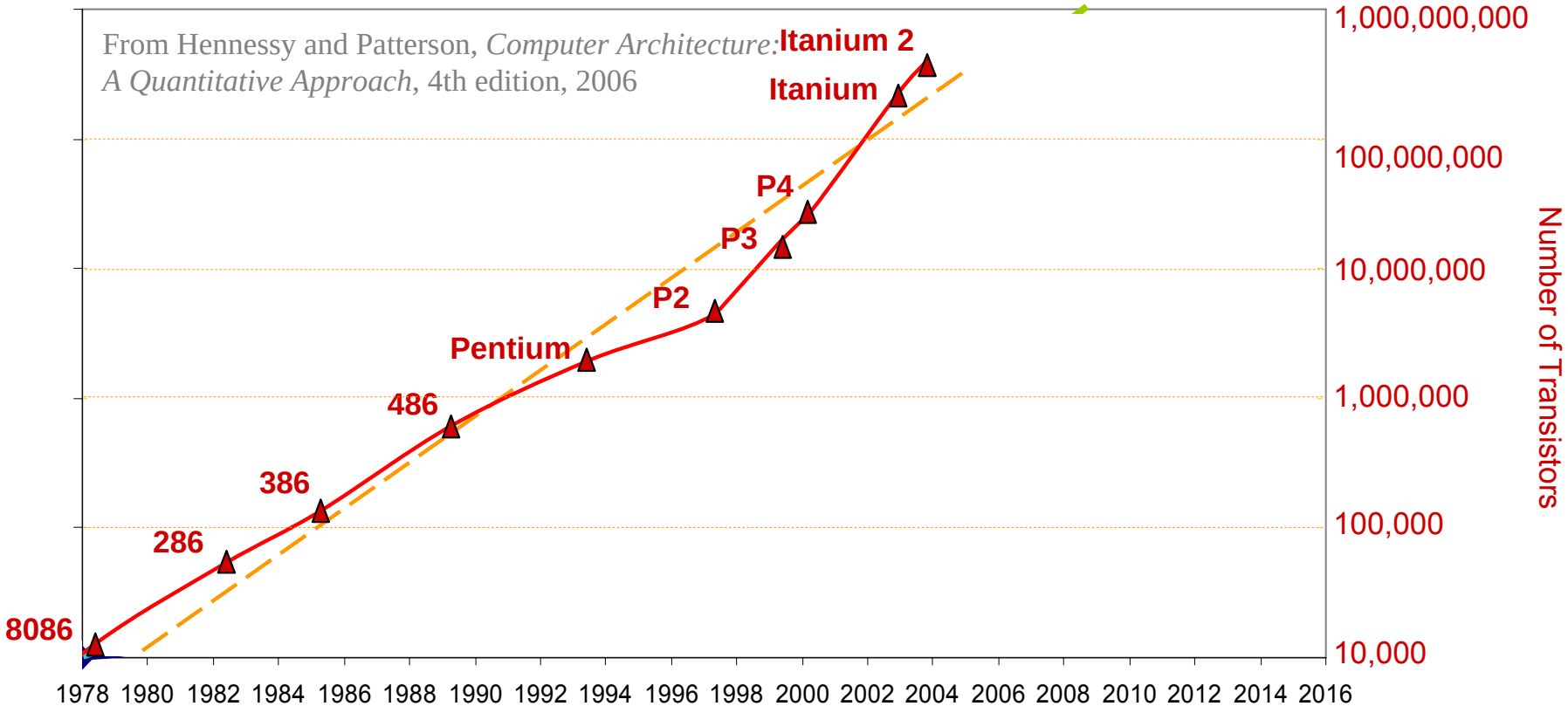
- Evolution
  - **Trends**
  - Architecture
  - Languages, Compilers and Tools
- Revolution
- Crossing the Abstraction Boundaries

Evolution

- Look at the trends
  - Moore's Law
  - Power Consumption
  - Wire Delay
  - Hardware Complexity
  - Parallelizing Compilers
  - Program Design Methodologies
- Design Drivers are different in Different Generations

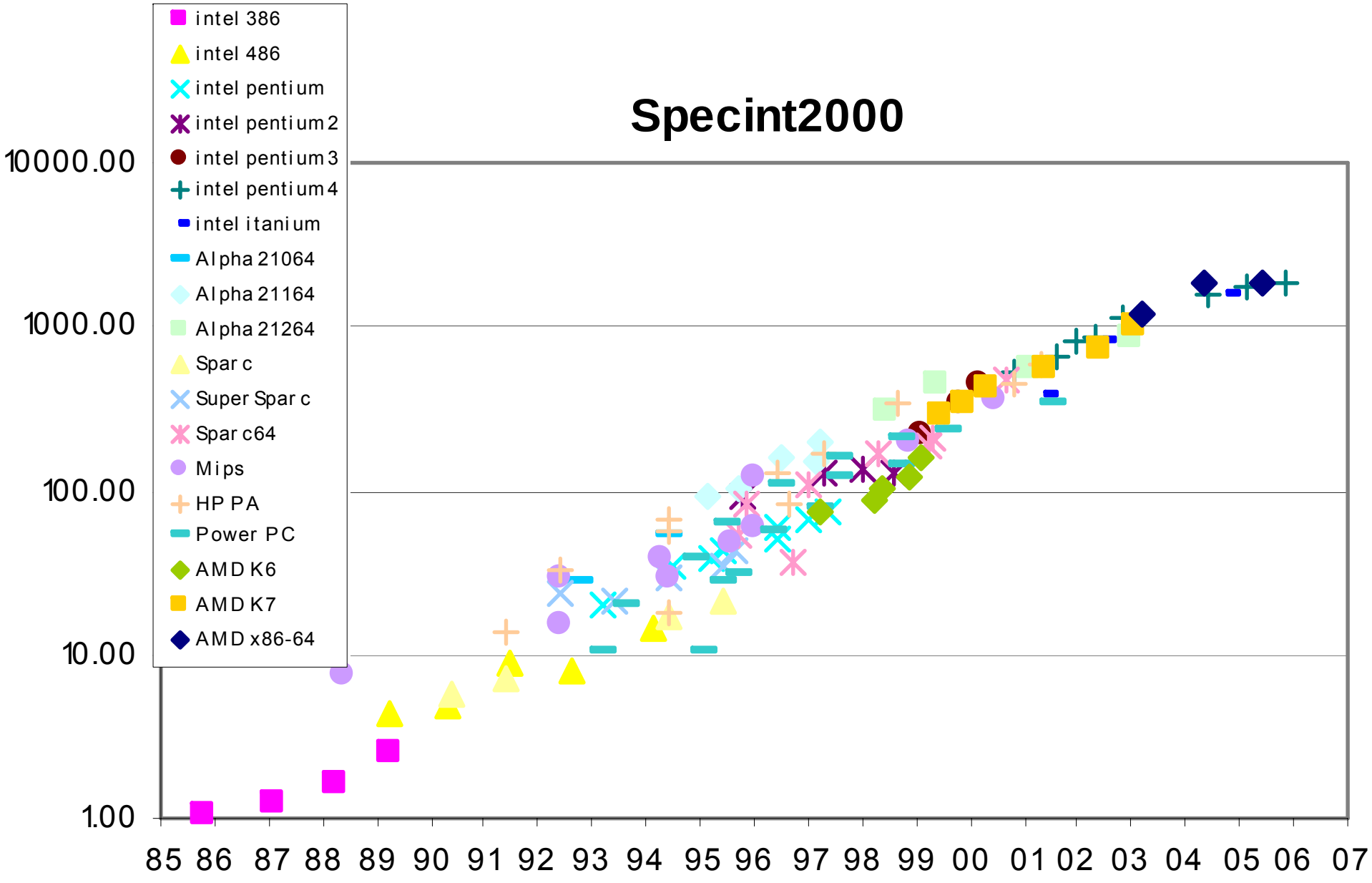


The March to Multicore: Moore's Law

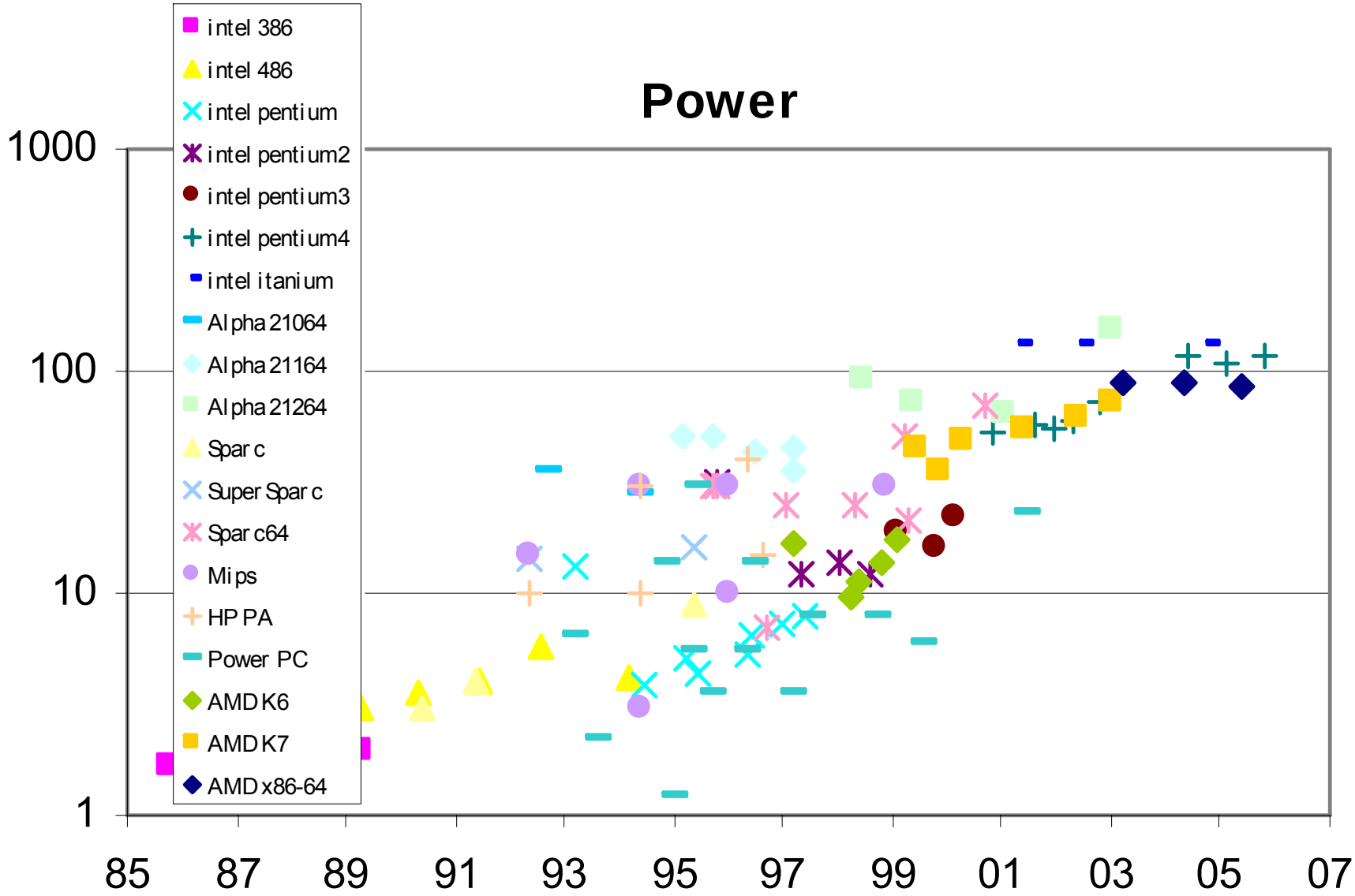


From David Patterson

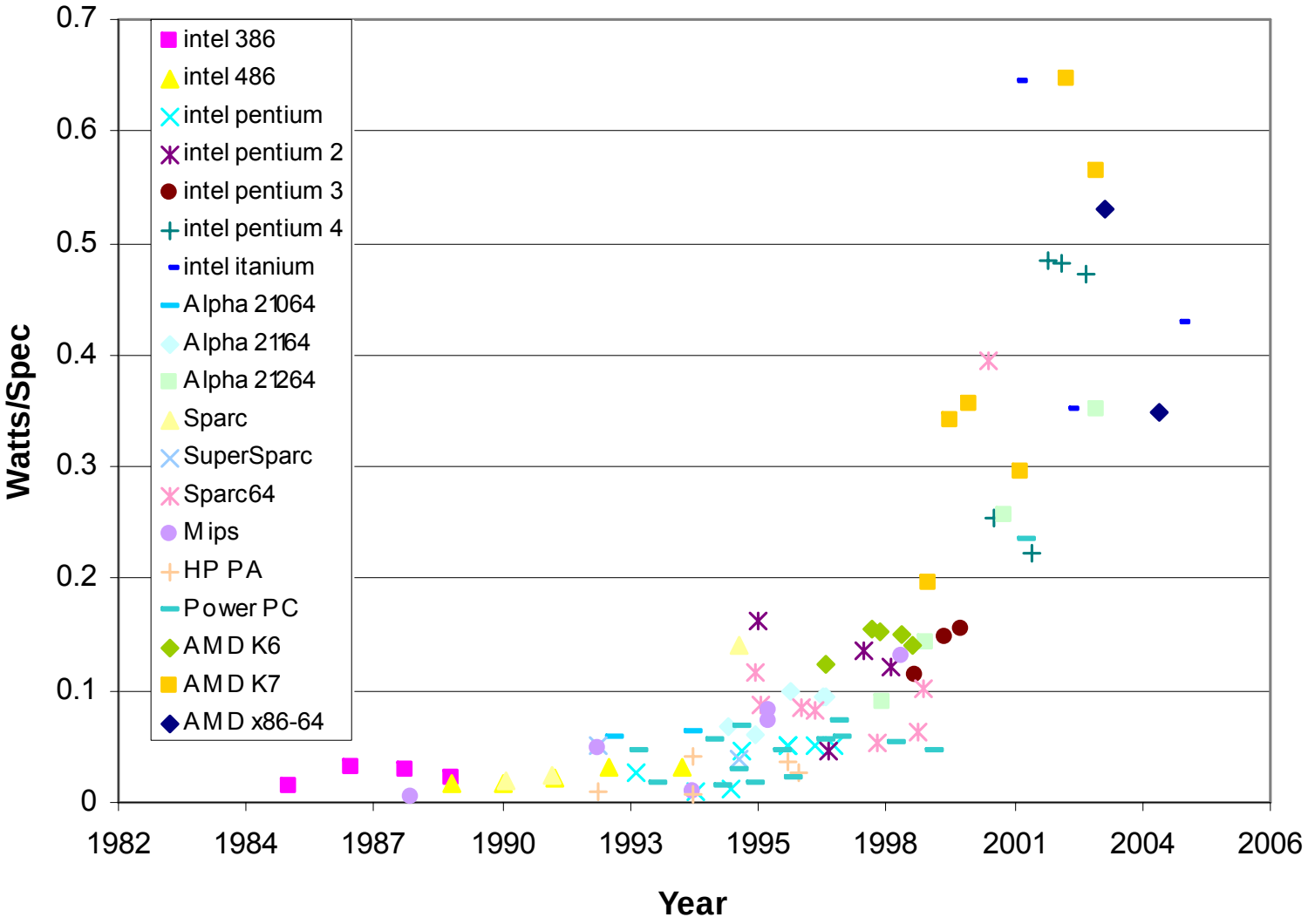
The March to Multicore: Uniprocessor Performance (SPECint)



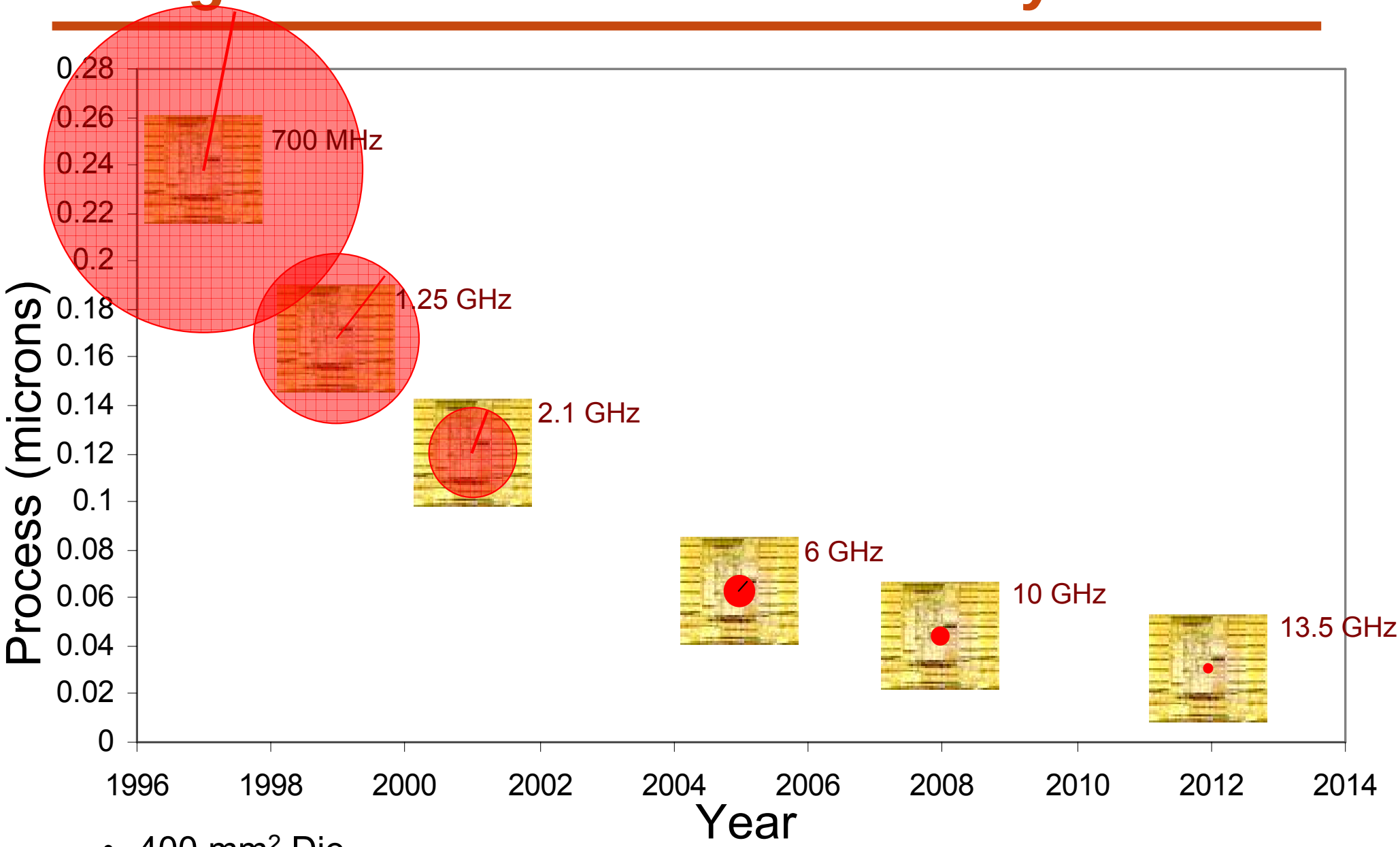
Power Consumption (watts)



Power Efficiency (watts/spec)

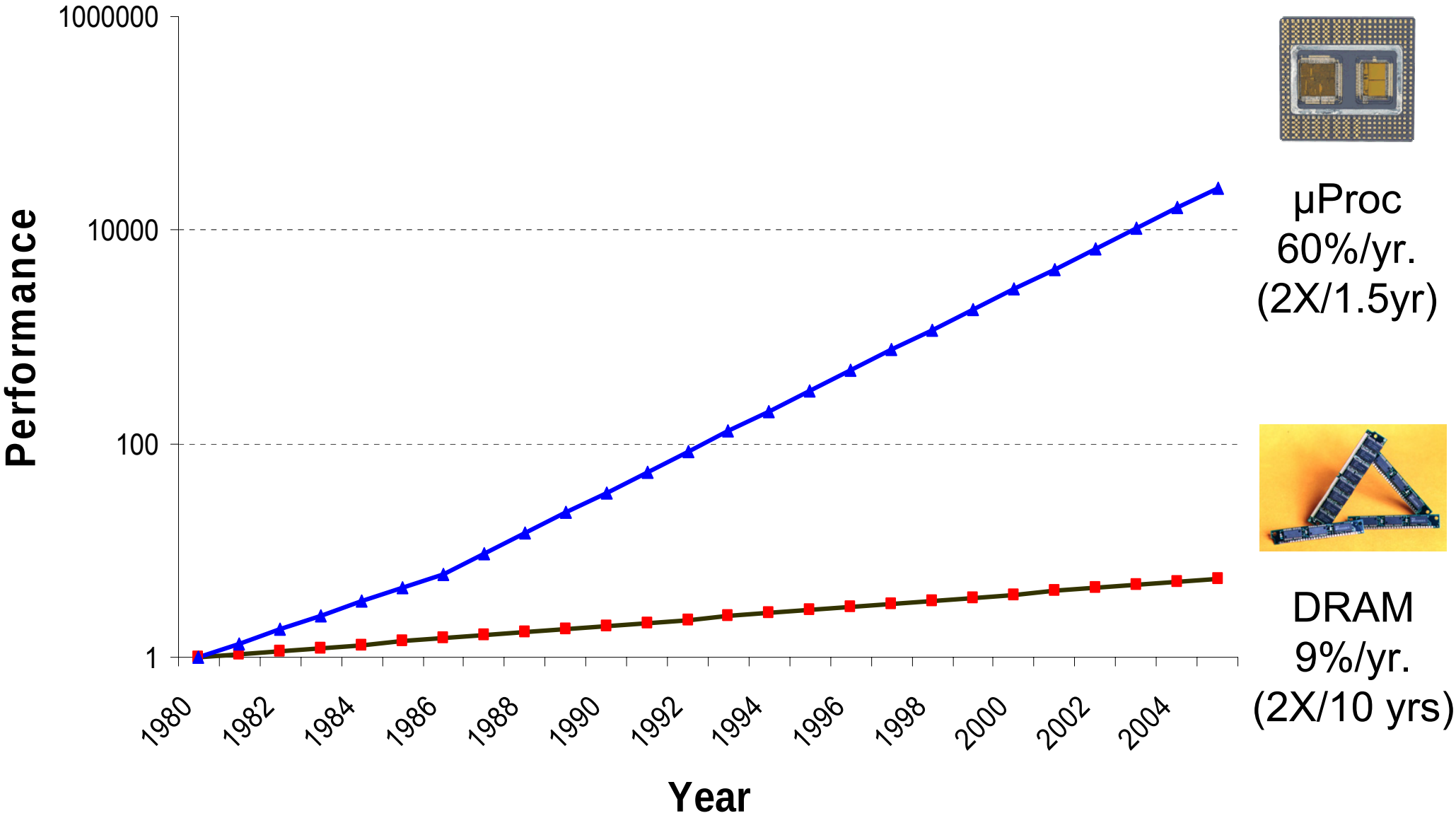


Range of a Wire in One Clock Cycle



- 400 mm^2 Die
- From the SIA Roadmap

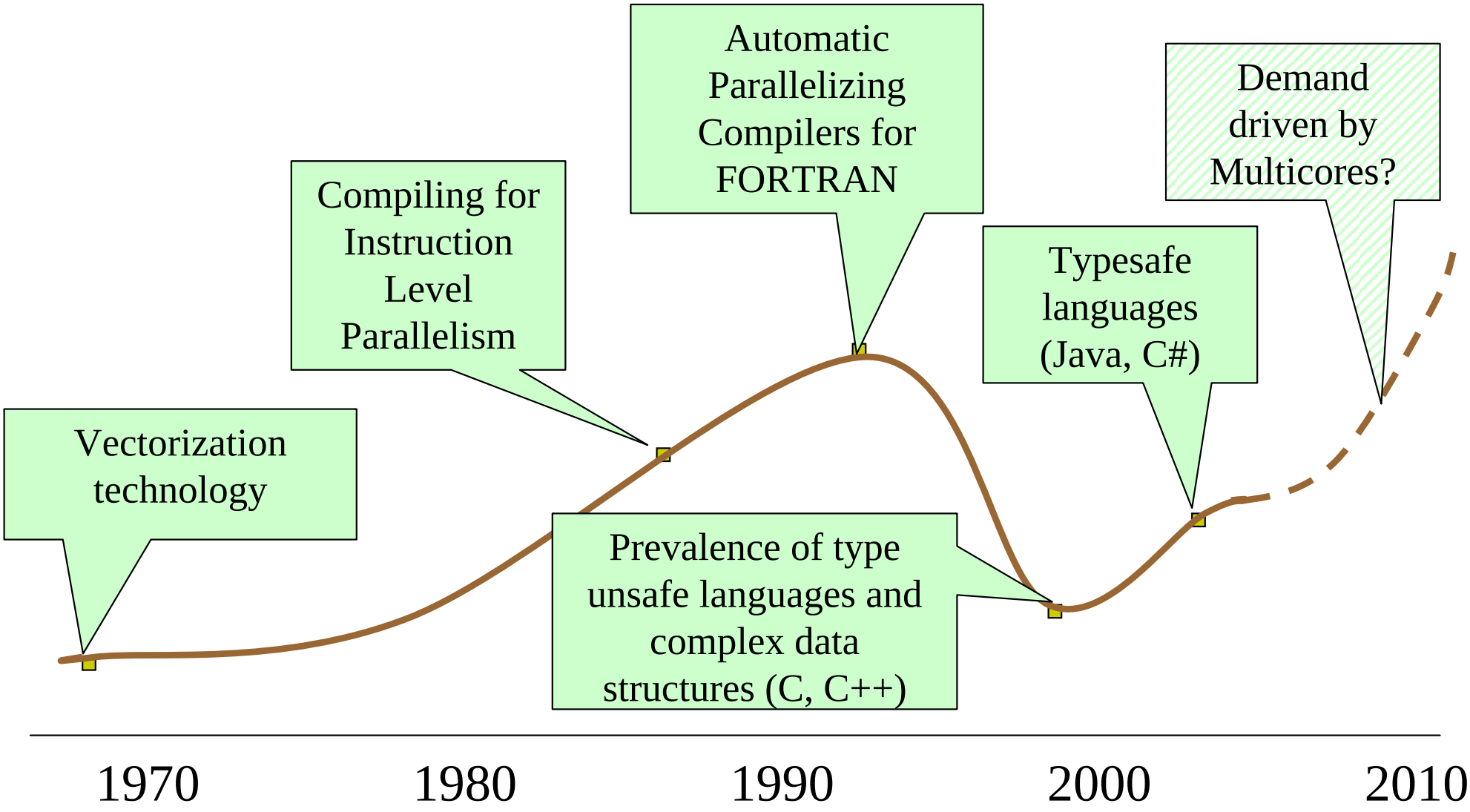
DRAM Access Latency



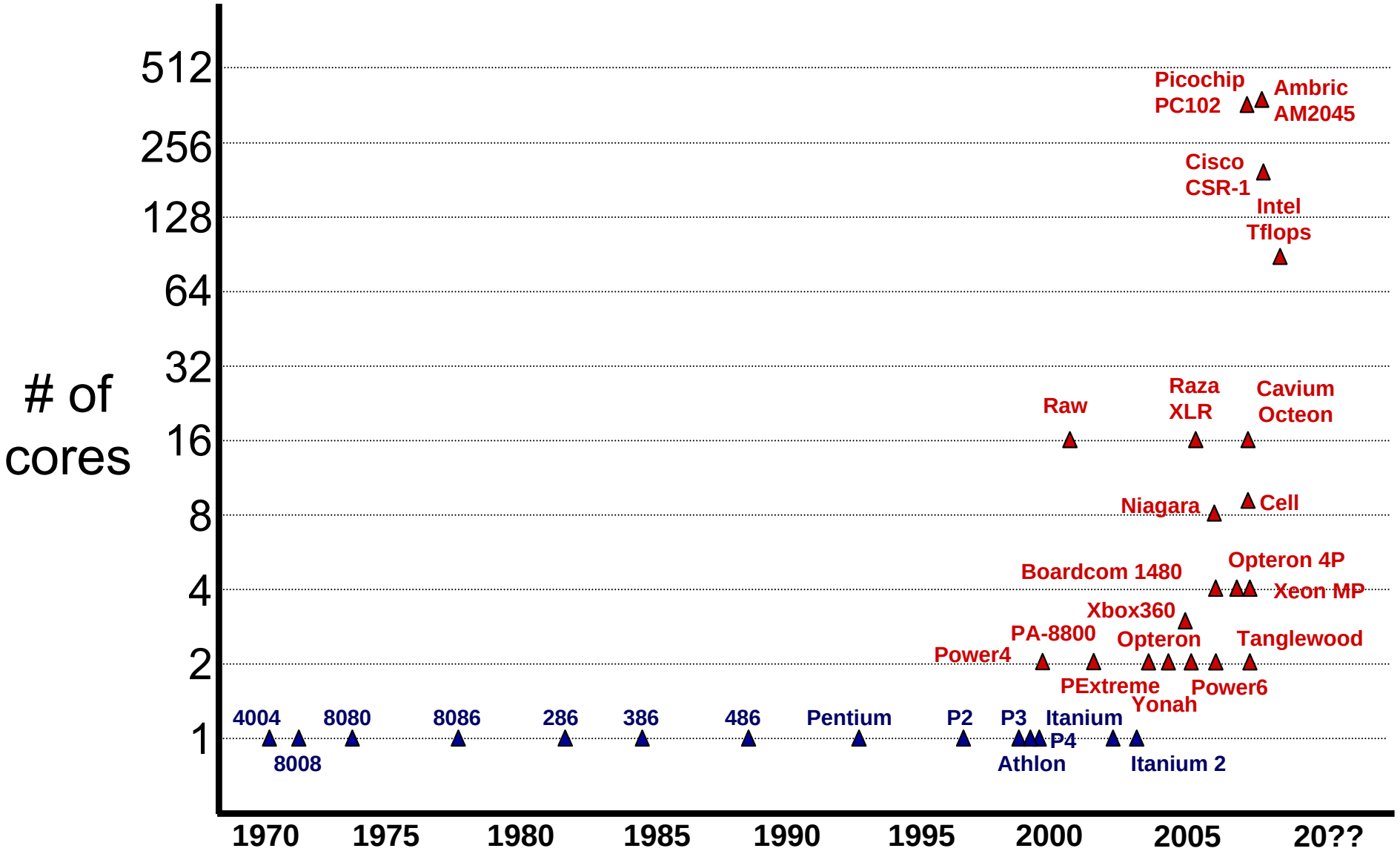
µProc
60%/yr.
(2X/1.5yr)

DRAM
9%/yr.
(2X/10 yrs)

Improvement in Automatic Parallelization



Multicores are here



Outline

- Evolution
  - Trends
  - **Architecture**
  - Languages, Compilers and Tools
- Revolution
- Crossing the Abstraction Boundaries

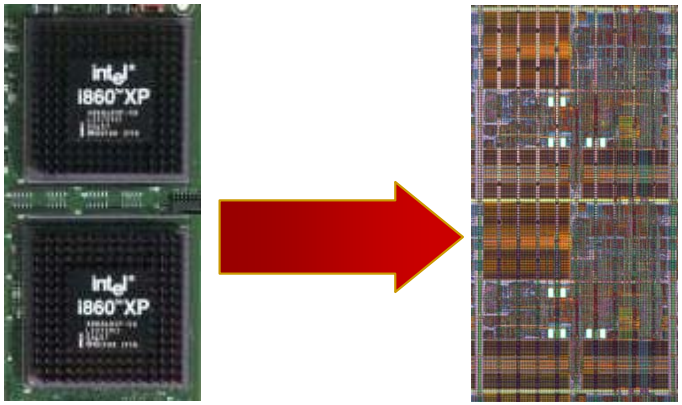
Novel Opportunities in Multicores

- Don't have to contend with uniprocessors
  - The era of Moore's Law induced performance gains is over!
  - Parallel programming will be required by the masses
    - not just a few supercomputer super-users

Novel Opportunities in Multicores

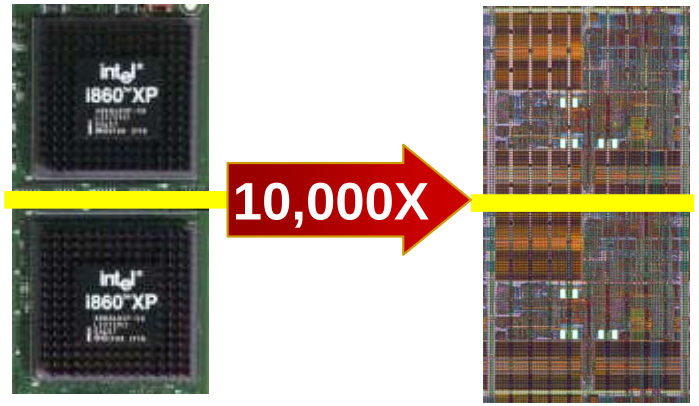
- Don't have to contend with uniprocessors
  - The era of Moore's Law induced performance gains is over!
  - Parallel programming will be required by the masses
    - not just a few supercomputer super-users

- Not your same old multiprocessor problem
  - How does going from Multiprocessors to Multicores impact programs?
  - What changed?
  - Where is the Impact?
    - Communication Bandwidth
    - Communication Latency



Communication Bandwidth

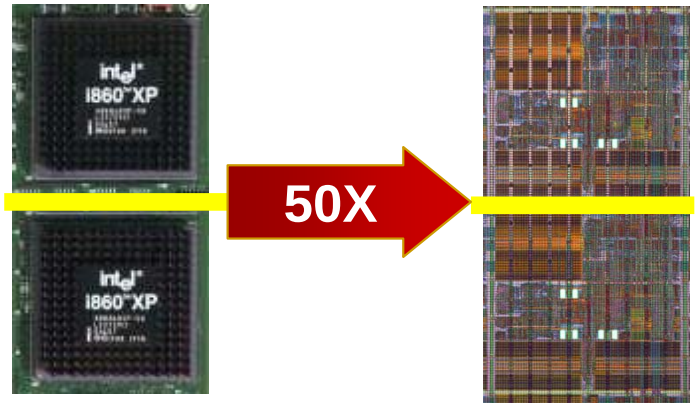
- How much data can be communicated between two cores?
- What changed?
  - Number of Wires
    - IO is the true bottleneck
    - On-chip wire density is very high
  - Clock rate
    - IO is slower than on-chip
  - Multiplexing
    - No sharing of pins
- Impact on programming model?
  - Massive data exchange is possible
  - Data movement is not the bottleneck → processor affinity not that important



32 Giga bits/sec ~300 Tera bits/sec

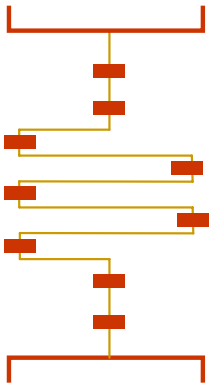
Communication Latency

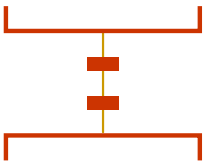
- How long does it take for a round trip communication?
- What changed?
  - Length of wire
    - Very short wires are faster
  - Pipeline stages
    - No multiplexing
    - On-chip is much closer
    - Bypass and Speculation?
- Impact on programming model?
  - Ultra-fast synchronization
  - Can run real-time apps on multiple cores



~200 Cycles

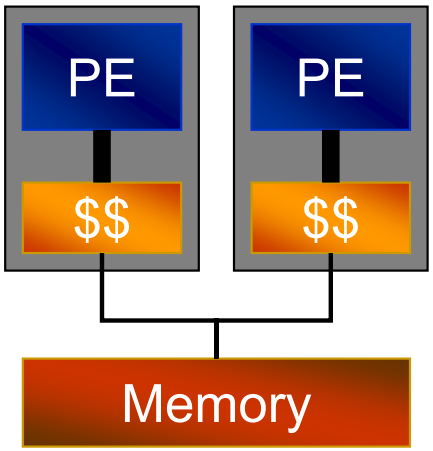
~4 cycles



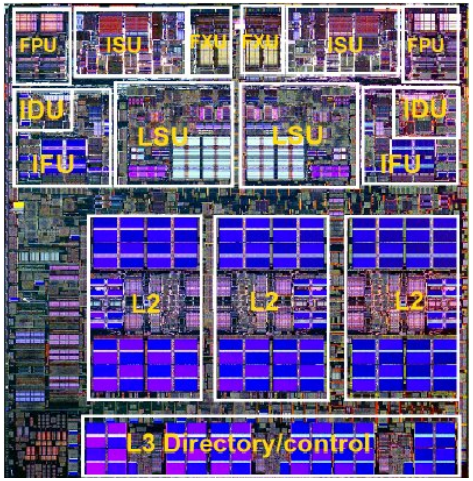


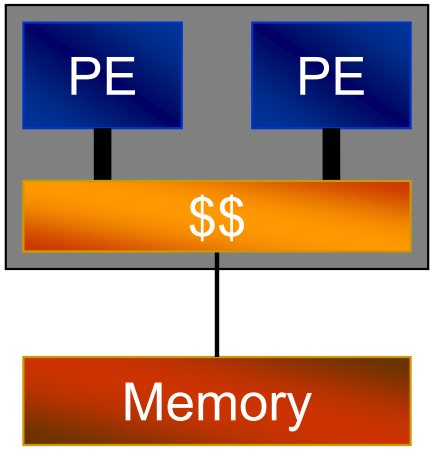
Past, Present and the *Future?*

Traditional Multiprocessor

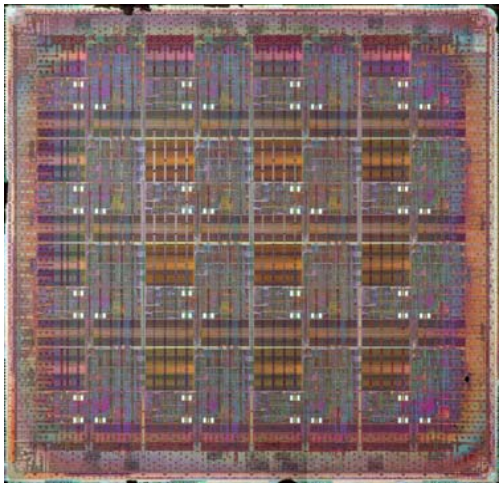


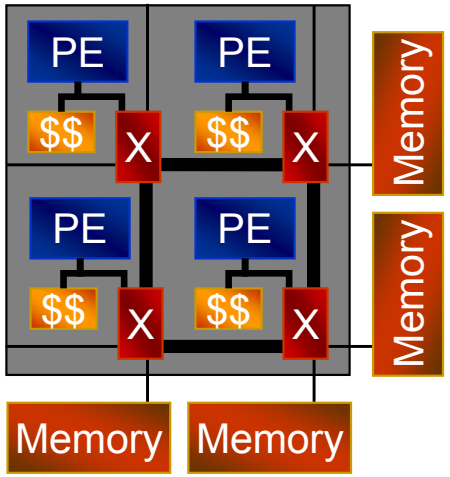
Basic Multicore
IBM Power5





Integrated Multicore
16 Tile MIT Raw





Outline

- Evolution
  - Trends
  - Architecture
  - **Languages, Compilers and Tools**
- Revolution
- Crossing the Abstraction Boundaries

The OO Revolution

- Object Oriented revolution did not come out of a vacuum
- Hundreds of small experimental languages
- Rely on lessons learned from lesser-known languages
  - C++ grew out of C, Simula, and other languages
  - Java grew out of C++, Eiffel, SmallTalk, Objective C, and Cedar/Mesa[1]
- Depend on results from research community

[1] *J. Gosling, H. McGilton, The Java Language Environment*

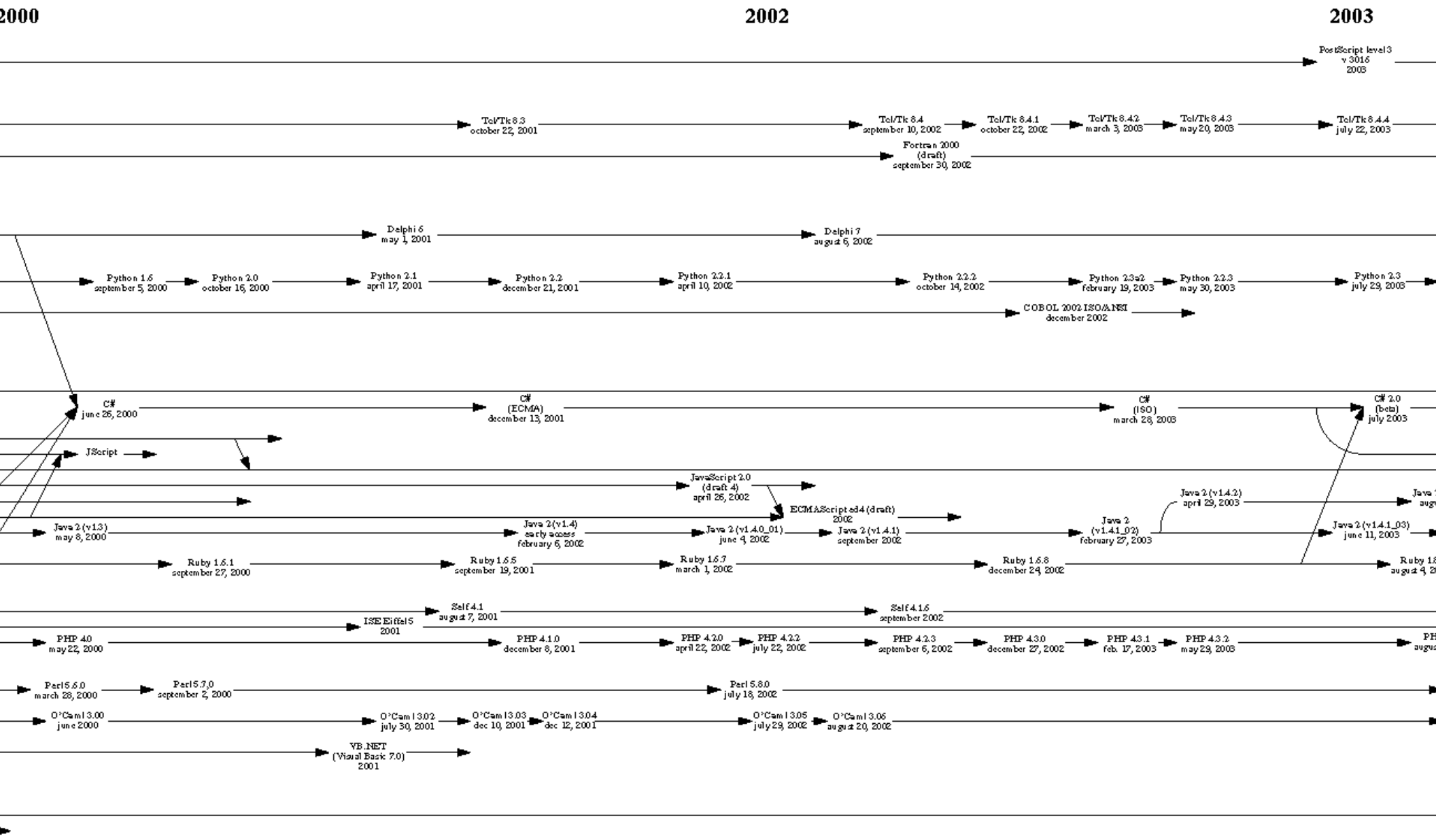
Object Oriented Languages

- Ada 95
- BETA
- Boo
- C++
- C#
- ColdFusion
- Common Lisp
- COOL (Object Oriented COBOL)
- CorbaScript
- Clarion
- Corn
- D
- Dylan
- Eiffel
- F-Script
- Fortran 2003
- Gambas
- Graphtalk
- IDLscript
- incr Tcl
- J
- JADE
- Java
- Lasso
- Lava
- Lexico
- Lingo
- Modula-2
- Modula-3
- Moto
- Nemerle
- Nuva
- NetRexx
- Nuva
- Oberon (Oberon-1)
- Object REXX
- Objective-C
- Objective Caml
- Object Pascal (Delphi)
- Oz
- Perl 5
- PHP
- Pliant
- PRM
- PowerBuilder
- ABCL
- Python
- REALbasic
- Revolution
- Ruby
- Scala
- Simula
- Smalltalk
- Self
- Squeak
- Squirrel
- STOOP (Tcl extension)
- Superx++
- TADS
- Ubercode
- Visual Basic
- Visual FoxPro
- Visual Prolog
- Tcl
- ZZT-oop

Source: Wikipedia

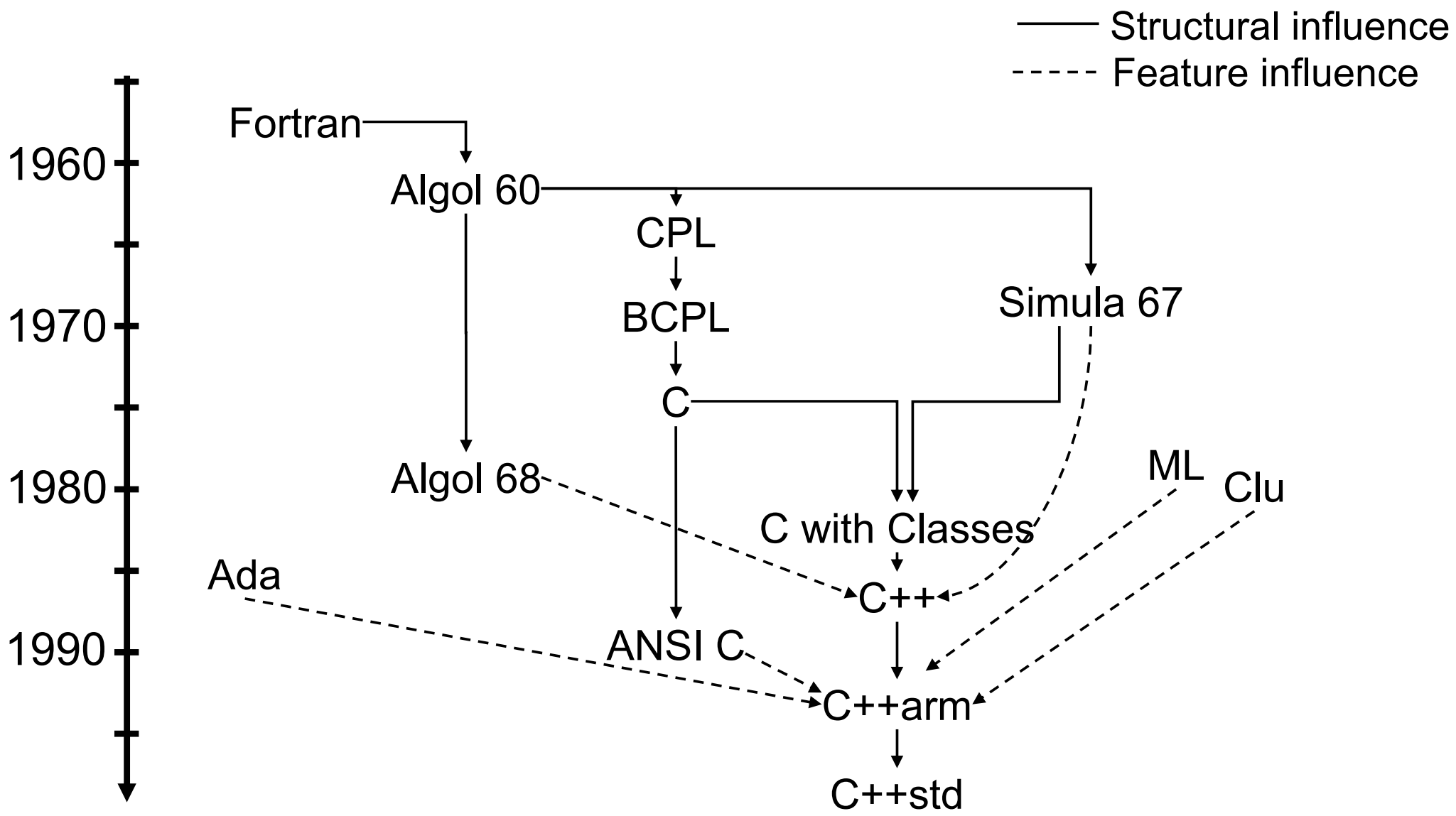
Language Evolution

From FORTRAN to a few present day languages



Source: Eric Levenez

Origins of C++



Source: *B. Stroustrup, The Design and Evolution of C++*

Academic Influence on C++

"Exceptions were considered in the original design of C++, but were postponed because there wasn't time to do a thorough job of exploring the design and implementation issues.

...

In retrospect, the greatest influence on the C++ exception handling design was the work on fault-tolerant systems started at the University of Newcastle in England by Brian Randell and his colleagues and continued in many places since."

-- B. Stroustrup, A History of C++

Origins of Java

- Java grew out of C++, Eiffel, SmallTalk, Objective C, and Cedar/Mesa
- Example lessons learned:
  - Stumbling blocks of C++ removed (multiple inheritance, preprocessing, operator overloading, automatic coercion, etc.)
  - Pointers removed based on studies of bug injection
  - GOTO removed based on studies of usage patterns
  - Objects based on Eiffel, SmallTalk
  - Java interfaces based on Objective C protocols
  - Synchronization follows monitor and condition variable paradigm (introduced by Hoare, implemented in Cedar/Mesa)
  - Bytecode approach validated as early as UCSD P-System ('70s)

→ Lesser-known precursors essential to Java's success

Source: *J. Gosling, H. McGilton, The Java Language Environment*

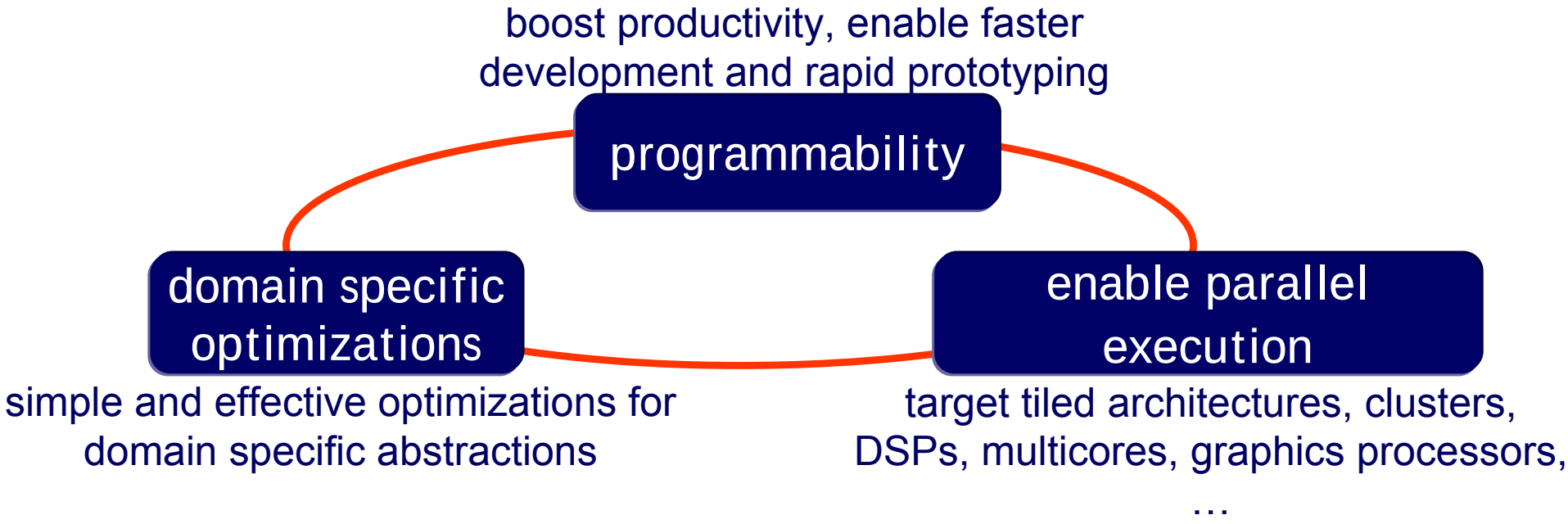
Why New Programming Models and Languages?

- Paradigm shift in architecture
  - From sequential to multicore
  - Need a new “common machine language”
- New application domains
  - Streaming
  - Scripting
  - Event-driven (real-time)
- New hardware features
  - Transactions
  - Introspection
  - Scalar Operand Networks or Core-to-core DMA
- New customers
  - Mobile devices
  - The average programmer!
- Can we achieve parallelism without burdening the programmer?

Domain Specific Languages

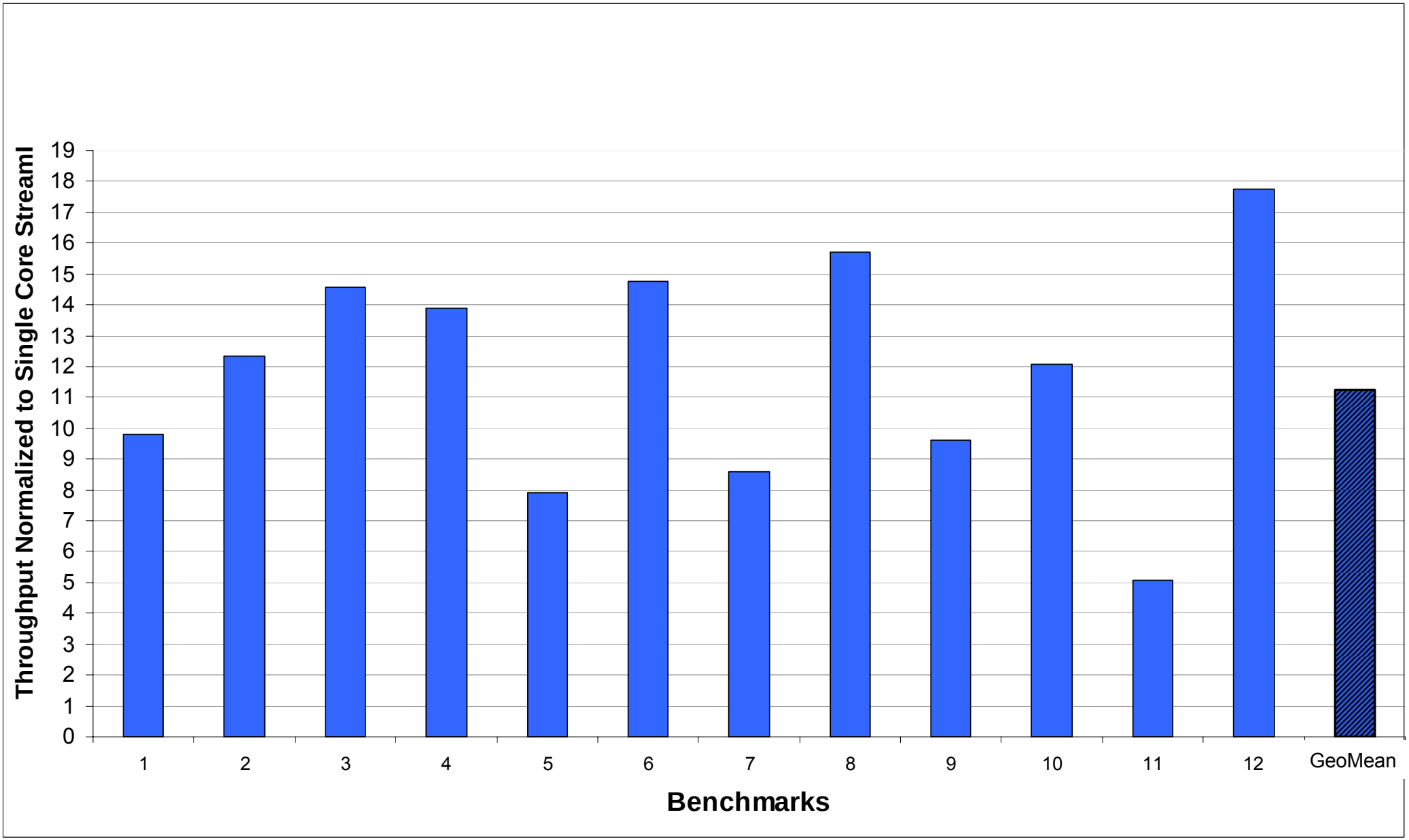
- There is no single programming domain!
  - Many programs don't fit the OO model (ex: scripting and streaming)
- Need to identify new programming models/domains
  - Develop domain specific end-to-end systems
  - Develop languages, tools, applications $\Rightarrow$ a body of knowledge
- Stitching multiple domains together is a hard problem
  - A central concept in one domain may not exist in another
    - Shared memory is critical for transactions, but not available in streaming
  - Need conceptually simple and formally rigorous interfaces
  - Need integrated tools
  - But critical for many applications

Compiler-Aware Language Design: StreamIt Experience



- Some programming models are inherently concurrent
  - Coding them using a sequential language is…
    - Harder than using the right parallel abstraction
    - All information on inherent parallelism is lost
- There are win-win situations
  - Increasing the programmer productivity while extracting parallel performance

StreamIt Performance on Raw



Parallelizing Compilers: SUIF Experience

- Automatic Parallelism is not impossible
  - Can work well in many domains (example: ILP)
- Automatic Parallelism for multiprocessors “almost” worked in the ‘90s
  - SUIF compiler got the Best SPEC results by automatic parallelization
- But…
  - The compilers were not robust
  - Clients were impossible (performance at any cost)
  - Multiprocessor communication was expensive
  - Had to compete with improvements in sequential performance
  - The Dogfooding problem
- Today: Programs are even harder to analyze
  - Complex data structures
  - Complex control flow
  - Complex build process
  - Aliasing problem (type unsafe languages)

Compilers

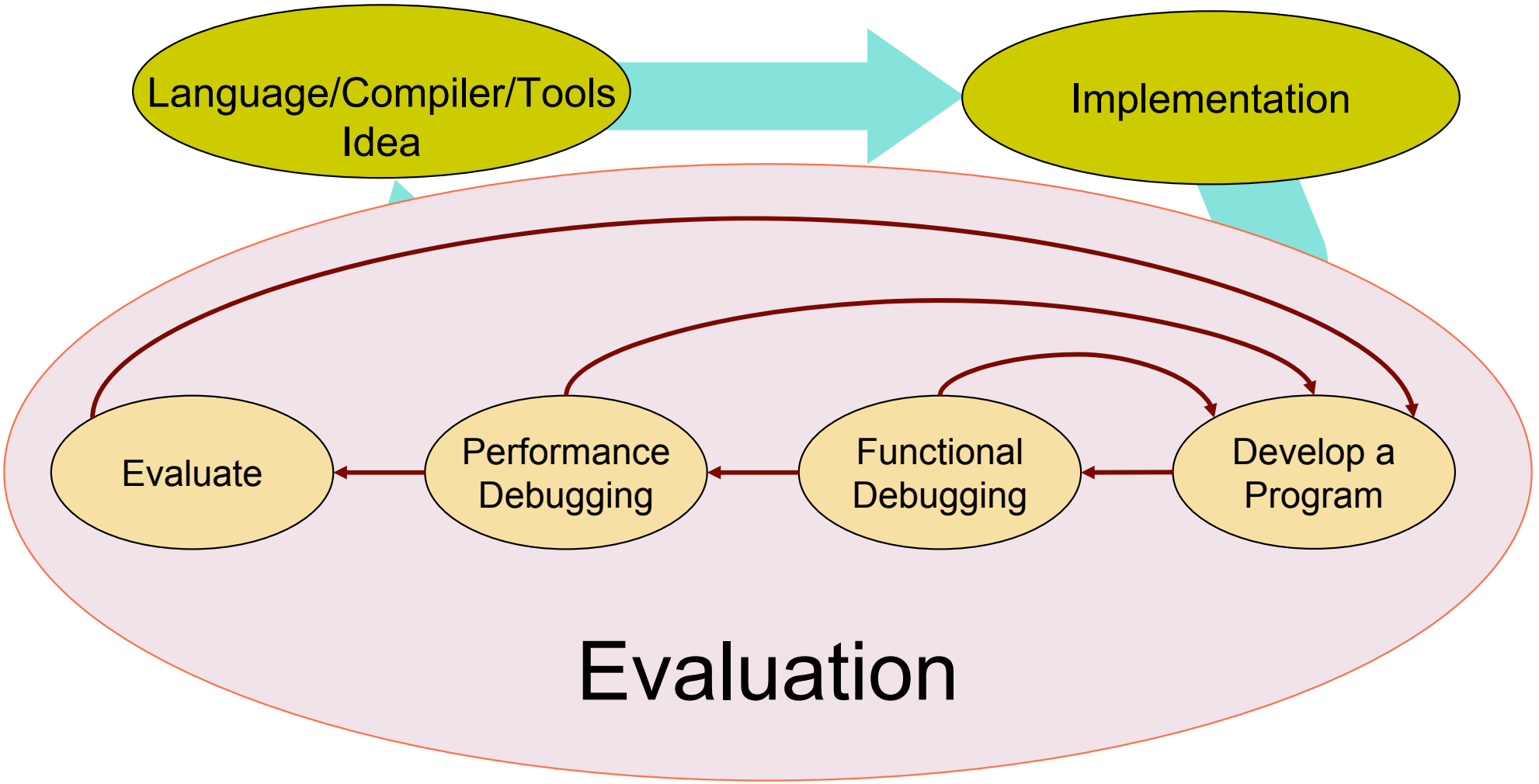
- Compilers are critical in reducing the burden on programmers
  - Identification of data parallel loops can be easily automated, but many current systems (Brook, PeakStream) require the programmer to do it.

- Need to revive the push for automatic parallelization
  - Best case: totally automated parallelization hidden from the user
  - Worst case: simplify the task of the programmer

Tools

- A lot of progress in tools to improve programmer productivity
- Need tools to
  - Identify parallelism
  - Debug parallel code
  - Update and maintain parallel code
  - Stitch multiple domains together
- Need an “Eclipse platform for multicores”

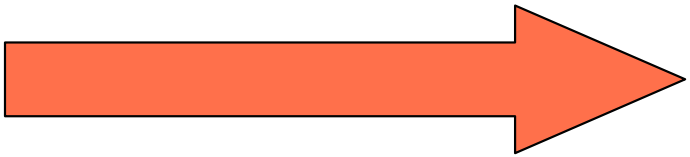
Facilitate Evaluation and Feedback for Rapid Evolution



The Dogfooding Problem
CAD Tools vs. OO Languages

- CAD Tools
  - Universally hated by the users
  - Only a few can hack it
  - Very painful to use
- Origins
  - Developed by CAD experts
  - User community is separate
- High Performance Languages
  - User community is separate
  - Hard to get feedback
  - Slow evolution

- Object Oriented Languages
  - User friendly
  - Universal acceptance
  - Use by ordinary programmers
  - Huge improvements in programmer productivity
- Origins
  - Developed by PL experts
  - The compiler is always written using the language/tools
  - Rapid feedback



Rapid Evaluation

- Extremely hard to get
  - Real users have no interest in flaky tools
  - Hard to quantify
  - Superficial users vs. Deep users will give different feedback
    - Fatal flaws as well as amazing uses may not come out immediately
- Need a huge, sophisticated (and expensive) infrastructure
  - How to get a lot of application experts to use the system?
  - How do you get them to become an expert?
  - How do you get them to use it for a long time?
  - How do you scientifically evaluate?
  - How go you get actionable feedback?
- A “Center for Evaluating Multicore Programming Environments”??

Identify, Collect, Standardize, Adopt

- Good languages/tools cannot be designed by committee
- However, you need a vibrant ecosystem of ideas
- Need a process of natural selection
  - Quantify Productivity and Performance
  - Competition between multiple teams
  - Winner(s) get to design the final language

Migrate the Dusty Deck

- Impossible to bring them to the new era automatically
  - Badly mangled, hand-optimized, impossible to analyze code
  - Automatic compilation, even with a heroic effort, cannot do anything
- Help rewrite the huge stack of dusty deck
  - Application in use
  - Source code available
  - Programmer long gone
- Getting the new program to have the same behavior is hard
  - “Word pagination problem”
- Can take advantage of many recent advances
  - Creating test cases
  - Extracting invariants
  - Failure oblivious computing

Outline

- Evolution
  - Trends
  - Architecture
  - Languages, Compilers and Tools
- **Revolution**
- Crossing the Abstraction Boundaries

How about Revolutions?

- What are the far-out technologies?
- Wishful Thinking?

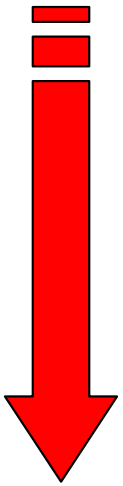
Outline

- Evolution
  - Trends
  - Architecture
  - Languages, Compilers and Tools
- Revolution
- **Crossing the Abstraction Boundaries**

Computer Systems from 10,000 feet

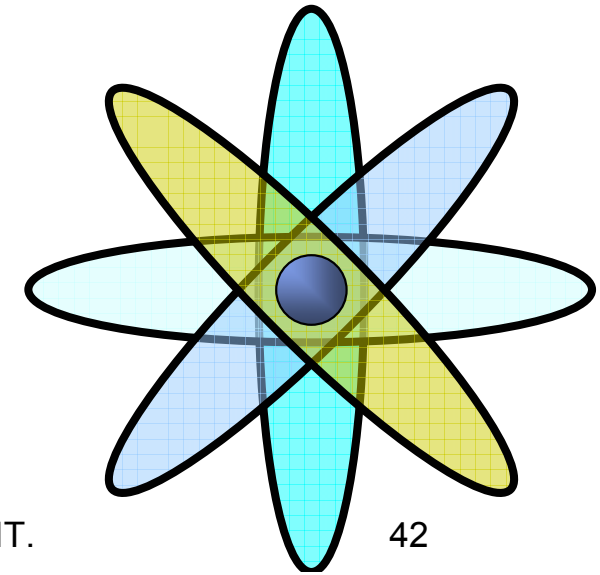
class of computation

```
foo(int x)
{ .. }
```



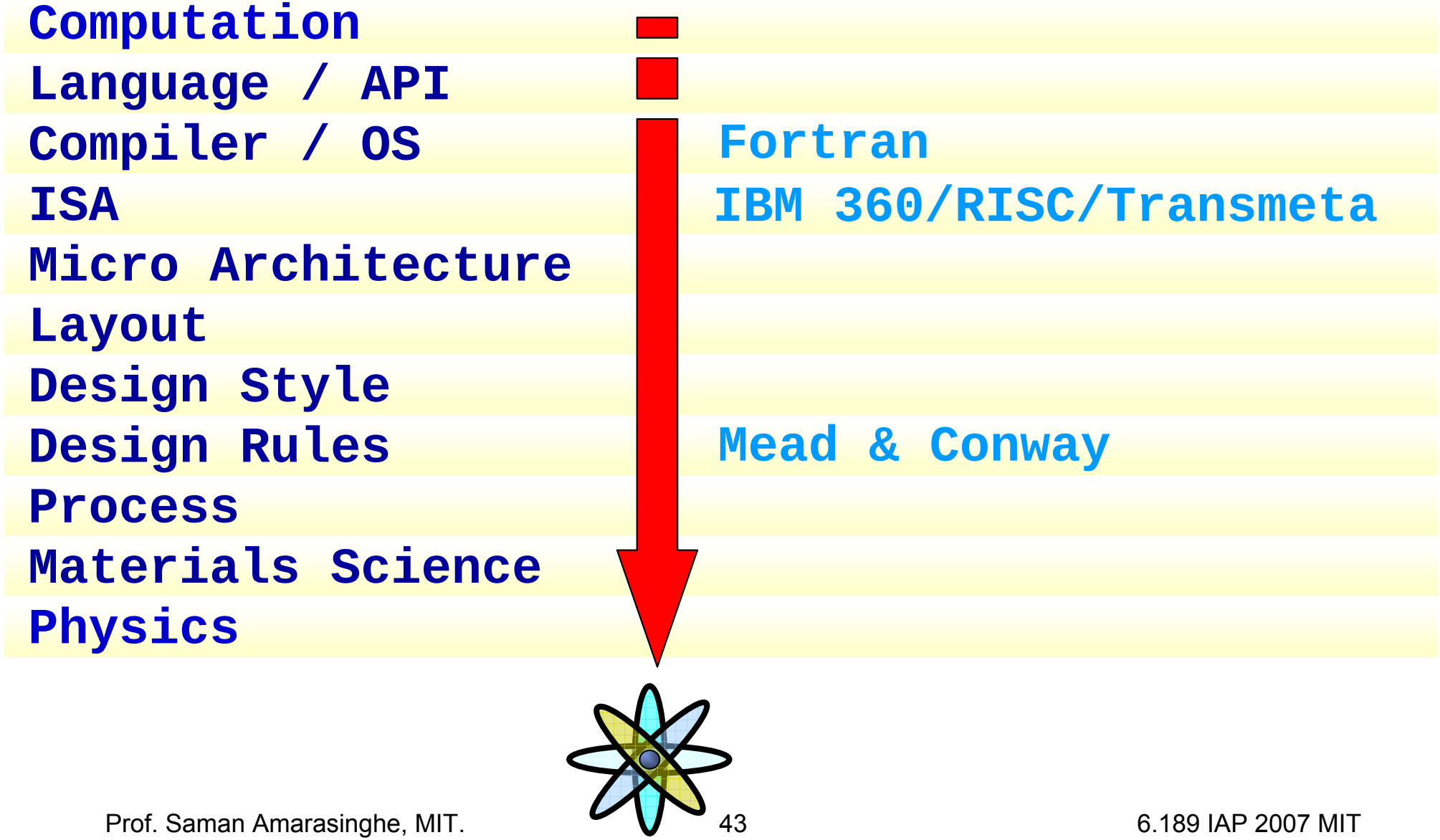
… we use abstractions to make this easier

convenient physical phenomenon



The Abstraction Layers Make This Easier

```
foo(int x) { .. }
```



A Case Against Entrenched Abstractions

```
foo(int x) { .. }
```

Computation
Language / API
Compiler / OS
ISA
Micro Architecture
Layout
Design Style
Design Rules
Process
Materials Science
Physics