The Looming Software Crisis due to the Multicore Menace

Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

The “Software Crisis”

“To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

-- E. Dijkstra, 1972 Turing Award Lecture

The First Software Crisis

- Time Frame: ’60s and ’70s
- Problem: Assembly Language Programming
  - Computers could handle larger more complex programs
- Needed to get Abstraction and Portability without losing Performance

How Did We Solve the First Software Crisis?

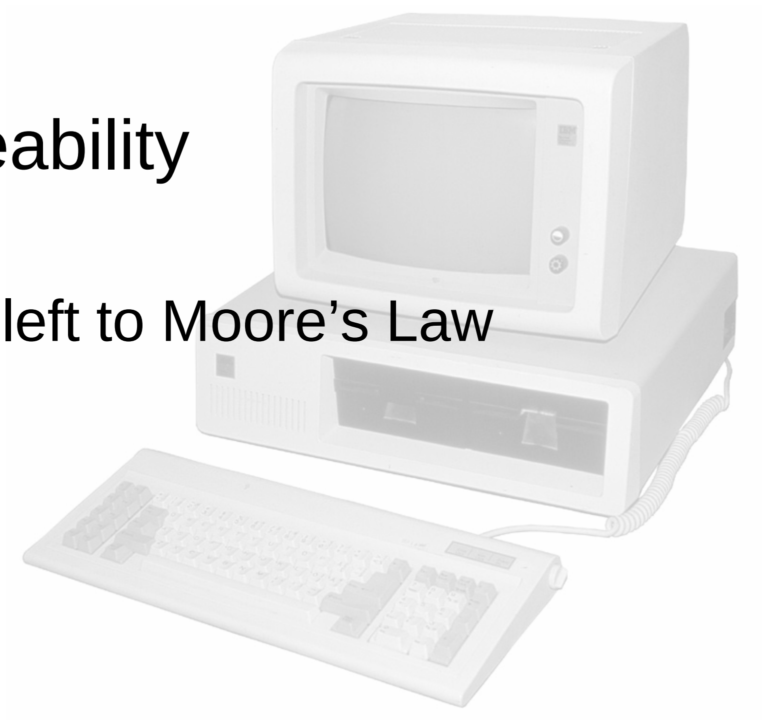
- High-level languages for von-Neumann machines
  - FORTRAN and C
- Provided “common machine language” for uniprocessors
  - Only the common properties are exposed
  - Hidden properties are backed-up by good compiler technology

Common Properties
Single flow of control
Single memory image

Differences:	
Register File	Register Allocation
ISA	Instruction Selection
Functional Units	Instruction Scheduling

The Second Software Crisis

- Time Frame: ’80s and ’90s
- Problem: Inability to build and maintain complex and robust applications requiring multi-million lines of code developed by hundreds of programmers
  - Computers could handle larger more complex programs
- Needed to get Composability, Malleability and Maintainability
  - High-performance was not an issue → left to Moore’s Law

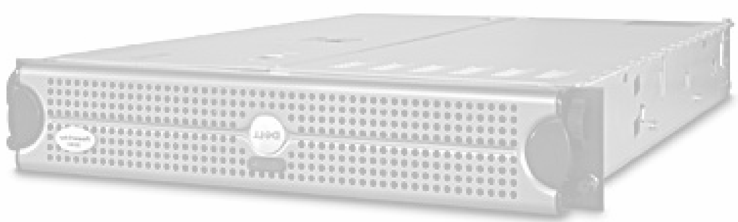


How Did We Solve the Second Software Crisis?

- Object Oriented Programming
  - C++
  - Now C# and Java
- Better tools
  - Component libraries, Purify
- Better software engineering methodology
  - Design patterns, specification, testing, code reviews

Today: Programmers are Oblivious about the Processors

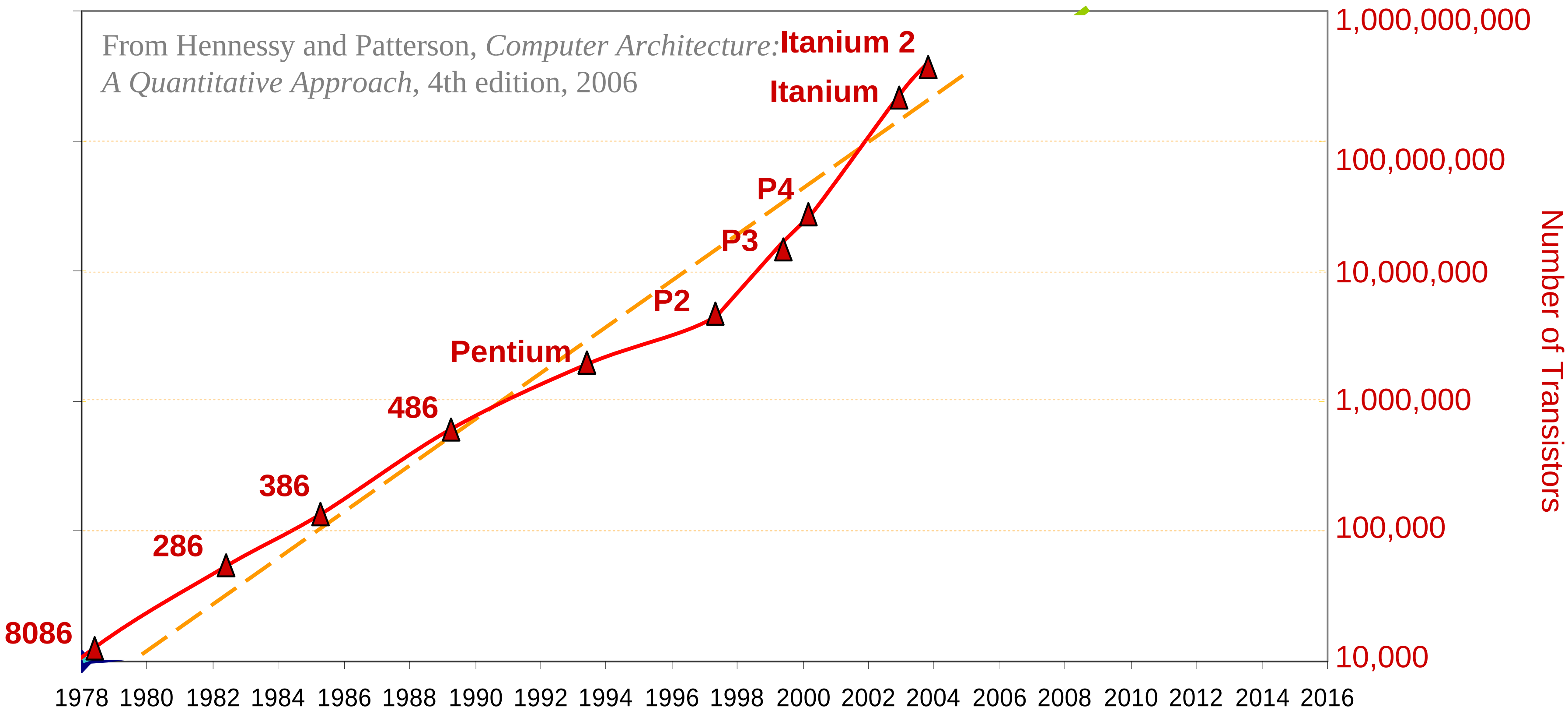
- Solid boundary between Hardware and Software
- Programmers don't have to know anything about the processor
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
  - Moore's law does not require the programmers to know anything about the processors to get good speedups
- Programs are oblivious to the processor → works on all processors
  - A program written in '70 using C still works and is much faster today
- This abstraction provides a lot of freedom for the programmers



The Origins of a Third Crisis

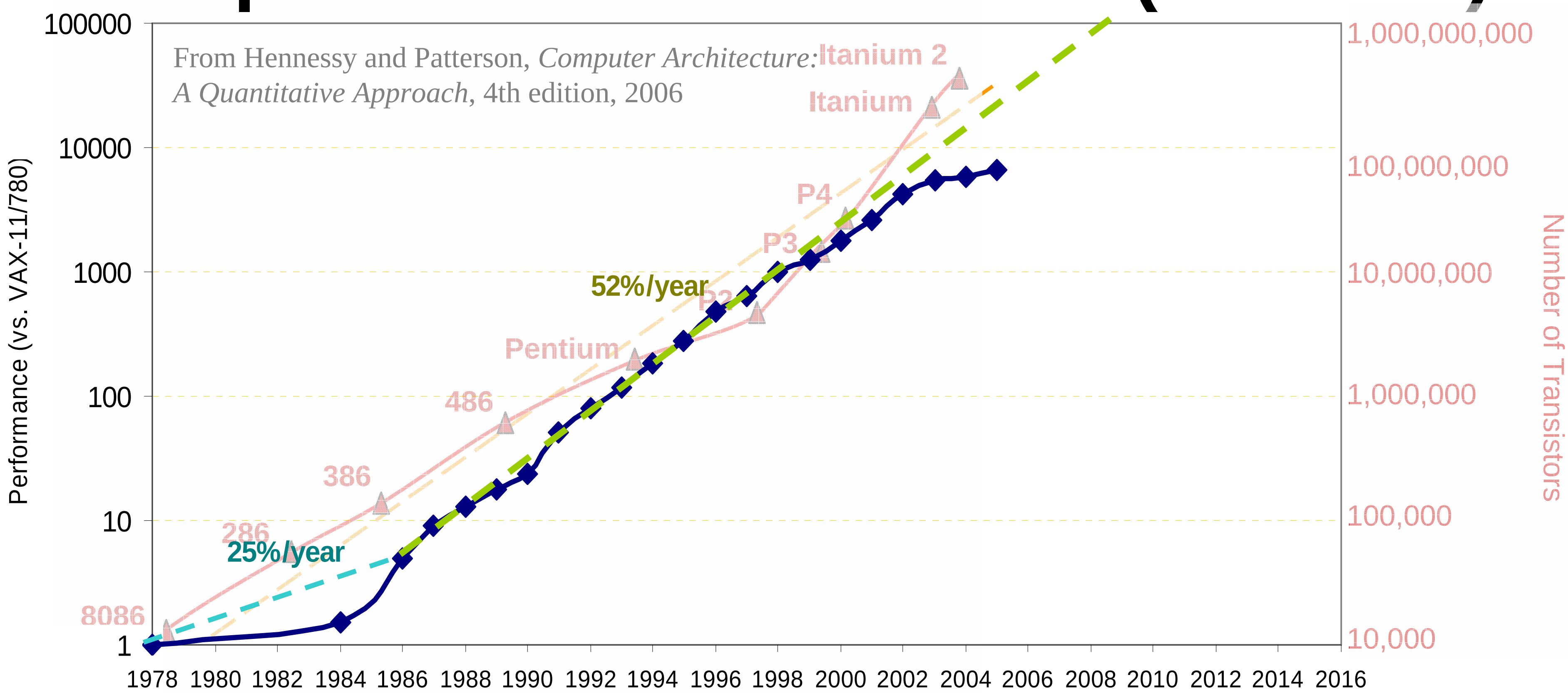
- Time Frame: 2010 to ??
- Problem: Sequential performance is left behind by Moore’s law
  – All software developers have abstracted away the processor assuming that Moore’s law will always provide performance gains!

The March to Multicore: Moore's Law



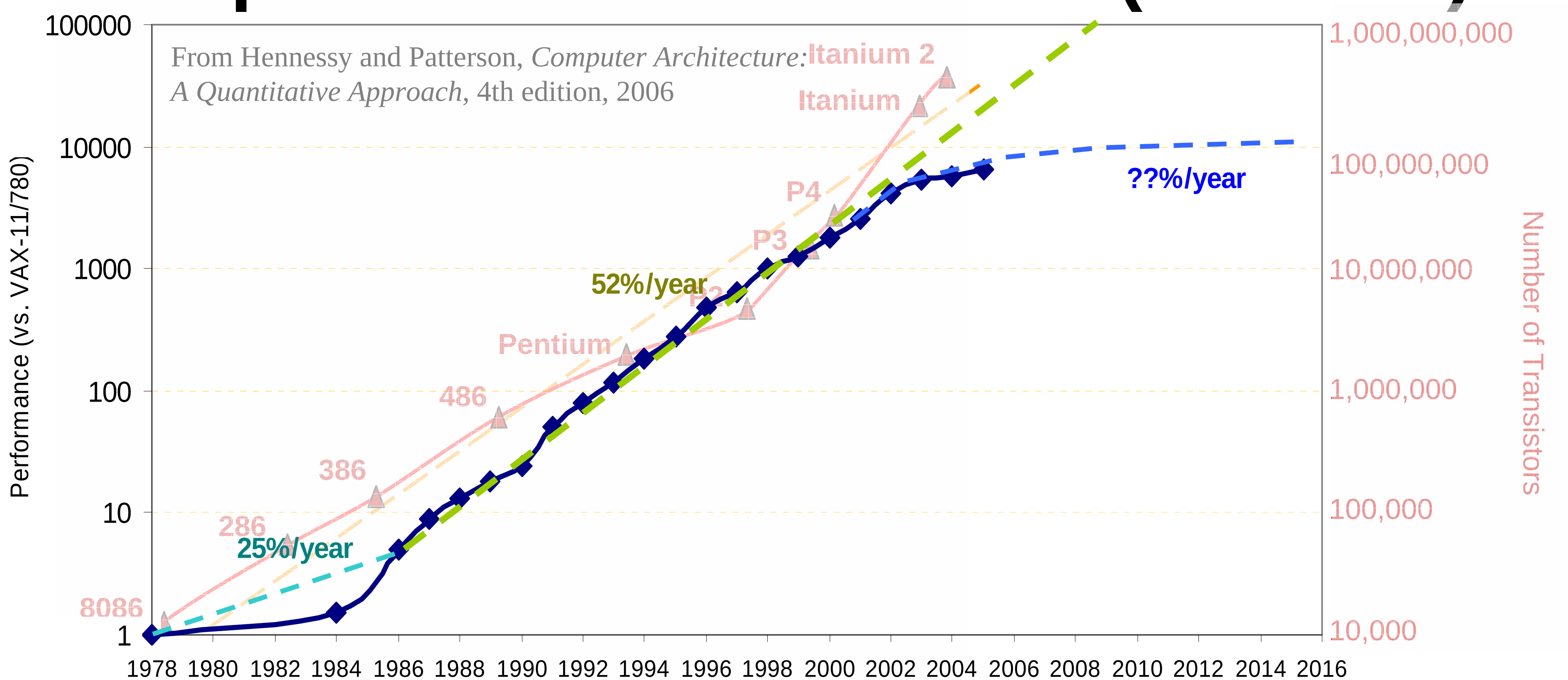
From David Patterson

The March to Multicore: Uniprocessor Performance (SPECint)



From David Patterson

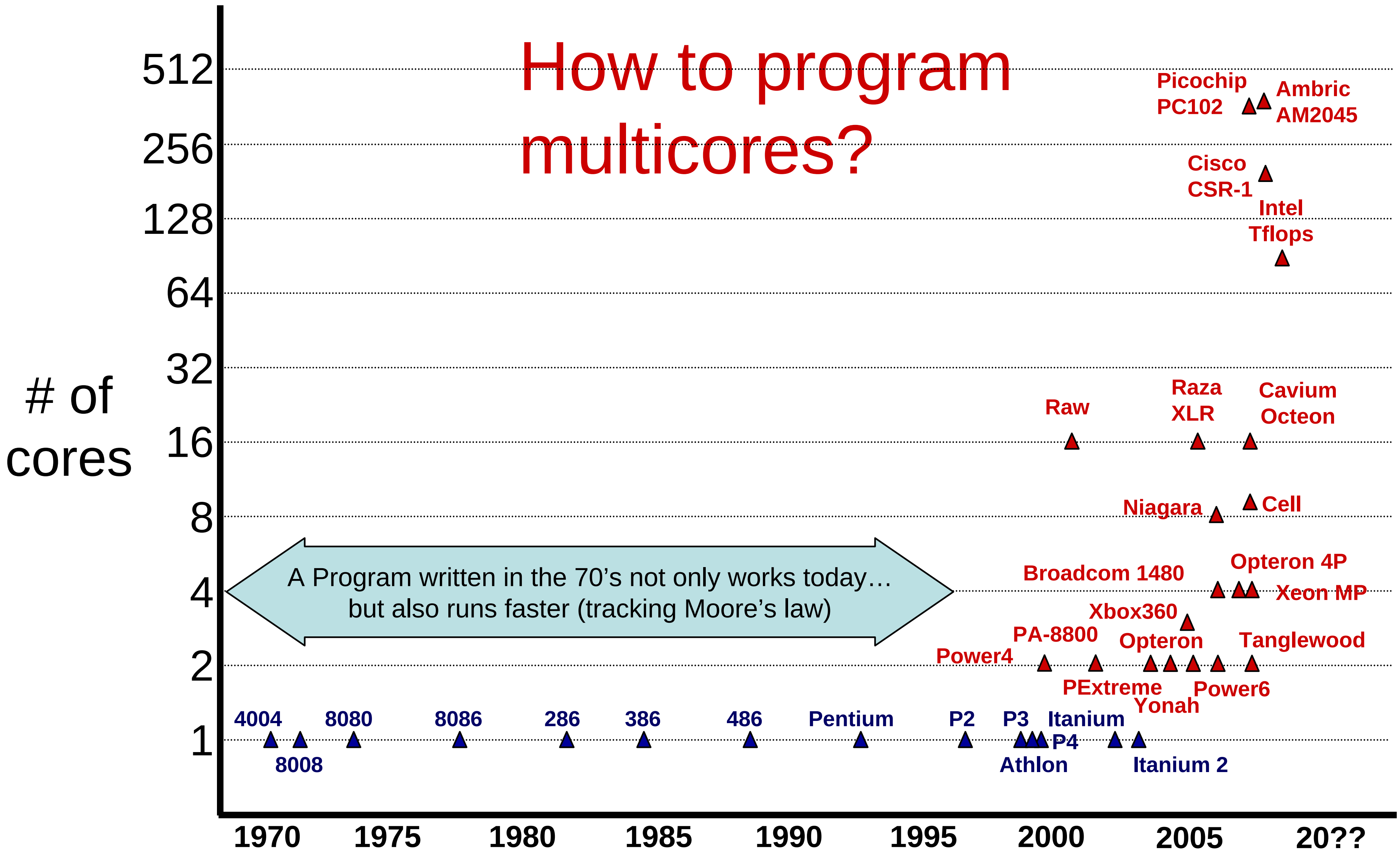
The March to Multicore: Uniprocessor Performance (SPECint)



- General-purpose unicores have stopped historic performance scaling
  - Power consumption
  - Wire delays
  - DRAM access latency
  - Diminishing returns of more instruction-level parallelism

From David Patterson

The Origins of a Third Crisis



The Origins of a Third Crisis

- Time Frame: 2010 to ??
- Problem: Sequential performance is left behind by Moore's law
- Needed continuous and reasonable performance improvements
  - to support new features
  - to support larger datasets
- While sustaining portability, malleability and maintainability without unduly increasing complexity faced by the programmer

→ critical to keep-up with the current rate of evolution in software

Why Parallelism is Hard

- A huge increase in complexity and work for the programmer
  - Programmer has to think about performance!
  - Parallelism has to be designed in at every level

- Humans are sequential beings
  - Deconstructing problems into parallel tasks is hard for many of us

- Parallelism is not easy to implement
  - Parallelism cannot be abstracted or layered away
  - Code and data has to be restructured in very different (non-intuitive) ways

- Parallel programs are very hard to debug
  - Combinatorial explosion of possible execution orderings
  - Race condition and deadlock bugs are non-deterministic and illusive
  - Non-deterministic bugs go away in lab environment and with instrumentation

Outline: Ideas on Solving the Third Software Crisis

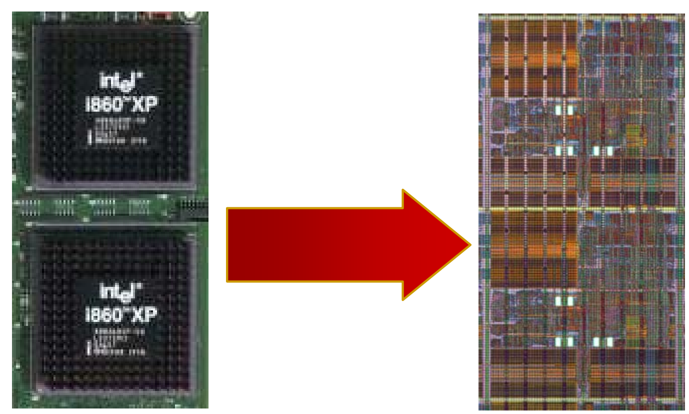
1. Advances in Computer Architecture
2. Novel Programming Models and Languages
3. Aggressive Compilers
4. Tools to support parallelization, debugging and migration

Computer Architecture

- Current generation of multicores
  - How can we cobble together something with existing parts/investments?
  - Impact of multicores haven't hit us yet
- The move to multicore will be a disruptive shift
  - An inversion of the cost model
  - A forced shift in the programming model
- A chance to redesign the microprocessor from scratch.
- What are the innovations that will reduce/eliminate the extra burden placed on the programmer?

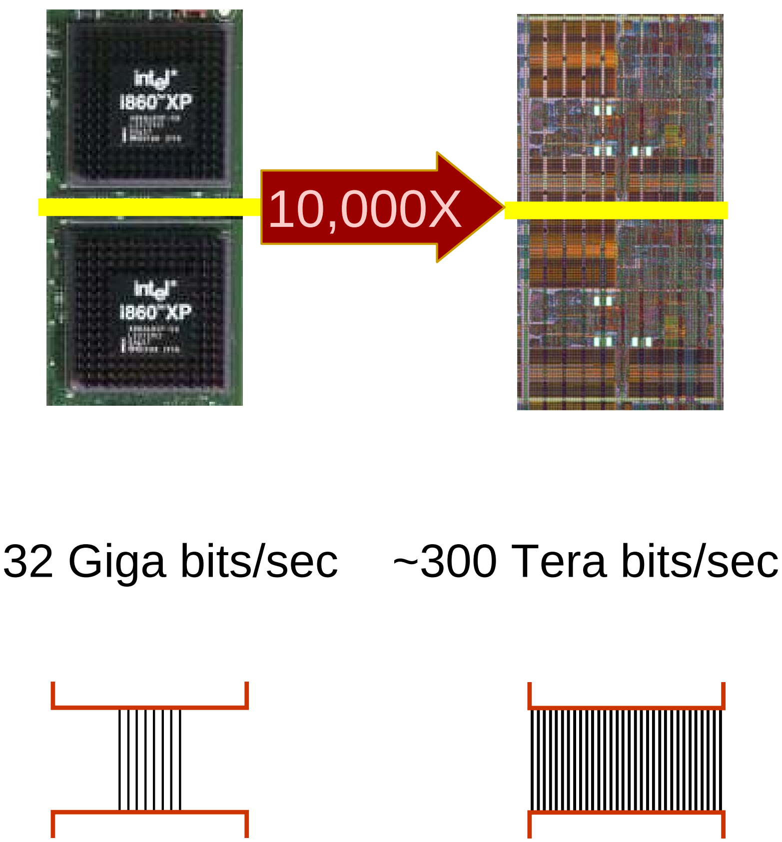
Novel Opportunities in Multicores

- Don't have to contend with uniprocessors
- 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



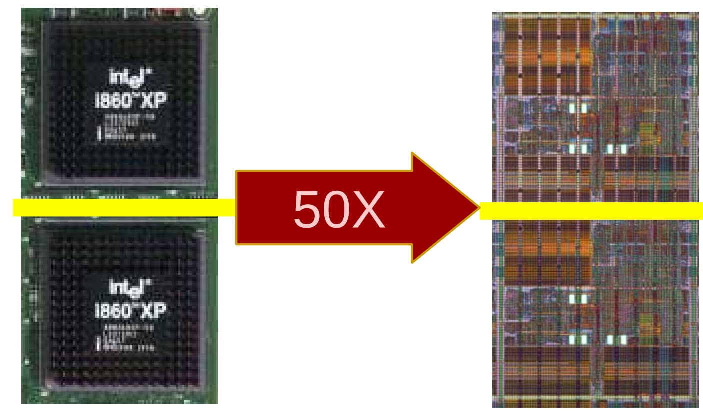
Communication Bandwidth

- How much data can be communicated between two cores?
- What changed?
  - Number of Wires
  - Clock rate
  - Multiplexing
- Impact on programming model?
  - Massive data exchange is possible
  - Data movement is not the bottleneck
    → processor affinity not that important



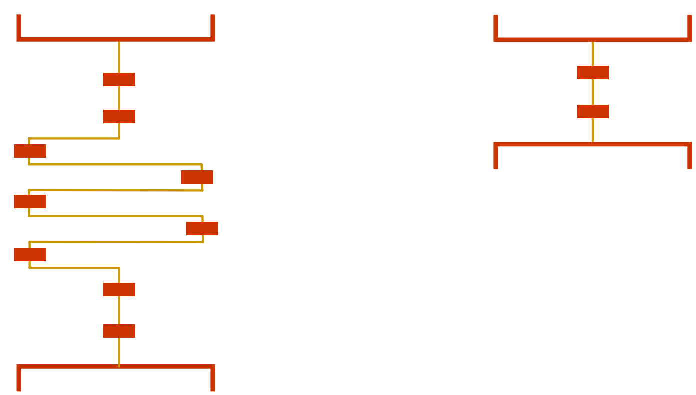
Communication Latency

- How long does it take for a round trip communication?
- What changed?
  - Length of wire
  - Pipeline stages
- Impact on programming model?
  - Ultra-fast synchronization
  - Can run real-time apps on multiple cores



~200 Cycles

~4 cycles



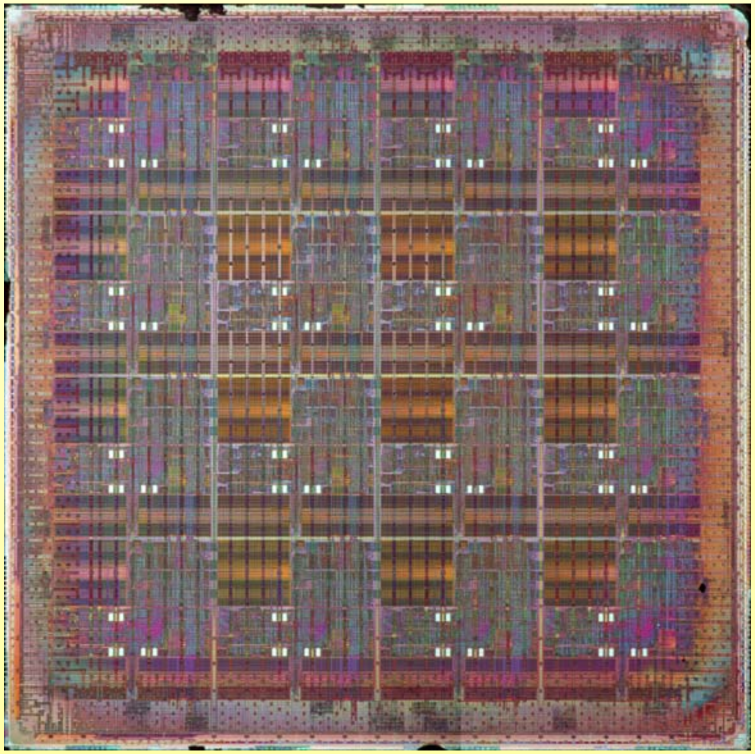
Architectural Innovations

The Raw Experience

Supported by a CISE
Experimental Partnership

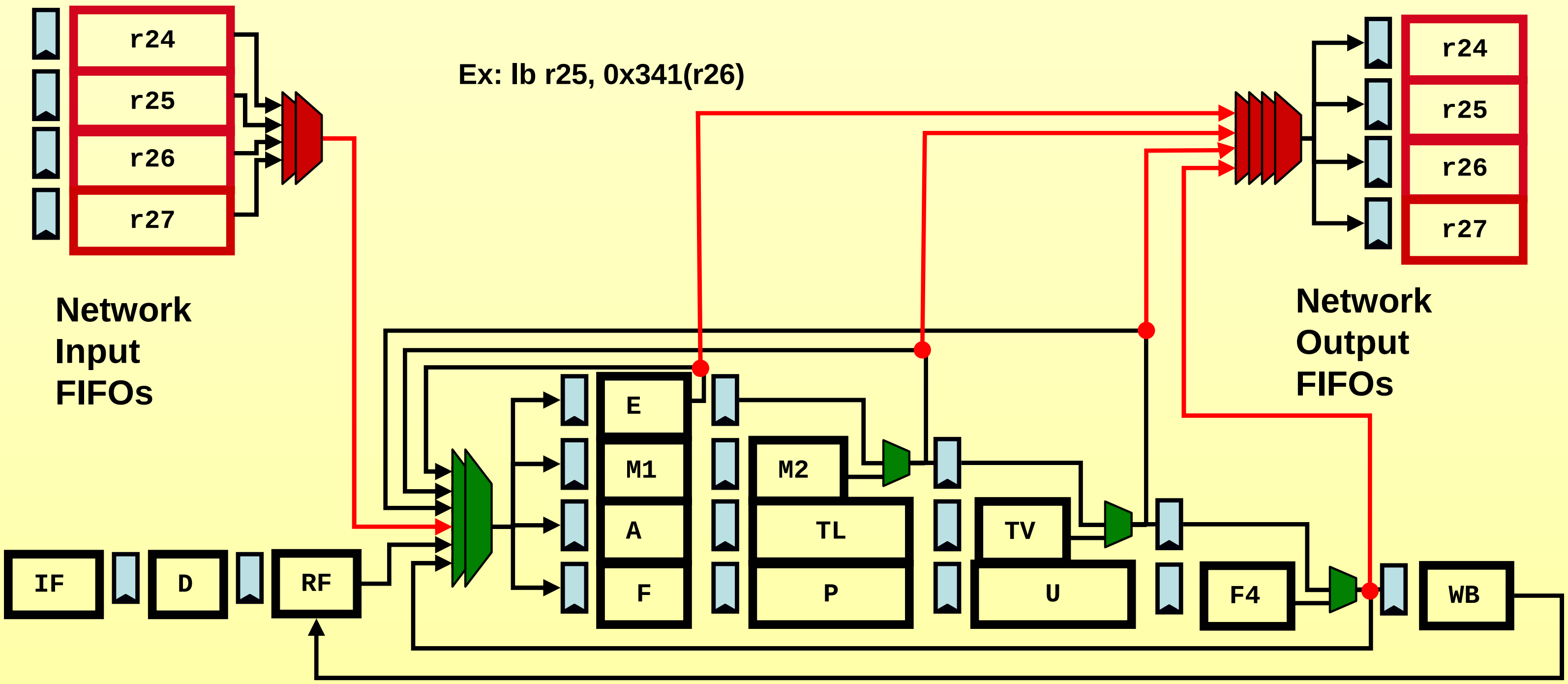
The MIT Raw Processor

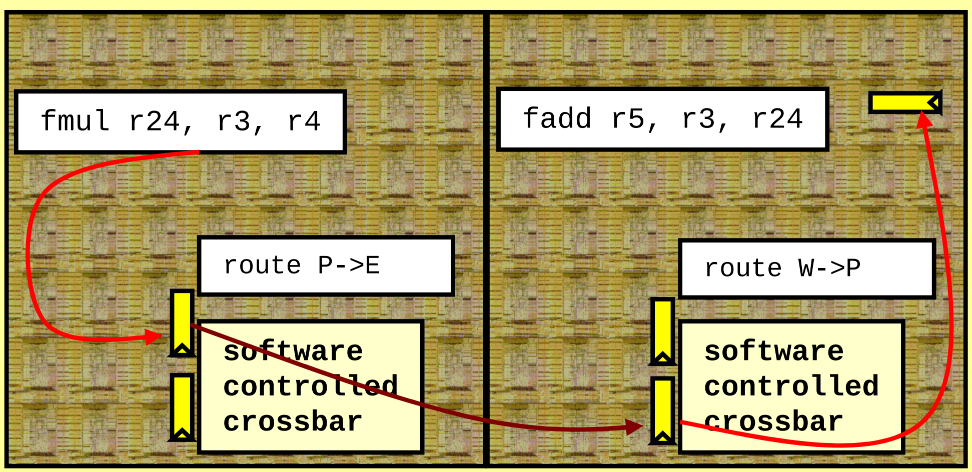
- Raw project started in 1997
  Prototype operational in 2003
- The Problem: How to keep the Moore's Law going with
  - Increasing processor complexity
  - Longer wire delays
  - Higher power consumption
- Raw philosophy
  - Build a tightly integrated multicore
  - Off-load most functions to compilers and software
- Raw design
  - 16 single issue cores
  - 4 register-mapped networks
  - Huge IO bandwidth
- Raw power
  - 16 Flops/ops per cycle
  - 16 Memory Accesses per cycle
  - 208 Operand Routes per cycle
  - 12 IO Operations per cycle



180 nm ASIC (IBM SA-27E)
18.2mm x 18.2mm
~100 million transistors
Designed for 225 MHz
Tested at 425 MHz

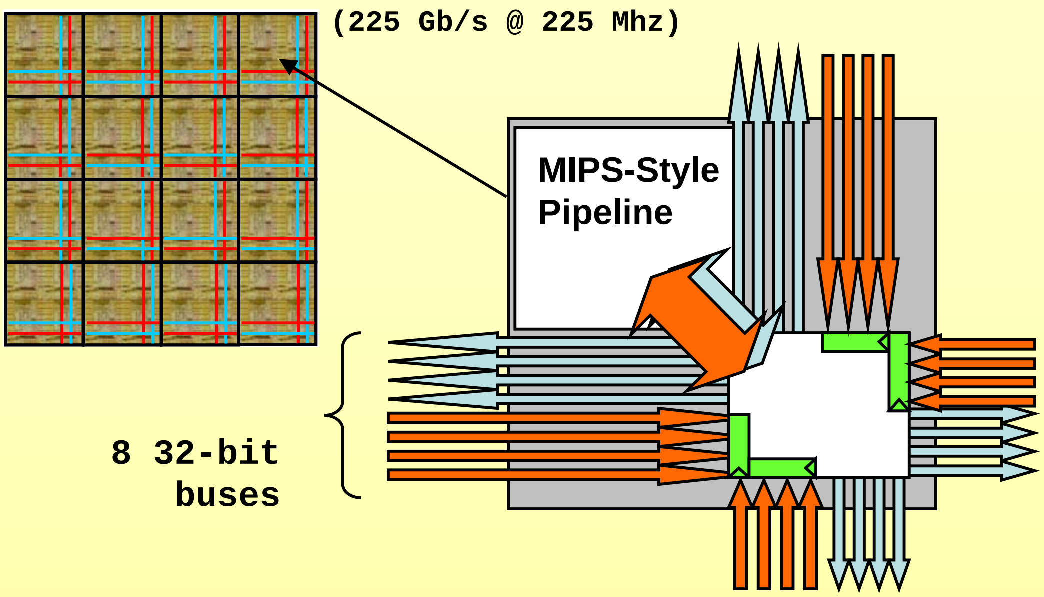
Raw's networks are tightly coupled into the bypass paths

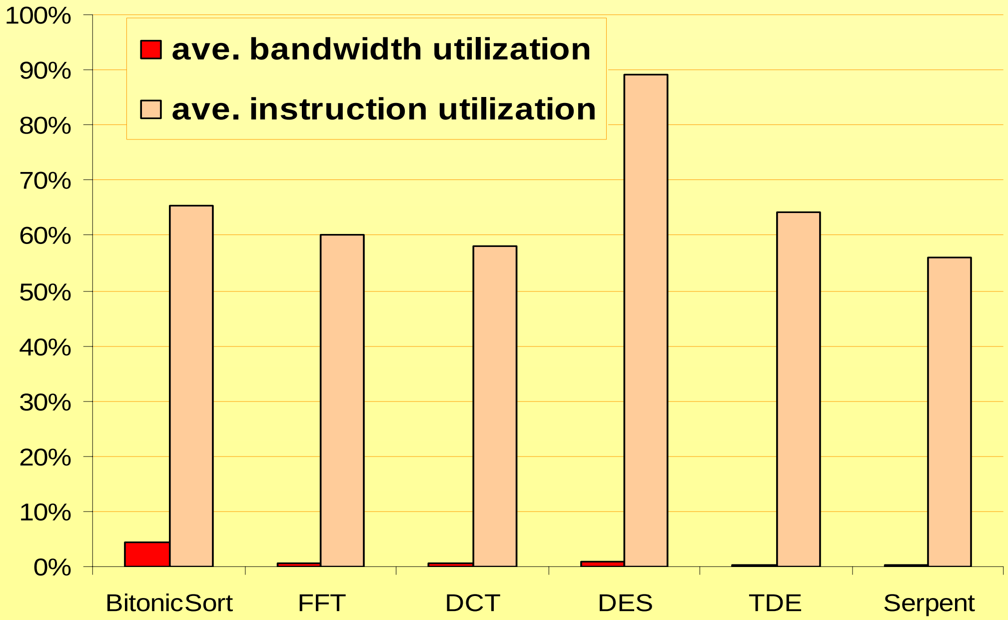




Raw Networks is Rarely the Bottleneck

- Raw has 4 bidirectional, point-to-point mesh networks
  - Two of them statically routed
  - Two of the dynamically routed
- A single issue core may read from or write to one network in a given cycle
- The cores cannot saturate the network!





Outline: Ideas on Solving the Third Software Crisis

1. Advances in Computer Architecture

2. Novel Programming Models and Languages

3. Aggressive Compilers

4. Tools to support parallelization, debugging and migration

Programming Models and Languages

- Critical to solving the third software crisis
- Novel languages were the central solution in the last two crises

Lessons from the Last Crisis: 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 Enviornment*

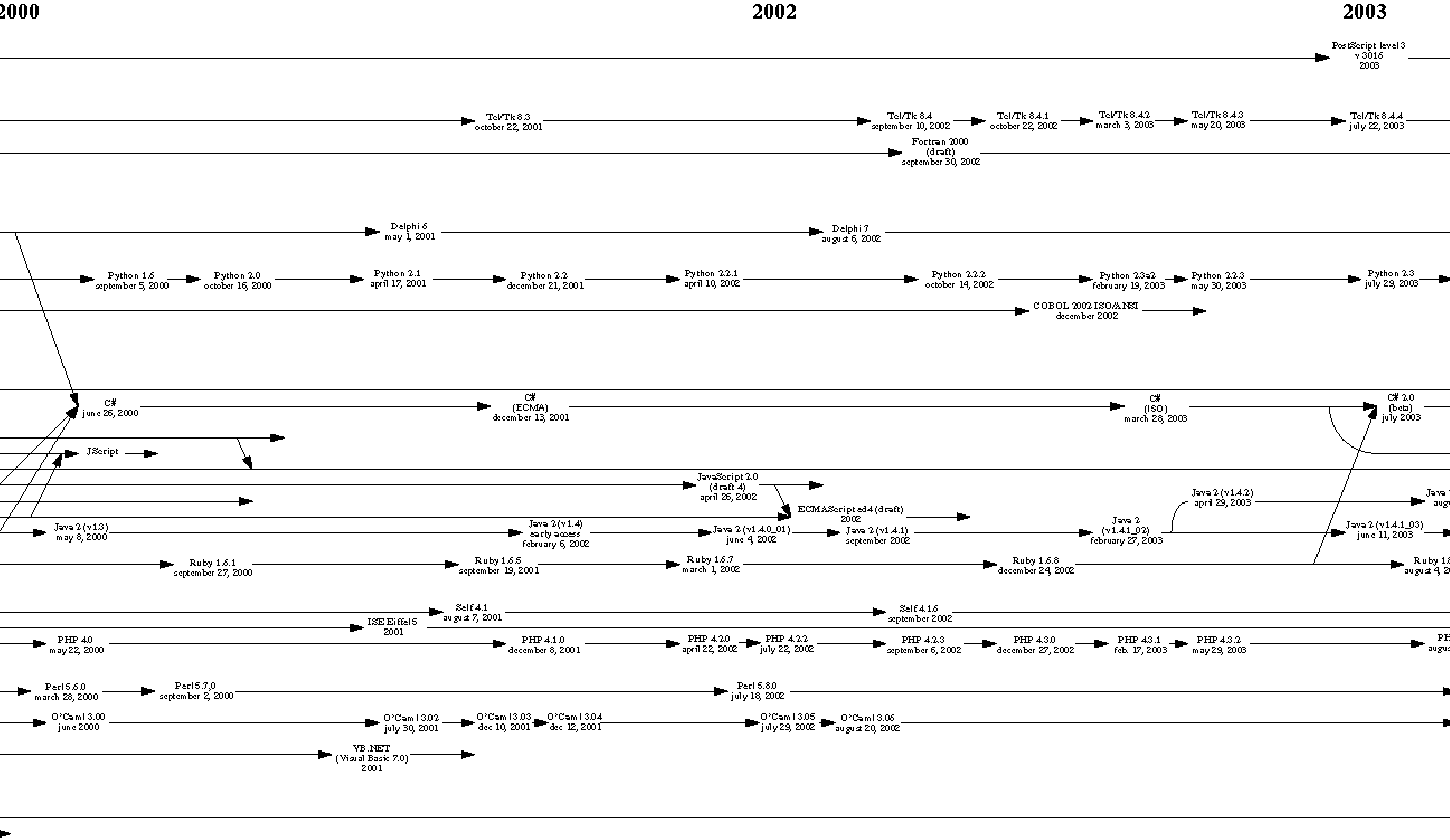
Object Oriented Languages

- 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

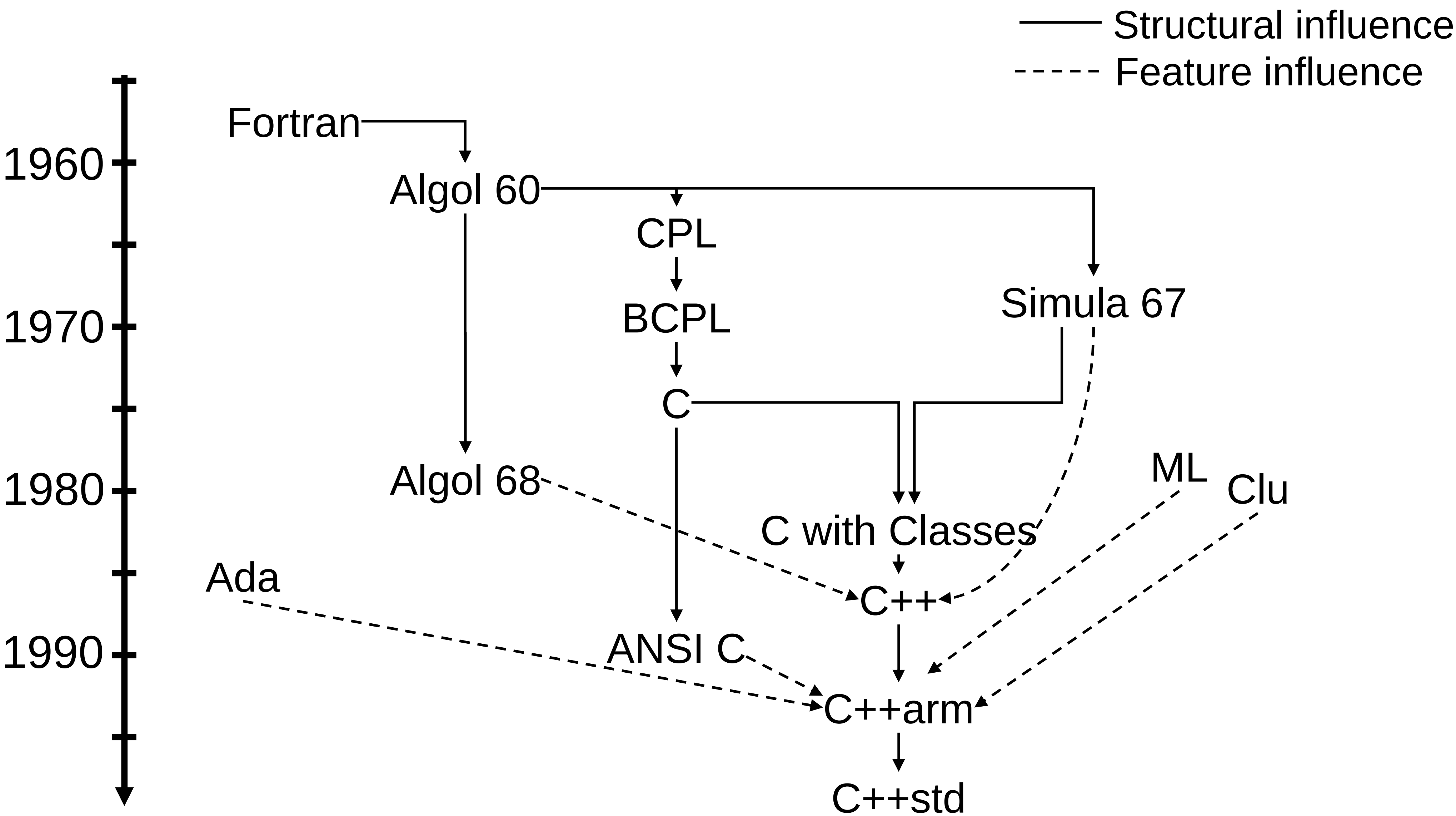
Language Evolution

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 Enviornment*

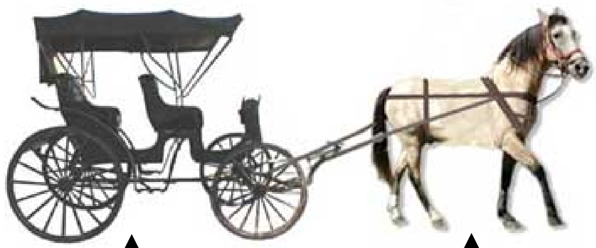
Why New Programming Models and Languages?

- 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 ⇒ 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 DOD and other applications

Programming Languages and Architectures



C ⇔ von-Neumann machine

Modern architecture

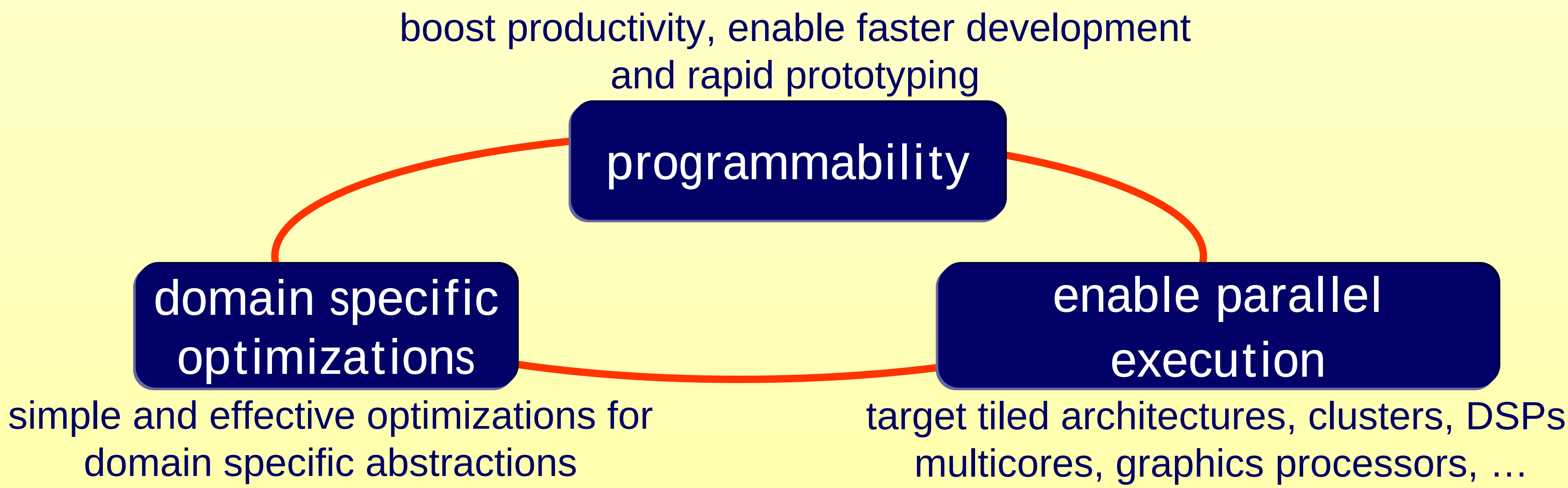
- Two choices:
  - Bend over backwards to support old languages like C/C++
  - Develop parallel architectures that are hard to program

Compiler-Aware Language Design

The StreamIt Experience

Supported by ITR
and NGS Awards

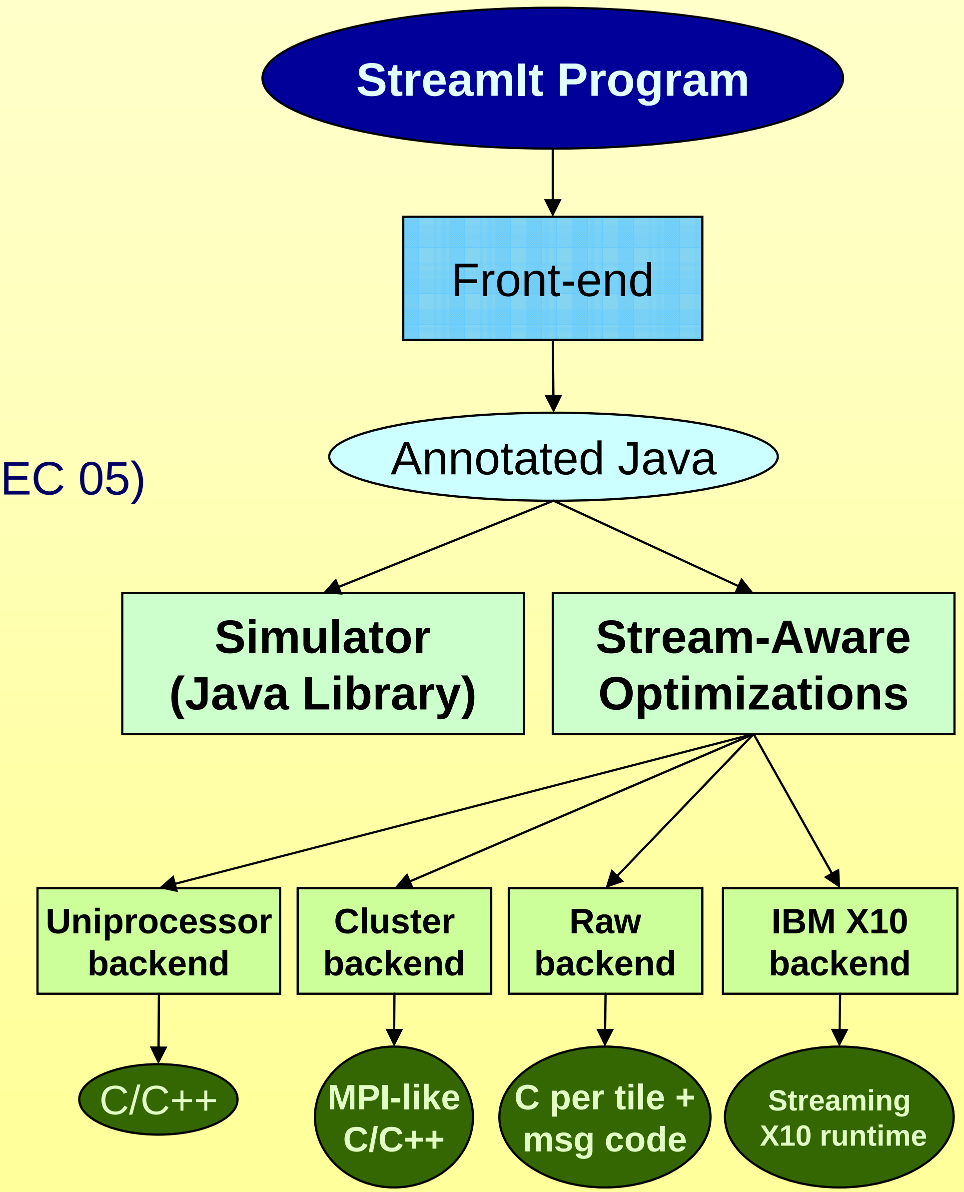
Is there a win-win situation?



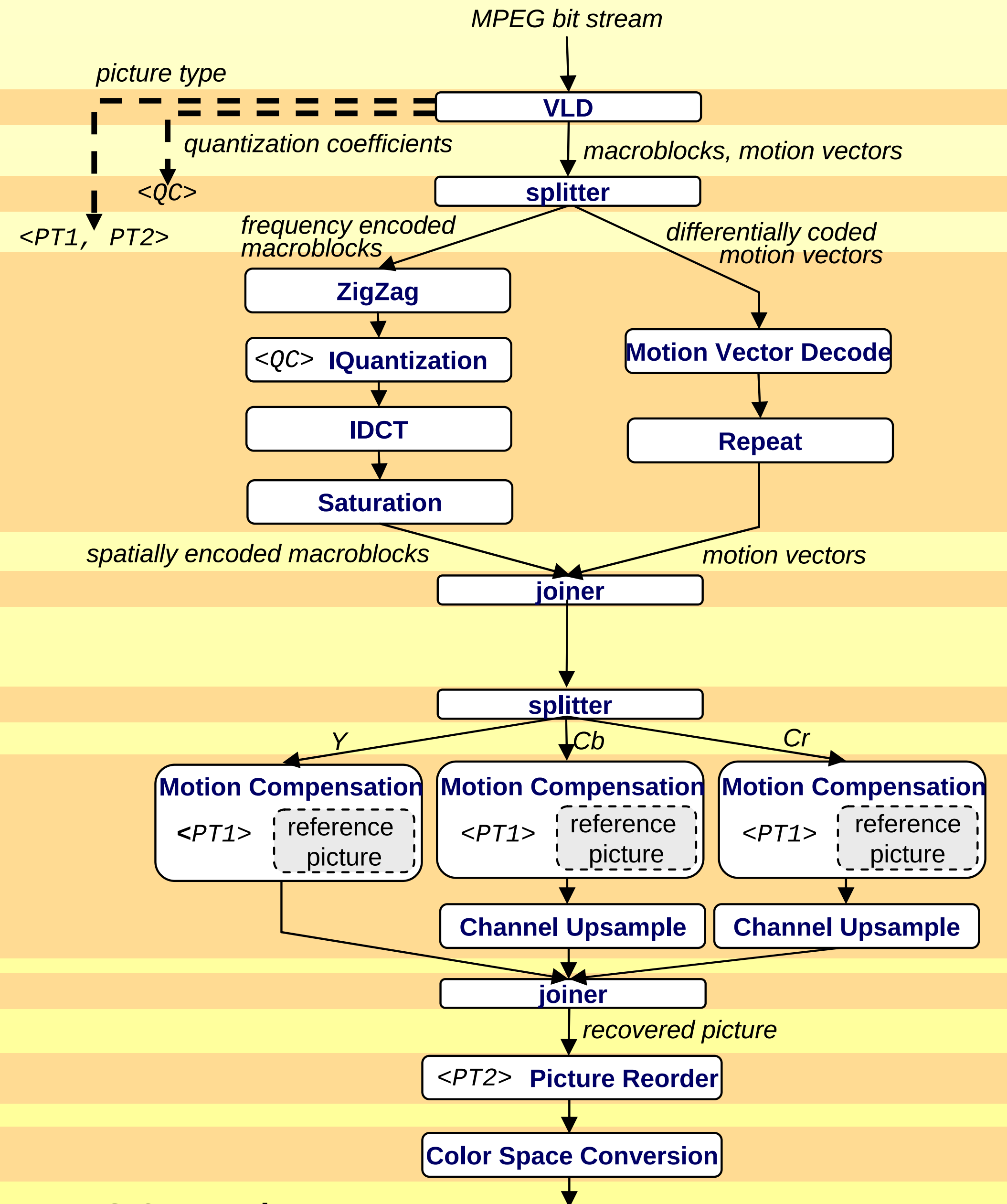
- Some programming models are inherently concurrent
  - Coding them using a sequential language is…
    - Harder than using the right parallel abstraction
    - All information on inherent parallelism is lost
- There are win-win situations
  - Increasing the programmer productivity while extracting parallel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

The StreamIt Project

- **Applications**
  - DES and Serpent [PLDI 05]
  - MPEG-2 [IPDPS 06]
  - SAR, DSP benchmarks, JPEG, …
- **Programmability**
  - StreamIt Language (CC 02)
  - Teleport Messaging (PPOPP 05)
  - Programming Environment in Eclipse (P-PHEC 05)
- **Domain Specific Optimizations**
  - Linear Analysis and Optimization (PLDI 03)
  - Optimizations for bit streaming (PLDI 05)
  - Linear State Space Analysis (CASES 05)
- **Architecture Specific Optimizations**
  - Compiling for Communication-Exposed Architectures (ASPLOS 02)
  - Phased Scheduling (LCTES 03)
  - Cache Aware Optimization (LCTES 05)
  - Load-Balanced Rendering (Graphics Hardware 05)
  - Task, Data and Pipeline Parallelism (ASPLOS 06)



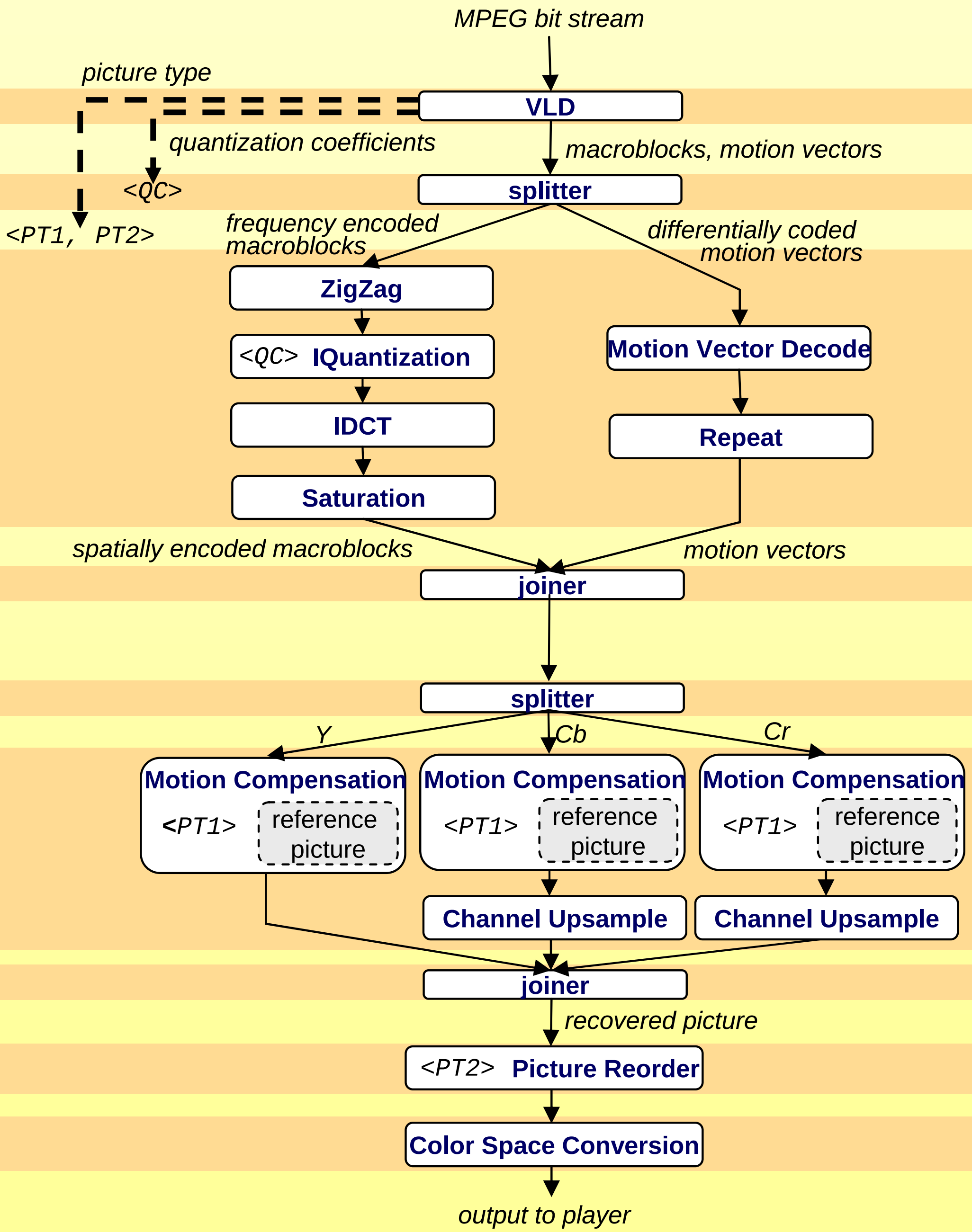
Streaming Application Abstraction



MPEG-2 Decoder

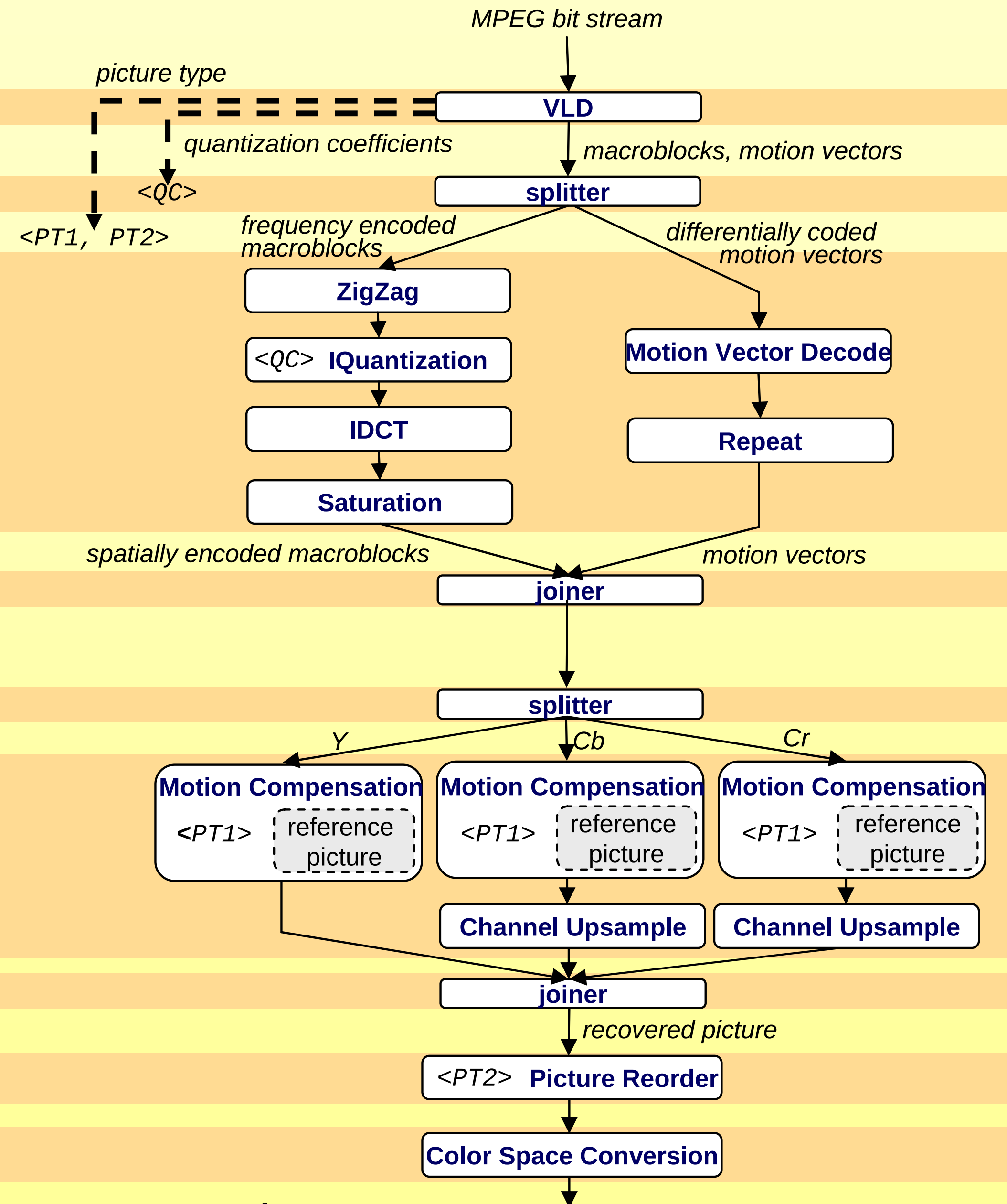
- Structured block level diagram describes computation and flow of data
- Conceptually easy to understand
  - Clean abstraction of functionality
- Mapping to C (sequentialization) destroys this simple view

StreamIt Improves Productivity




```
add VLD(QC, PT1, PT2);
add splitjoin {
    split roundrobin(N*B, V);

    add pipeline {
        add ZigZag(B);
        add IQuantization(B) to QC;
        add IDCT(B);
        add Saturation(B);
    }
    add pipeline {
        add MotionVectorDecode();
        add Repeat(V, N);
    }

    join roundrobin(B, V);
}
add splitjoin {
    split roundrobin(4*(B+V), B+V, B+V);

    add MotionCompensation(4*(B+V)) to PT1;
    for (int i = 0; i < 2; i++) {
        add pipeline {
            add MotionCompensation(B+V) to PT1;
            add ChannelUpsample(B);
        }
    }

    join roundrobin(1, 1, 1);
}
add PictureReorder(3*W*H) to PT2;

add ColorSpaceConversion(3*W*H);
```

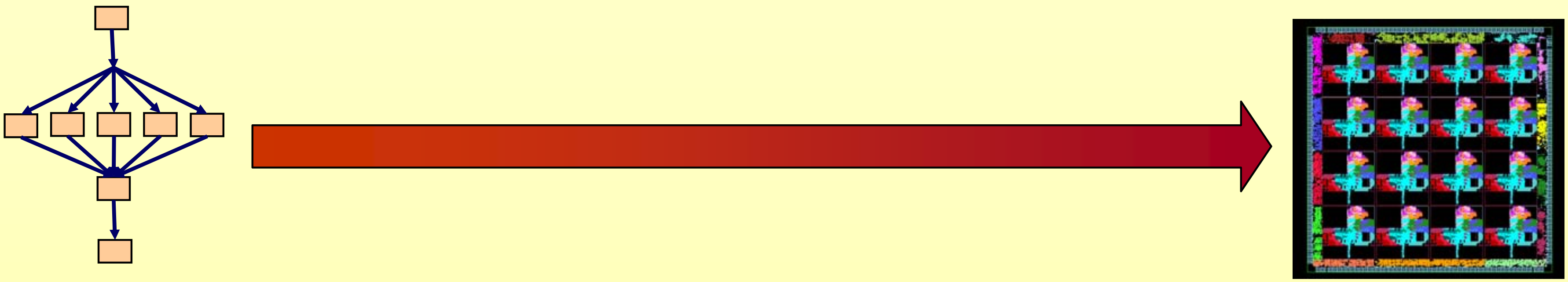
StreamIt Compiler Extracts Parallelism



MPEG-2 Decoder

- Task Parallelism
  - Thread (fork/join) parallelism
  - Parallelism explicit in algorithm
  - Between filters *without* producer/consumer relationship
- Data Parallelism
  - Data parallel loop (**forall**)
  - Between iterations of a *stateless* filter
  - Can't parallelize filters with state
- Pipeline Parallelism
  - Usually exploited in hardware
  - Between producers and consumers
  - *Stateful* filters can be parallelized

StreamIt Compiler
Parallelism → Processor Resources

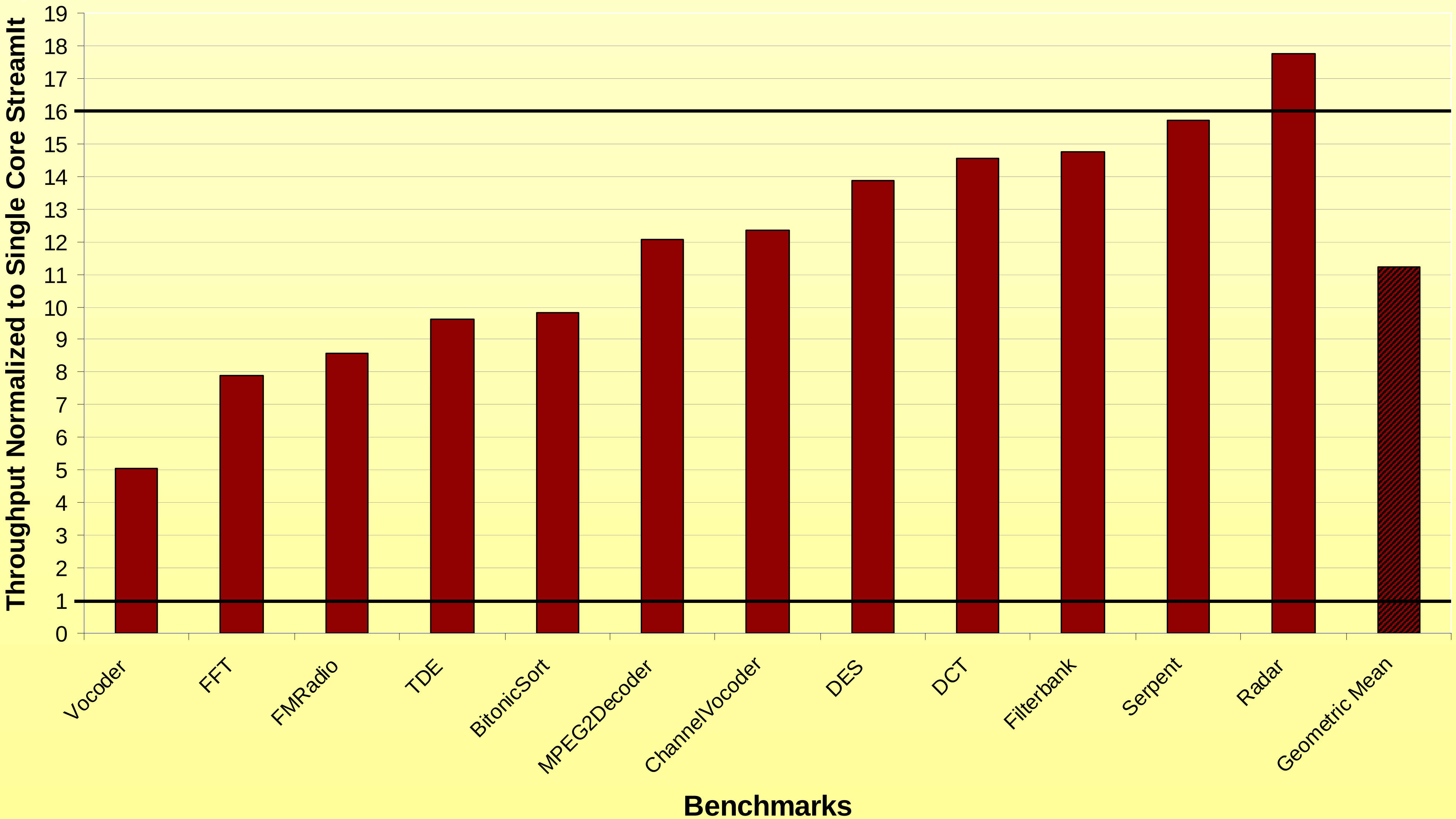


- StreamIt Compilers Finds the Inherent Parallelism
  - Graph structure is architecture independent
  - Abundance of parallelism in the StreamIt domain

- Too much parallelism is as bad as too little parallelism
  - (remember dataflow!)

- Map the parallelism in to the available resources in a given multicore
  - Use all available parallelism
  - Maximize load-balance
  - Minimize communication

StreamIt Performance on Raw



Outline: Ideas on Solving the Third Software Crisis

1. Advances in Computer Architecture
2. Novel Programming Models and Languages
3. Aggressive Compilers
4. Tools to support parallelization, debugging and migration

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

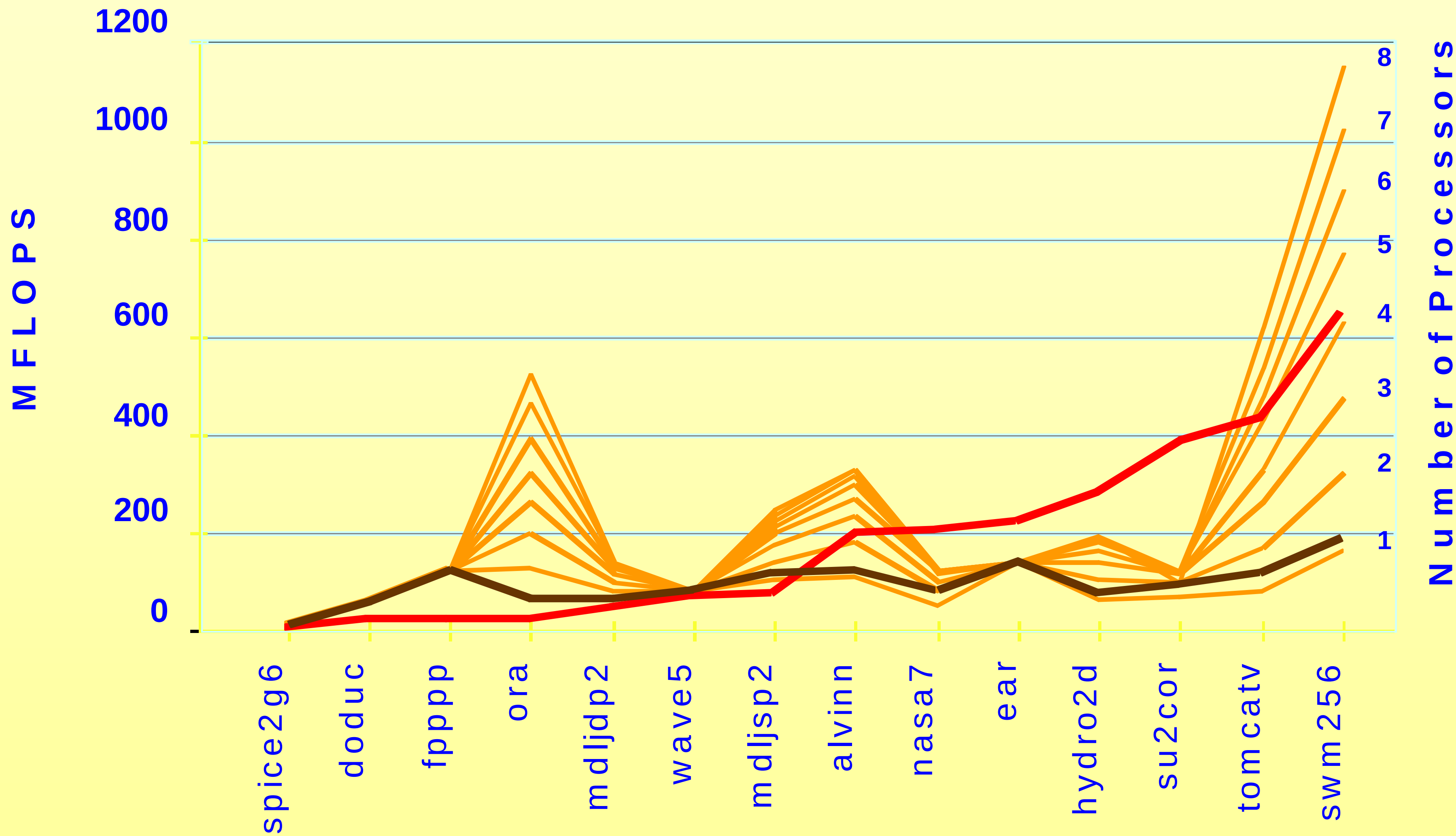
Parallelizing Compilers

The SUIF Experience

The SUIF Parallelizing Compiler

- The SUIF Project at Stanford in the ’90
  - Mainly FORTRAN
  - Aggressive transformations to undo “human optimizations”
  - Interprocedural analysis framework
  - Scalar and array data-flow, reduction recognition and a host of other analyses and transformations
- SUIF compiler had the Best SPEC results by automatic parallelization

SPECFP92 performance



• Vector processor	Cray C90	540
• Uniprocessor	Digital 21164	508
• SUIF on 8 processors	Digital 8400	1,016

Automatic Parallelization "Almost" Worked

- Why did not this reach mainstream?
  - 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)

Outline: Ideas on Solving the Third Software Crisis

1. Advances in Computer Architecture
2. Novel Programming Models and Languages
3. Aggressive Compilers
4. Tools to support parallelization, debugging and migration

Tools

- A lot of progress in tools to improve programmer productivity

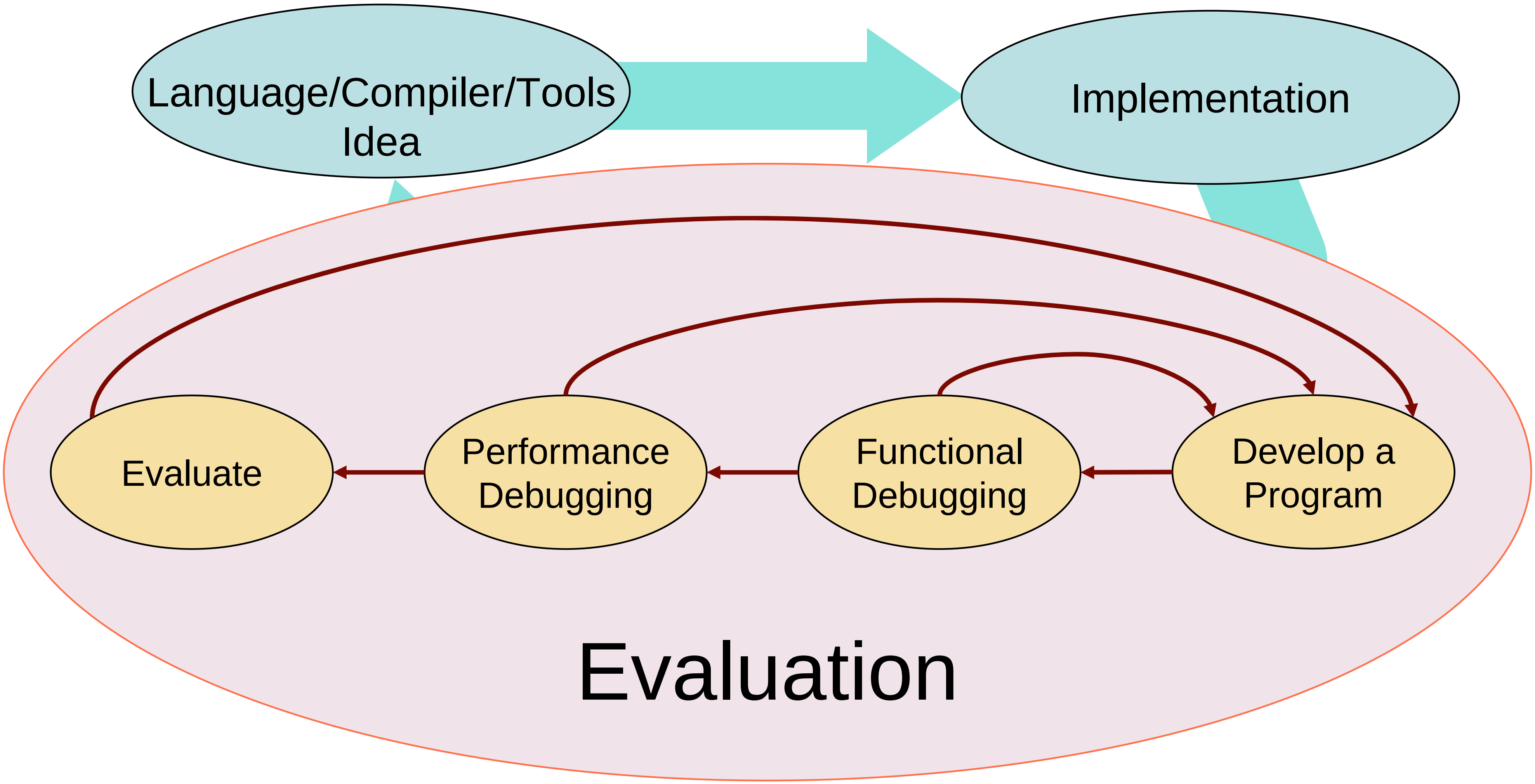
- Need tools to
  - Identify parallelism
  - Debug parallel code
  - Update and maintain parallel code
  - Stitch multiple domains together

- Need an “Eclipse platform for multicores”

Migrating 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

Facilitate Evaluation and Feedback for Rapid Evolution



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”??

Build and Mobilize the Community

- Bring the High Performance Languages/Compilers/Tools folks out of the woodwork!
  - Then: A few customers with the goal of performance at any cost.
  - Then: Had to compete with Moore's law
  - Now: Reasonable performance improvements for the masses

- Bring the folks who won the second crisis
  - Then: the focus is improving programmer productivity
  - Now: how to maintain performance in a multicore world
  - Now: If not solved, all the productivity gains will be lost!

- Get architects to listen to language/compiler people
  - Then: We don't need you, we can do everything in hardware
  - Then: Or here is a cool hardware feature, you figure out how to use it.
  - Now: Superscalars crashed and burned; cannot keep the status quo!
  - Now: Need to create a useable programming model

Conclusions

- Programming language research is a critical long-term investment
  - In the 1950s, the early background for the Simula language was funded by the Norwegian Defense Research Establishment
  - In 2002, the designers received the ACM Turing Award "for ideas fundamental to the emergence of object oriented programming."
- Compilers and Tools are also essential components
- Computer Architecture is at a cross roads
  - Once in a lifetime opportunity to redesign from scratch
  - How to use the Moore's law gains to improve the programmability?
- Switching to multicores without losing the gains in programmer productivity may be the Grandest of the Grand Challenges
  - Half a century of work $\Rightarrow$ still no winning solution
  - Will affect everyone!
- Need a Grand Partnership between the Government, Industry and Academia to solve this crisis!