StreamIt-A Programming Language for the Era of Multicores

Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory

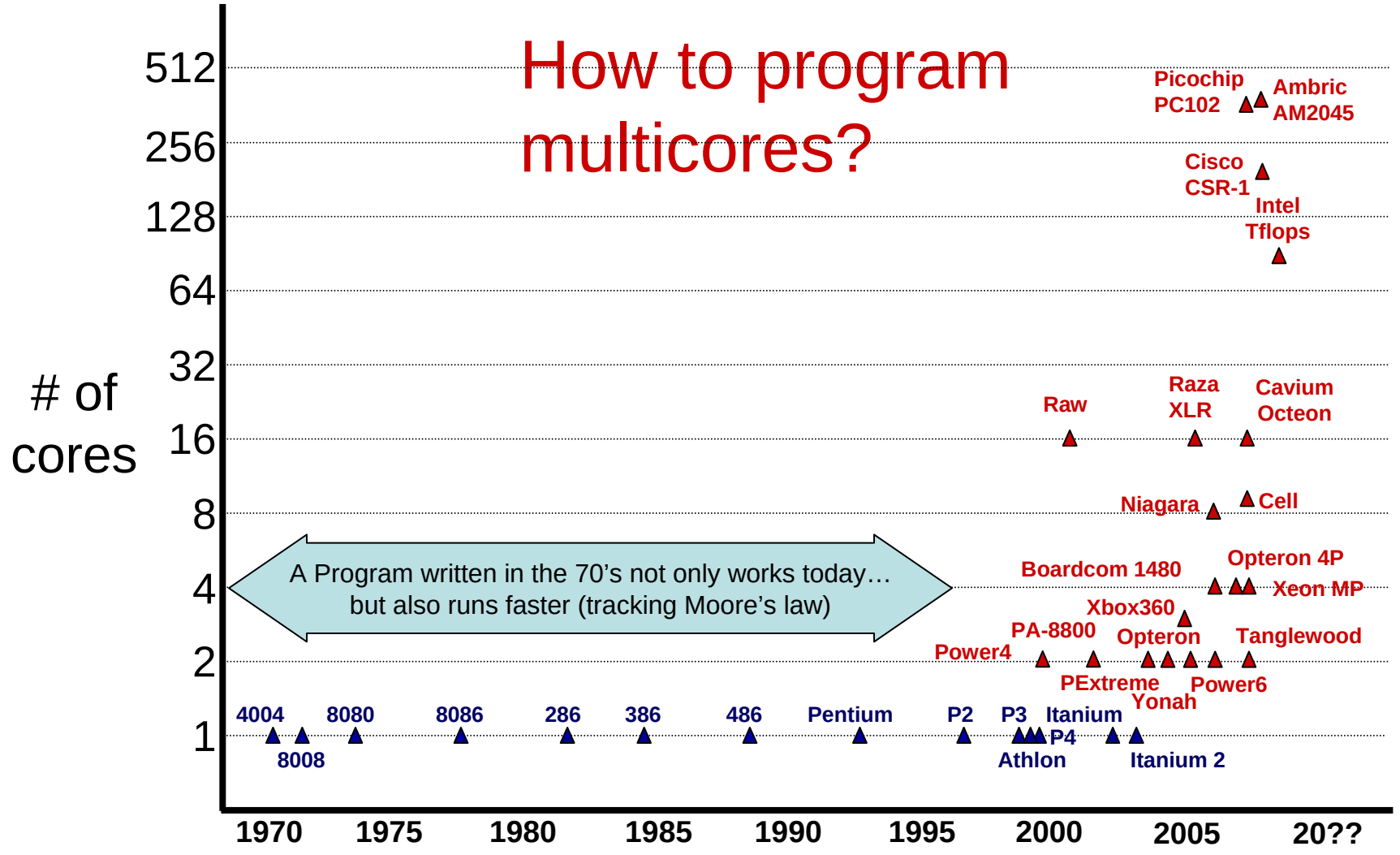
Massachusetts Institute of Technology

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
- This abstraction has provided a lot of freedom for Joe

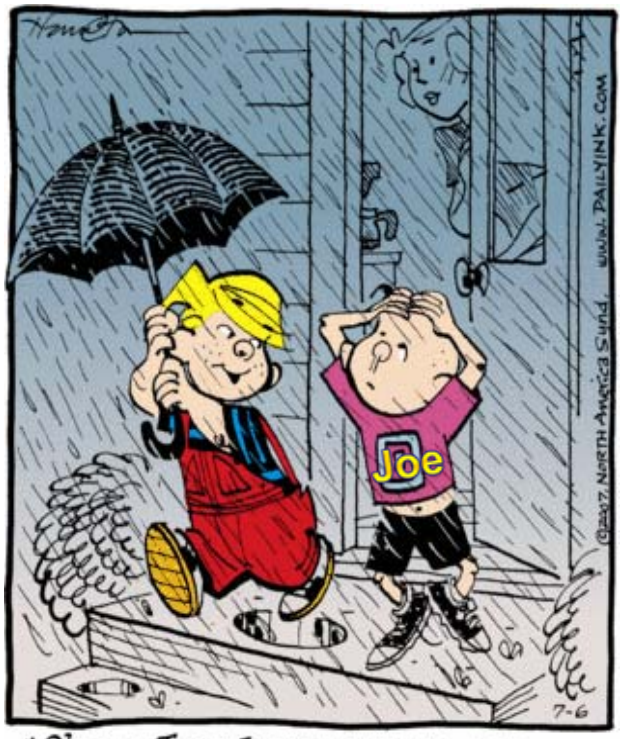
- Parallel Programming is only practiced by a few experts

Where Joe is heading



Joe the Parallel Programmer

- Moore's law is not bringing anymore performance gains
- If Joe needs performance he has to deal with multicores
  - Joe has to deal with performance
  - Joe has to deal with parallelism



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: Who can help Joe?

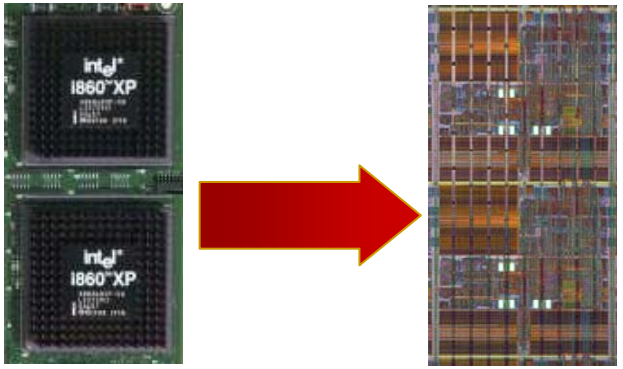
1. Advances in Computer Architecture
2. Novel Programming Models and Languages
3. Aggressive Compilers

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 poor Joe?

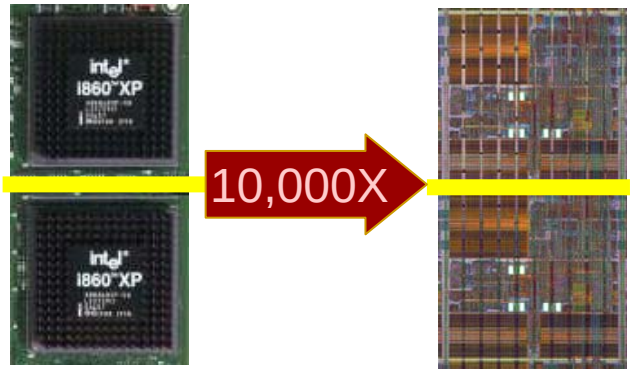
Novel Opportunities in Multicores

- 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

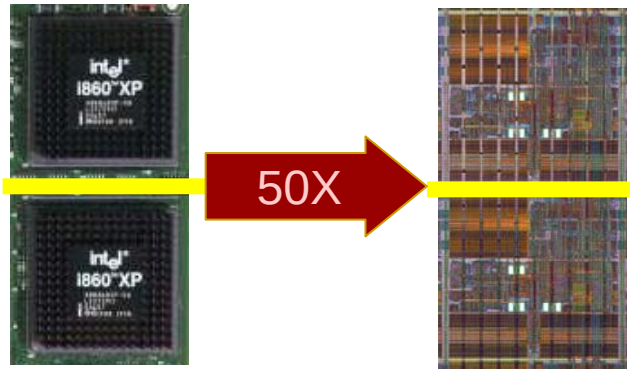
- How much data can be communicated between two cores?
- What changed?
  - Number of Wires
  - Clock rate
  - Multiplexing
- Impact on programming model?
  - Massive data exchange is possible
  - Data movement is not the bottleneck → processor affinity not that important



32 Giga bits/sec ~300 Tera bits/sec

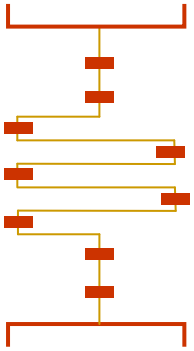
Communication Latency

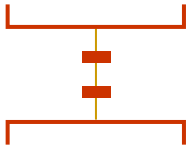
- How long does it take for a round trip communication?
- What changed?
  - Length of wire
  - Pipeline stages
- Impact on programming model?
  - Ultra-fast synchronization
  - Can run real-time apps on multiple cores



~200 Cycles

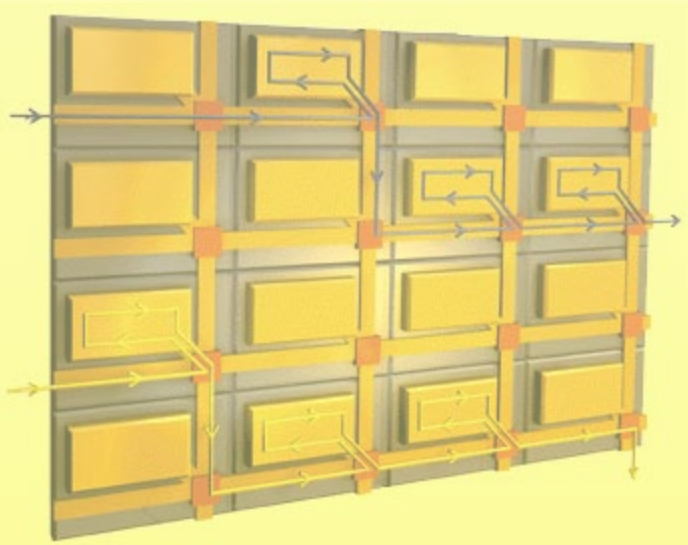
~4 cycles





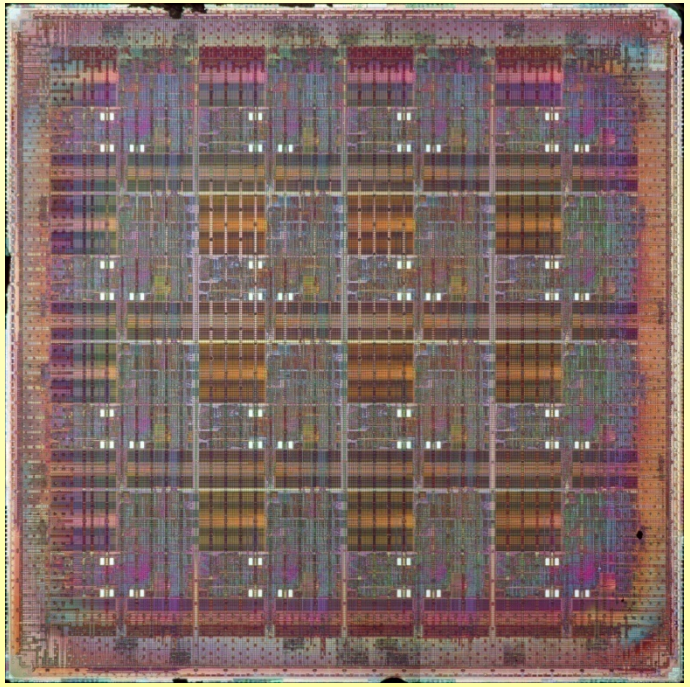
Architectural Innovations

The Raw Experience

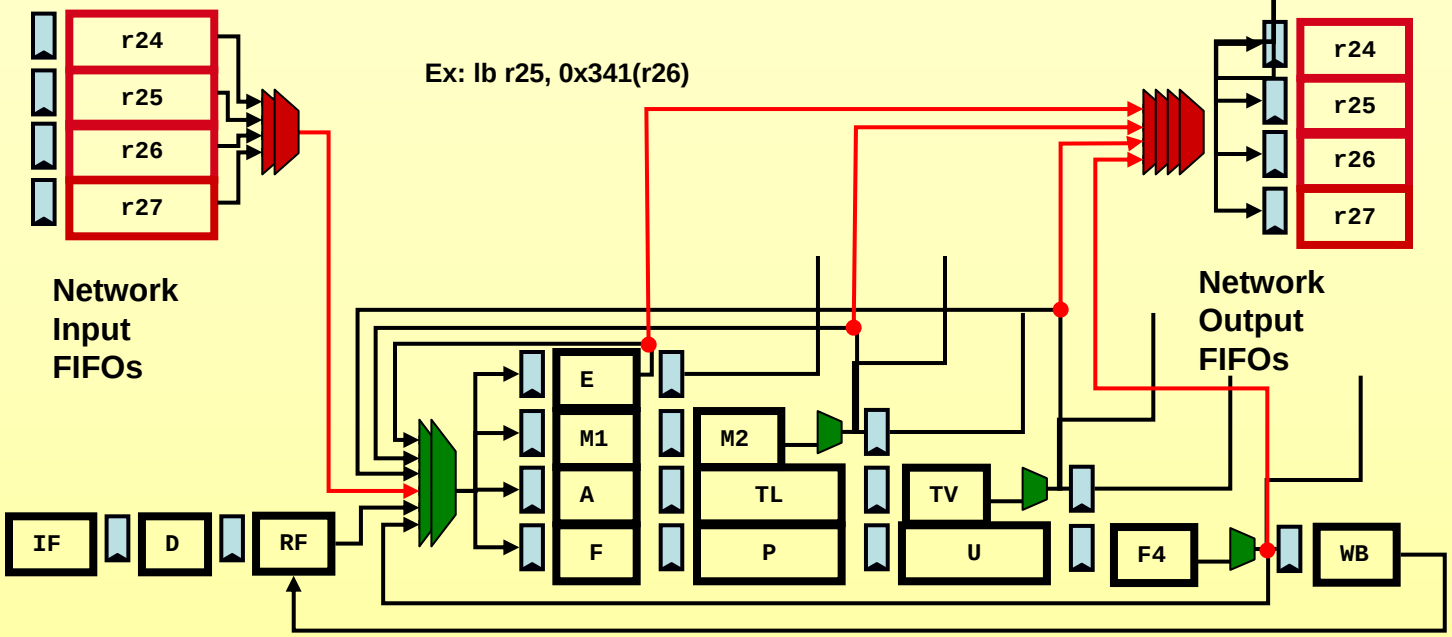


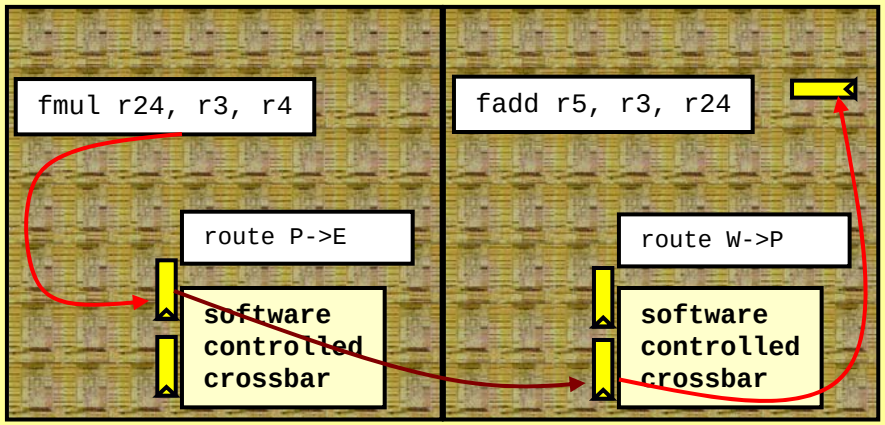
The MIT Raw Processor

- 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



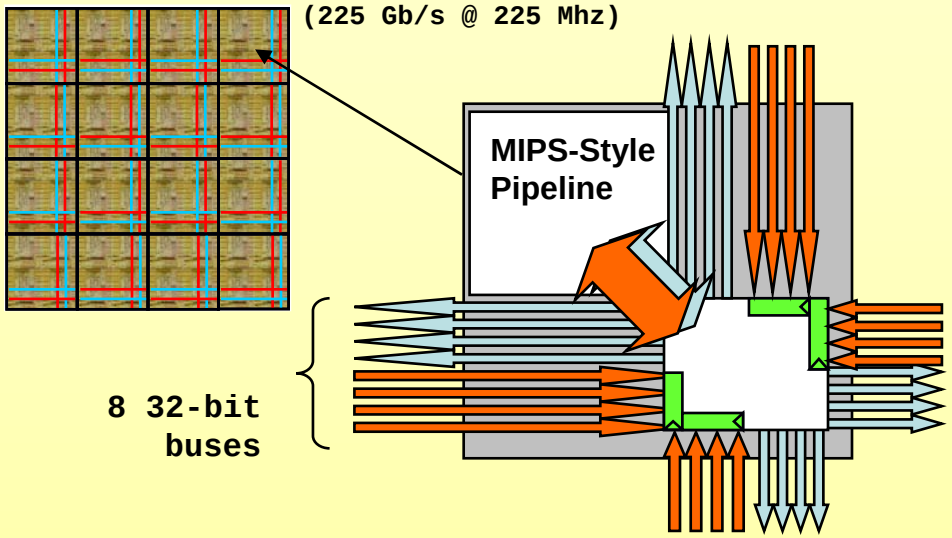
Raw's networks are tightly coupled into the bypass paths

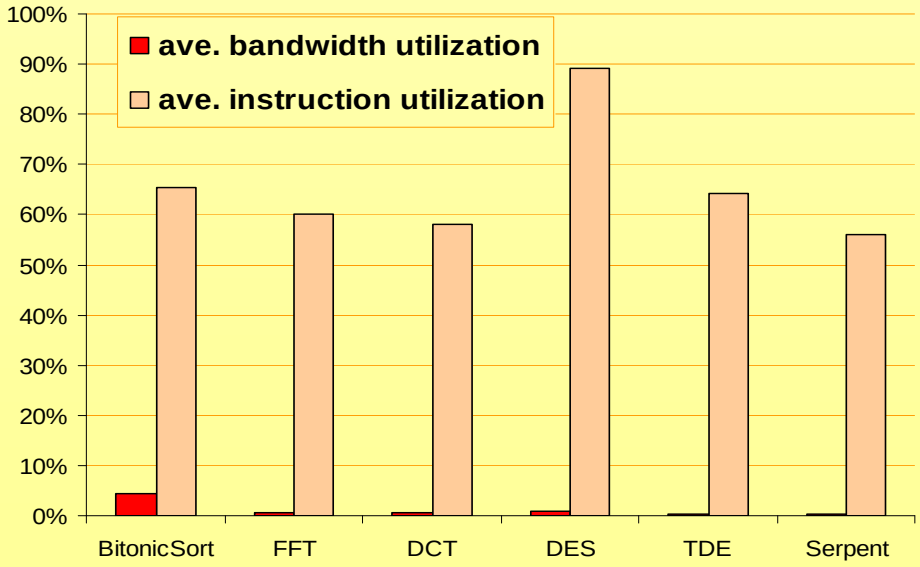




Raw Networks is Rarely the Bottleneck

- 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: Who can help Joe?

1. Advances in Computer Architecture
2. Novel Programming Models and Languages
3. Aggressive Compilers

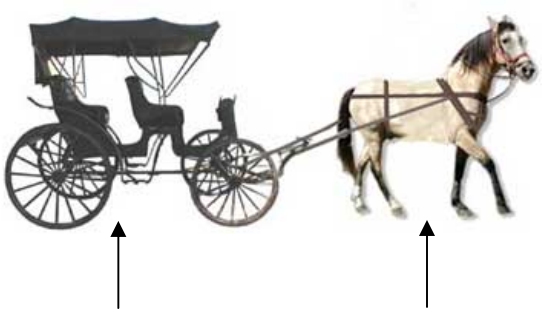
Why New Programming Models and Languages?

- 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 Joe 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



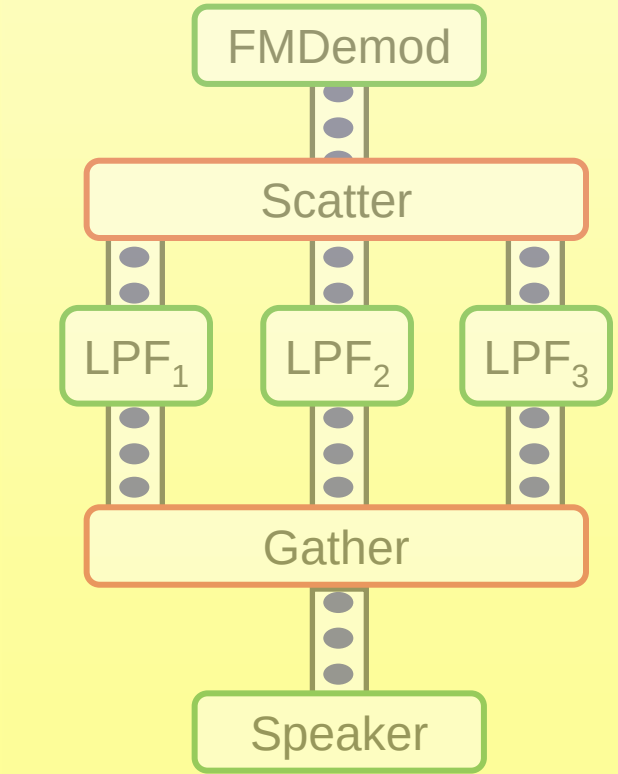
C ⇔ von-Neumann machine

Modern architecture

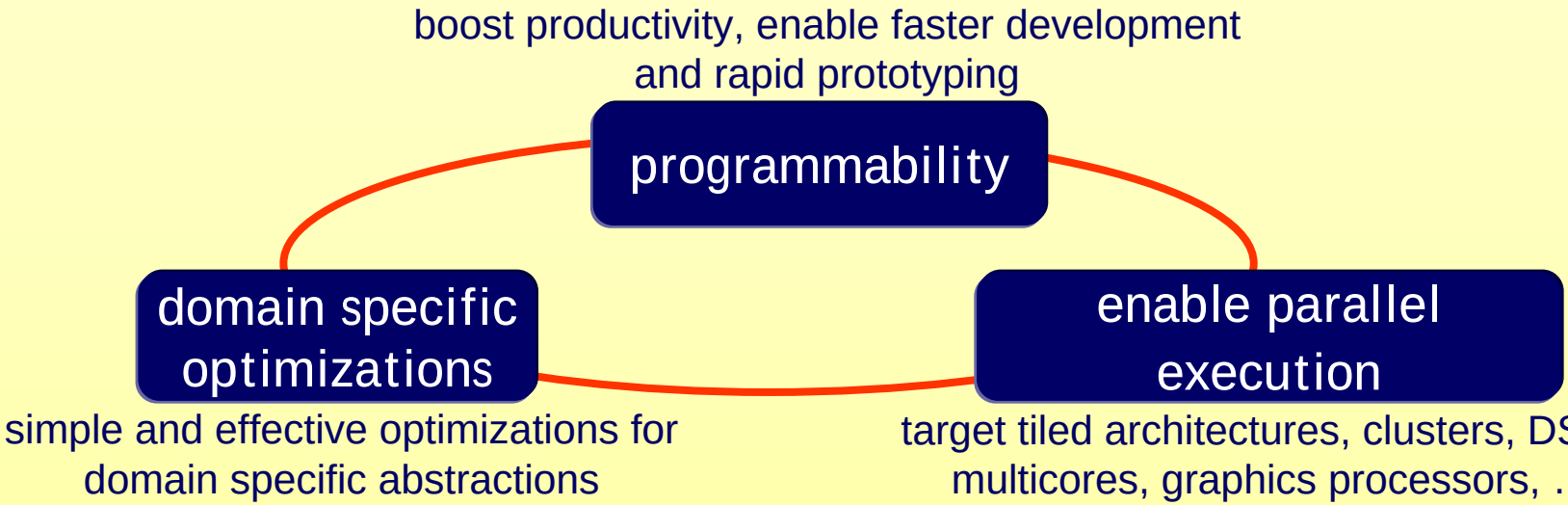
- Two choices:
  - Bend over backwards to support old languages like C/C++
  - Develop parallel architectures that are hard to program

Compiler-Aware Language Design

The StreamIt Experience

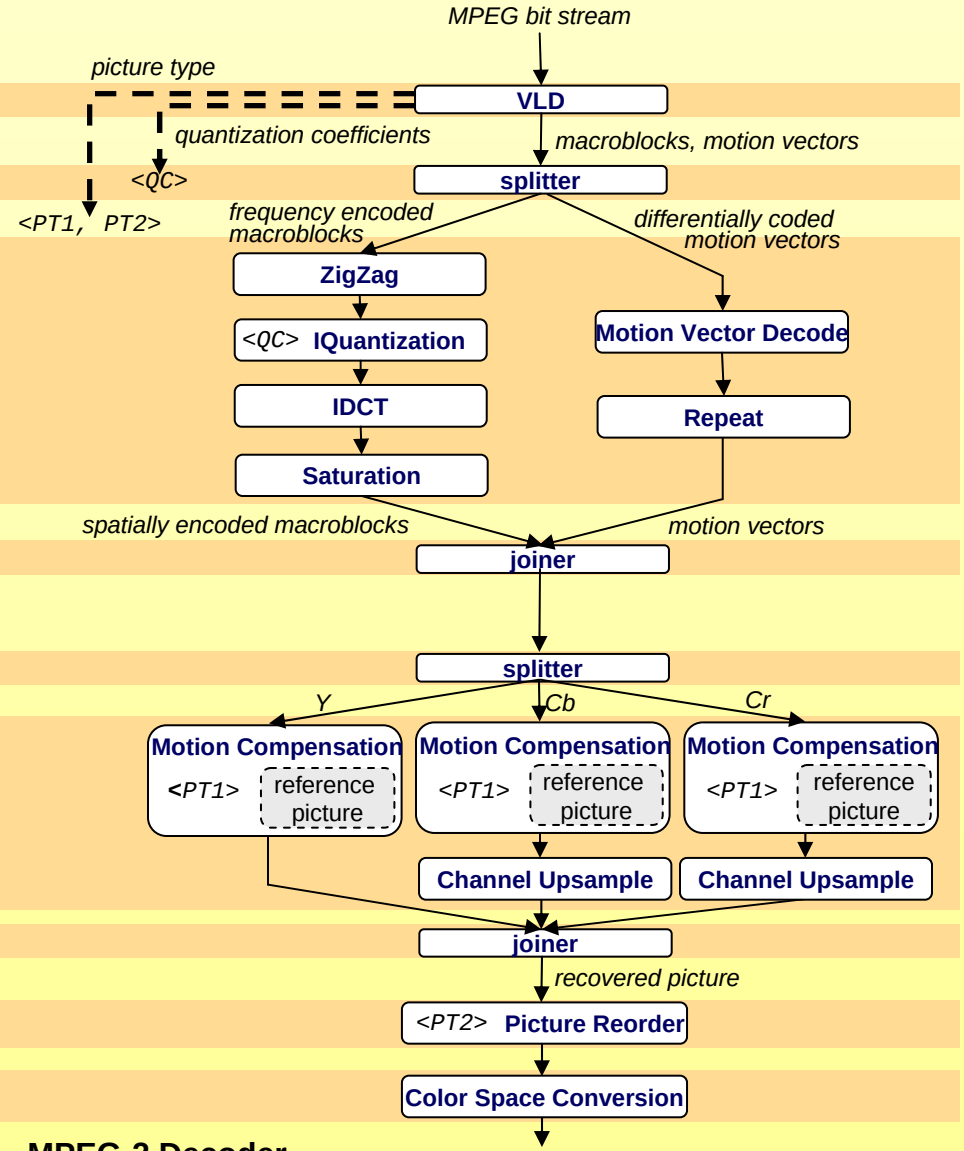


Is there a win-win situation?



- 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

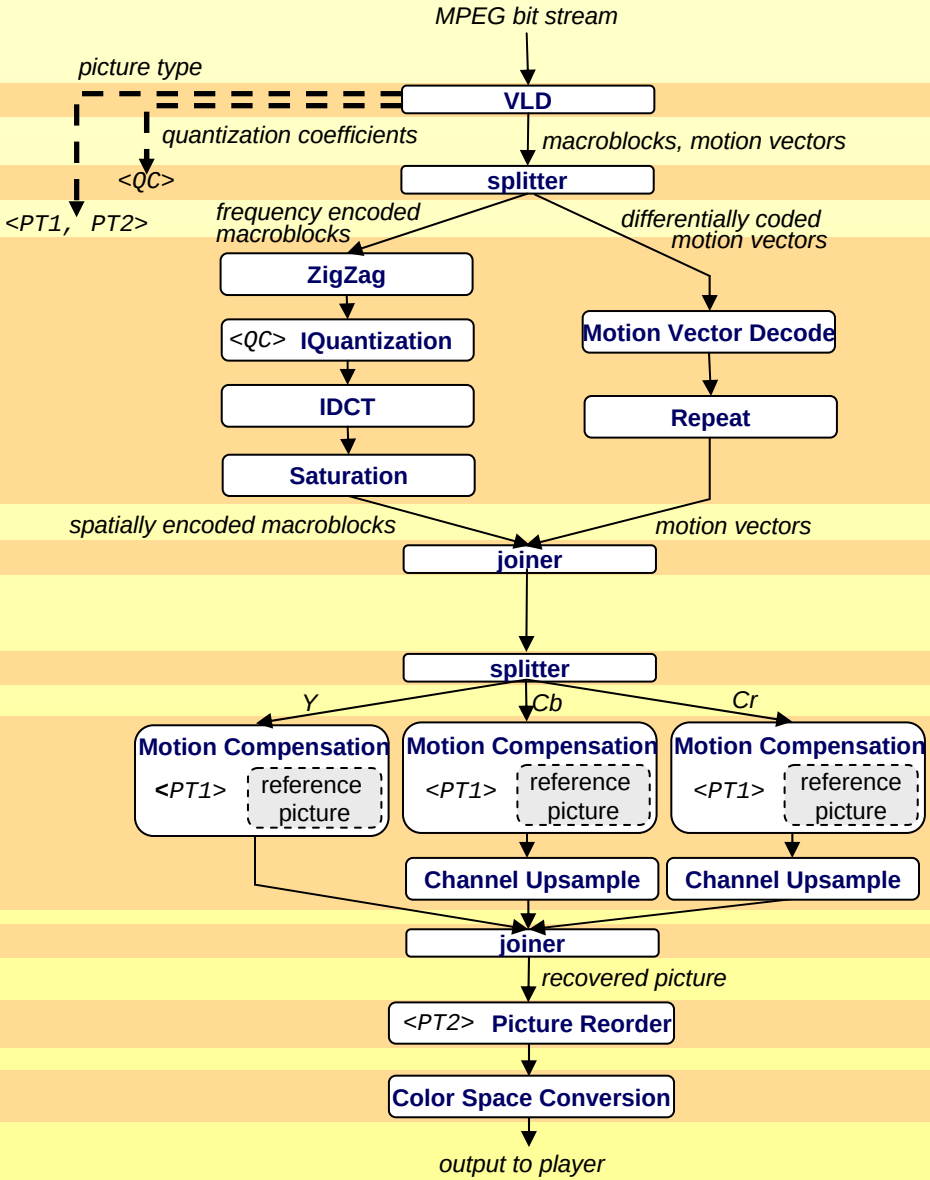
Streaming Application Abstraction



MPEG-2 Decoder

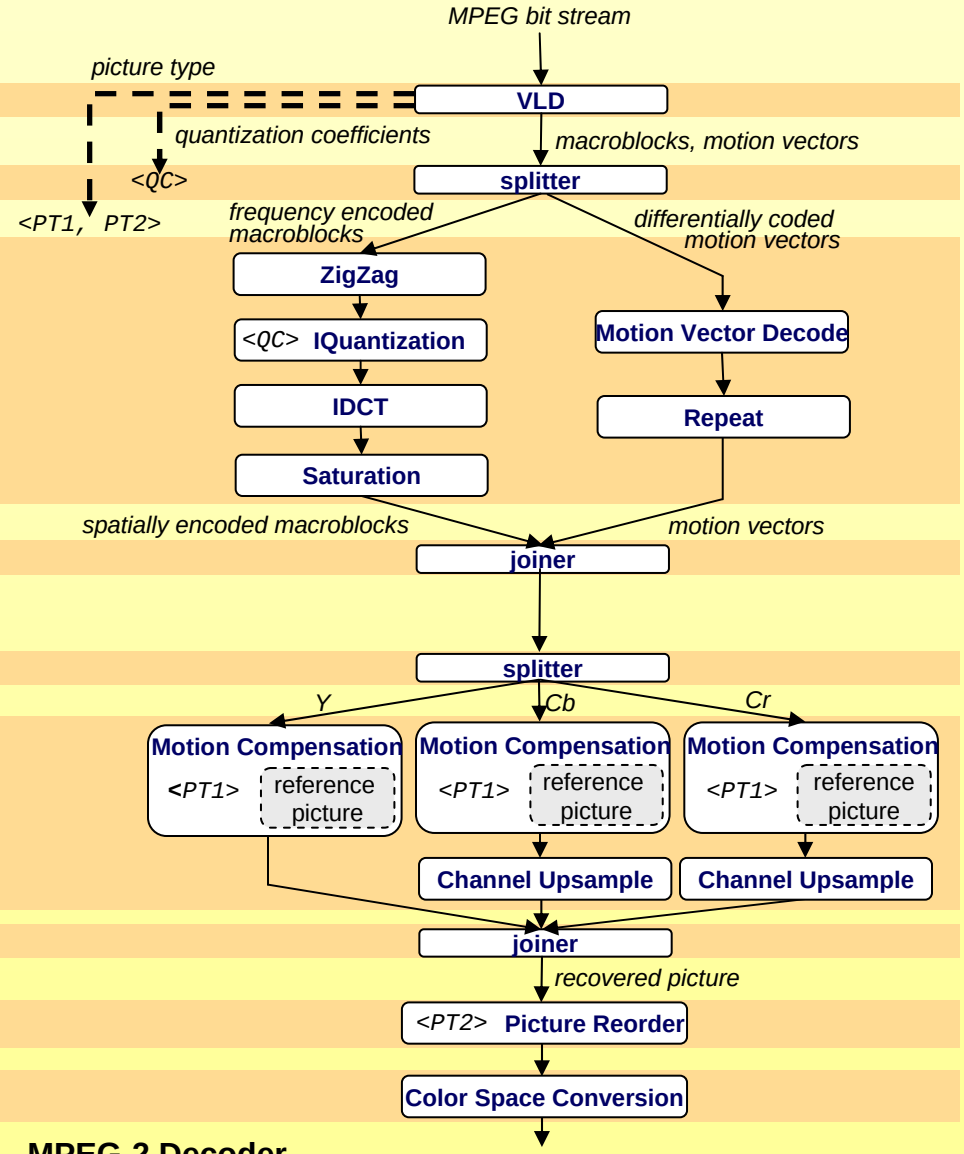
- Structured block level diagram describes computation and flow of data
- Conceptually easy to understand
  - Clean abstraction of functionality
- Mapping to C (sequentialization) destroys this simple view

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);
```

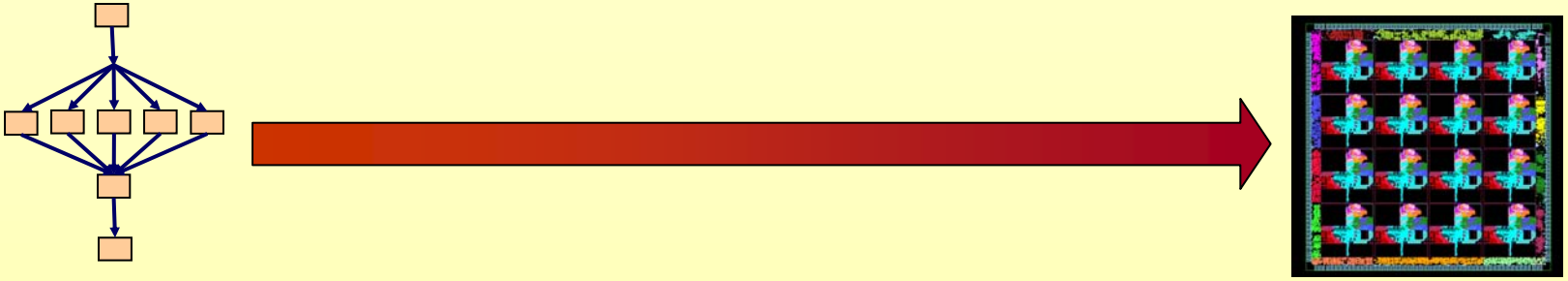
StreamIt Compiler Extracts Parallelism



MPEG-2 Decoder

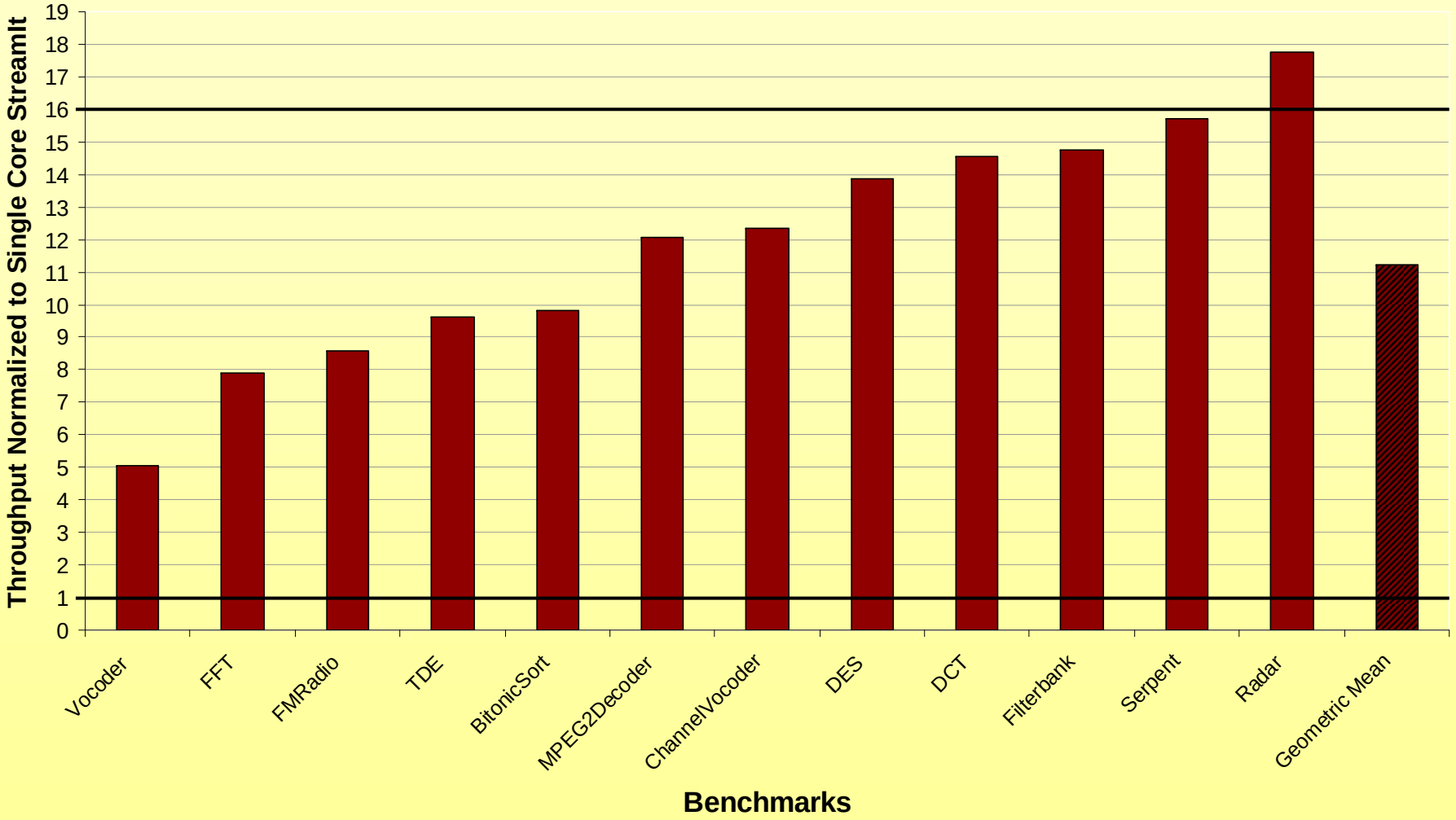
- Task Parallelism
  - Thread (fork/join) parallelism
  - Parallelism explicit in algorithm
  - Between filters *without* producer/consumer relationship
- Data Parallelism
  - Data parallel loop (**forall**)
  - Between iterations of a *stateless* filter
  - Can't parallelize filters with state
- Pipeline 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: Who can help Joe?

1. Advances in Computer Architecture
2. Novel Programming Models and Languages
3. Aggressive Compilers

Parallel Programming Models

- Current models are too primitive
  - Akin to assembly language programming
- We need new Parallel Programming Models that…
  - does not require any non-intuitive reorganization of data or code
  - will completely eliminate hard problems such as race conditions and deadlocks
    - akin to the elimination of memory bugs in Java
  - can inform the programmer if they have done something illegal
    - akin to a type system or runtime null-pointer checks
  - can take advantage of available parallelism without explicit user intervention
    - akin to virtual memory where the programmer is oblivious to physical size
  - the programmer can be oblivious to parallelism and performance issues
    - akin to ILP compilation to VLIW machine

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

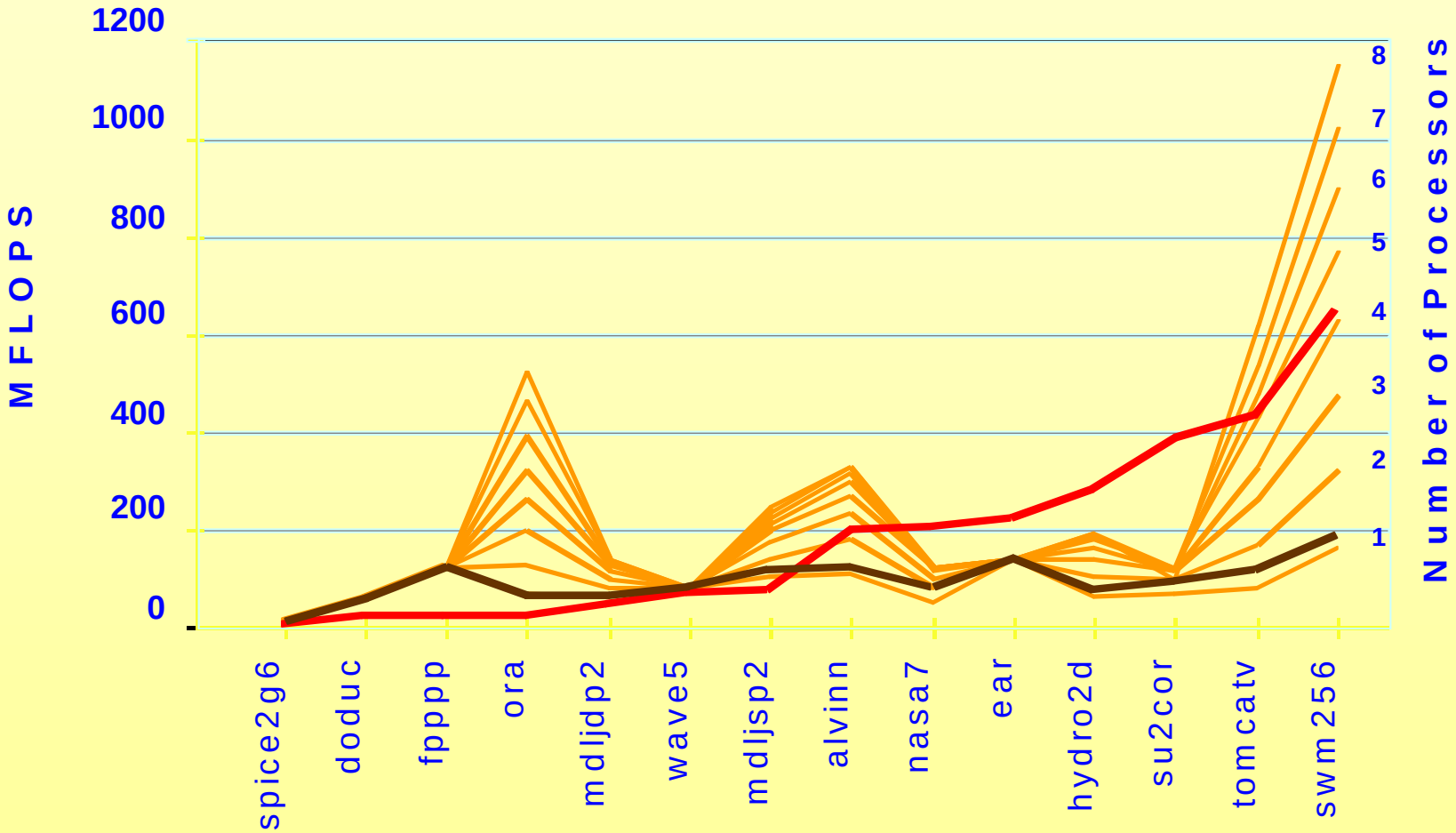
Parallelizing Compilers

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)

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!