Lecture 6
Instruction Level Parallelism (4)

EEC 171 Parallel Architectures

~~John Owens~~ Venkatesh Akella

UC Davis

Credits

- © John Owens / UC Davis 2007–17.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–7, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

Outline

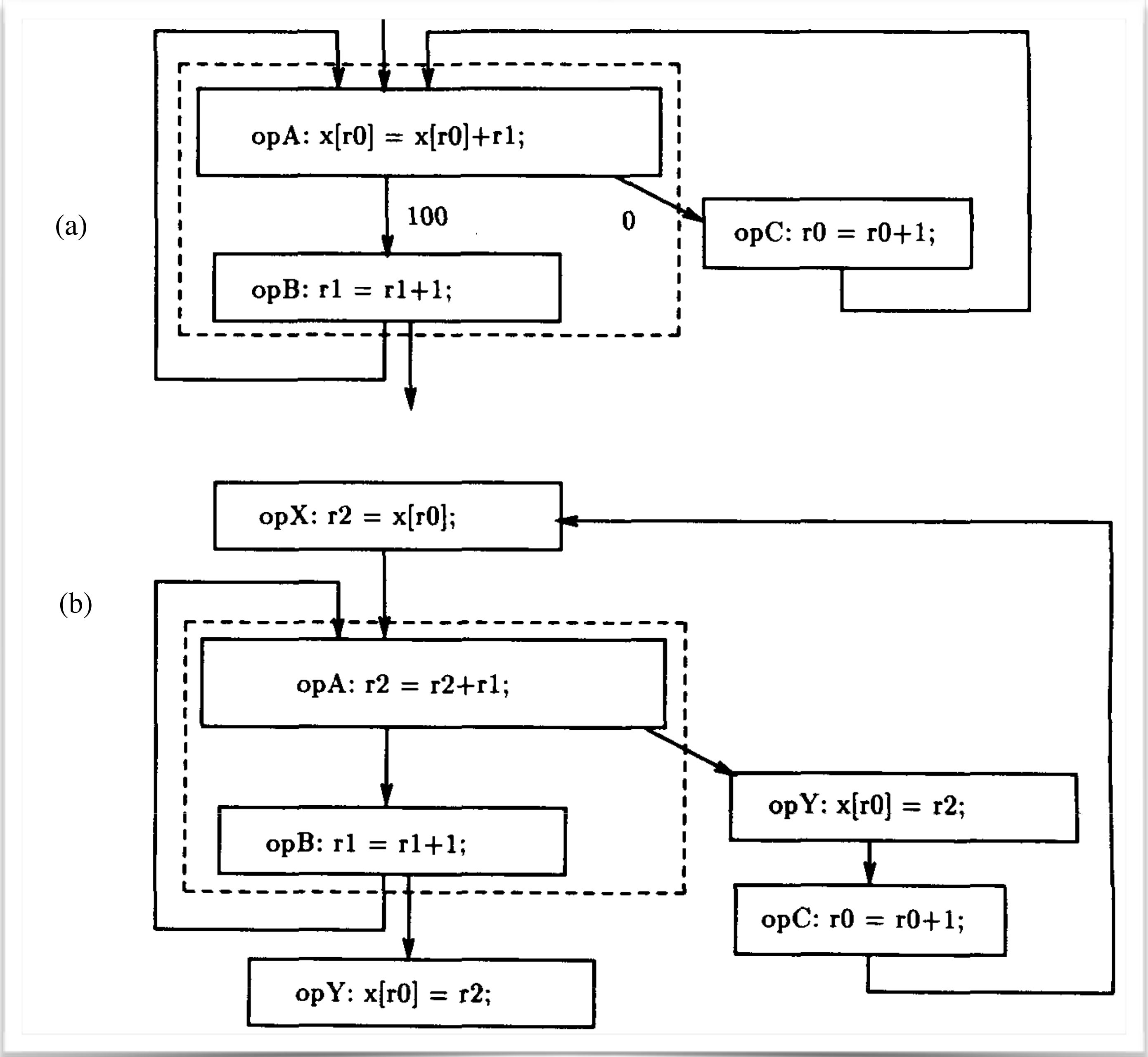
- Trace scheduling & trace caches
- Pentium 4
- Intel tips for performance
- Transmeta approach—alternate VLIW strategy
- Limits to ILP

Trace scheduling

- Problem: Can’t find enough ILP within single block
- Solution: Pretend blocks are bigger!
- Problem: But blocks aren’t bigger! Blocks are separated by branches.
- Solution 1: Let hardware speculate (branch prediction)
- Solution 2: Let software make its best guess based on history (trace scheduling) (historically: predominantly used in VLIW machines)
    - … and then implement this in hardware (trace cache, Pentium 4)

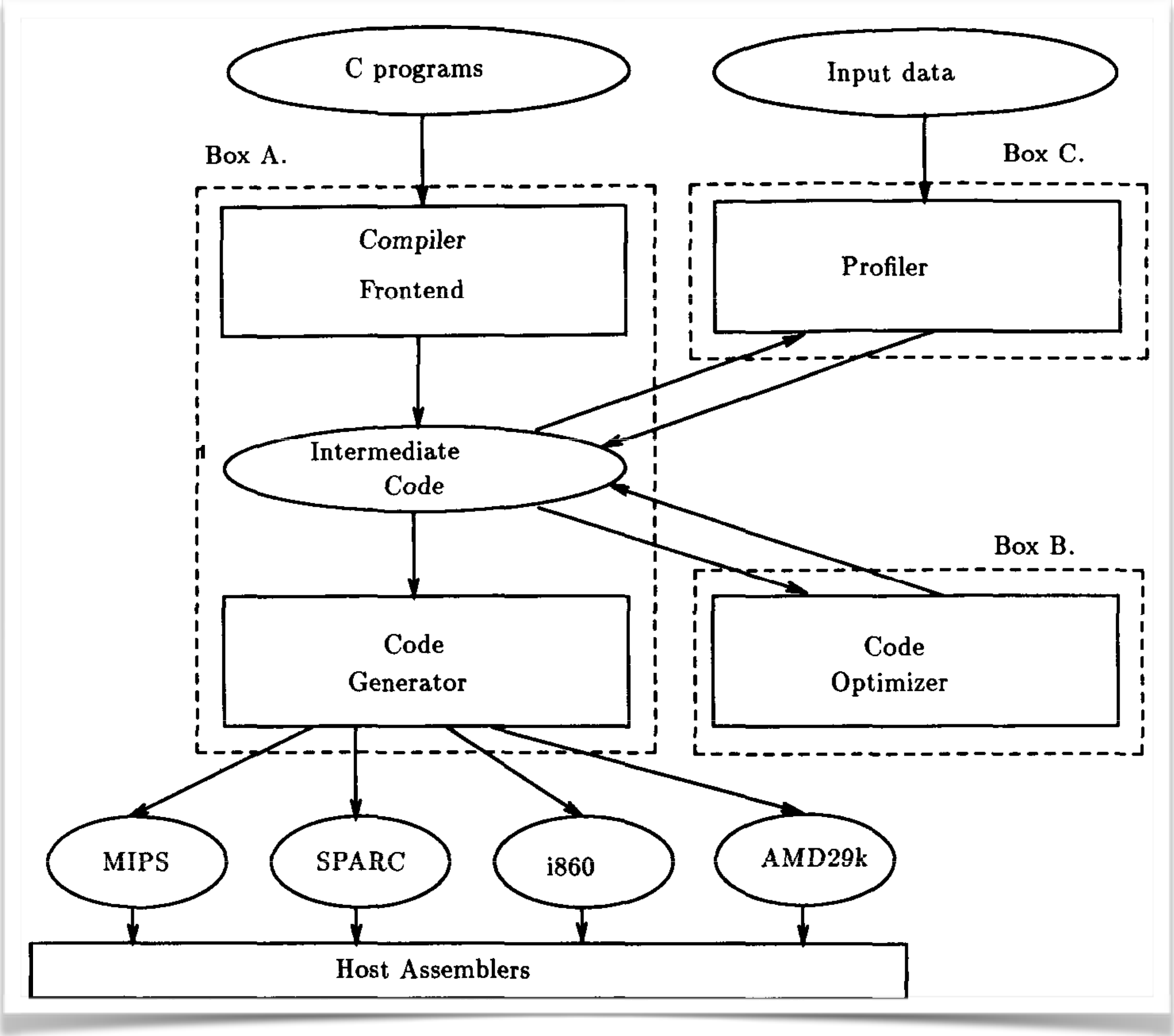
Example (SW)

- Original loop allows us to increment either r0 or r1: x[r0]=x[r0]+r1
- Profile says incrementing r1 is much more common
- Optimize for that case

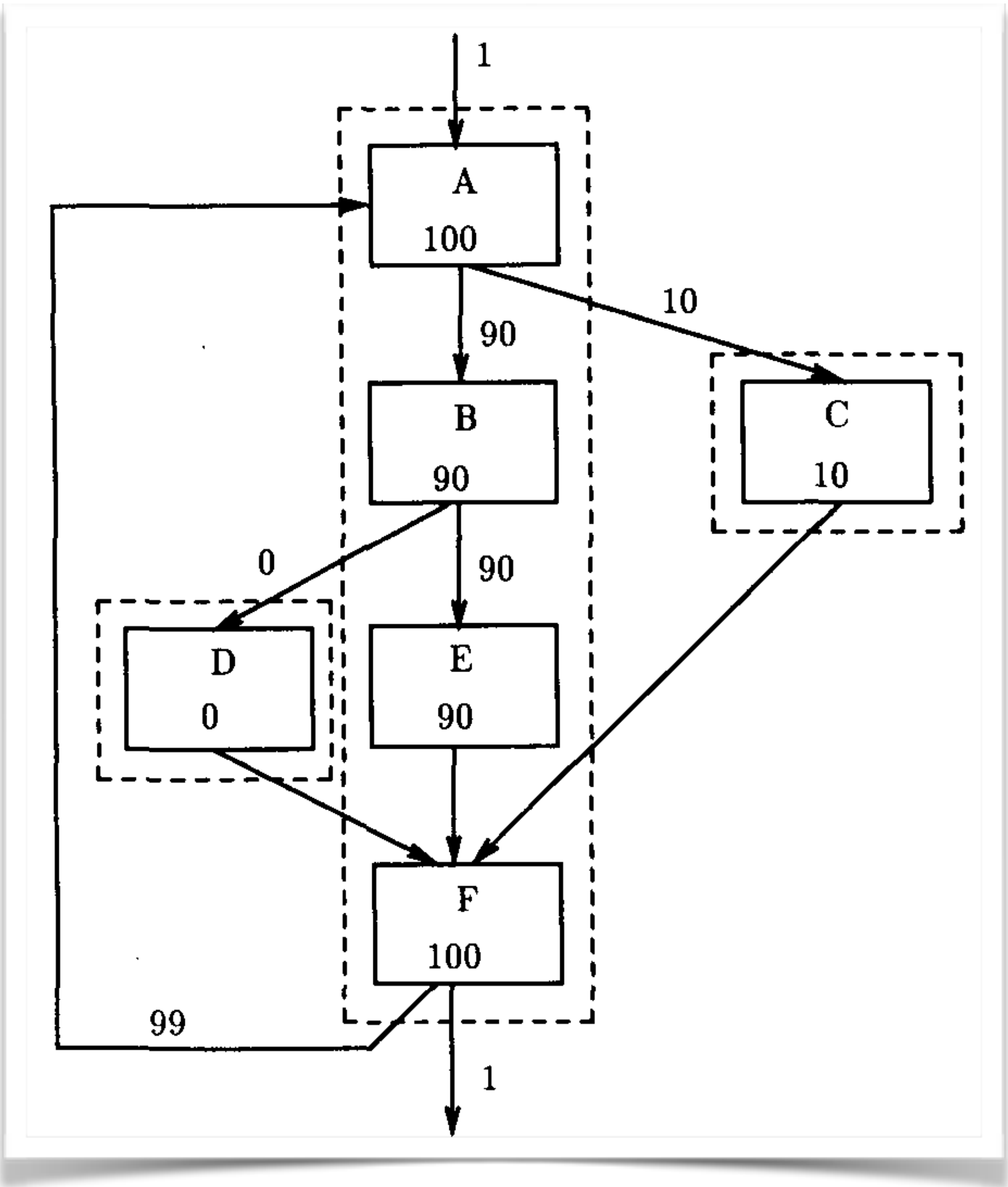


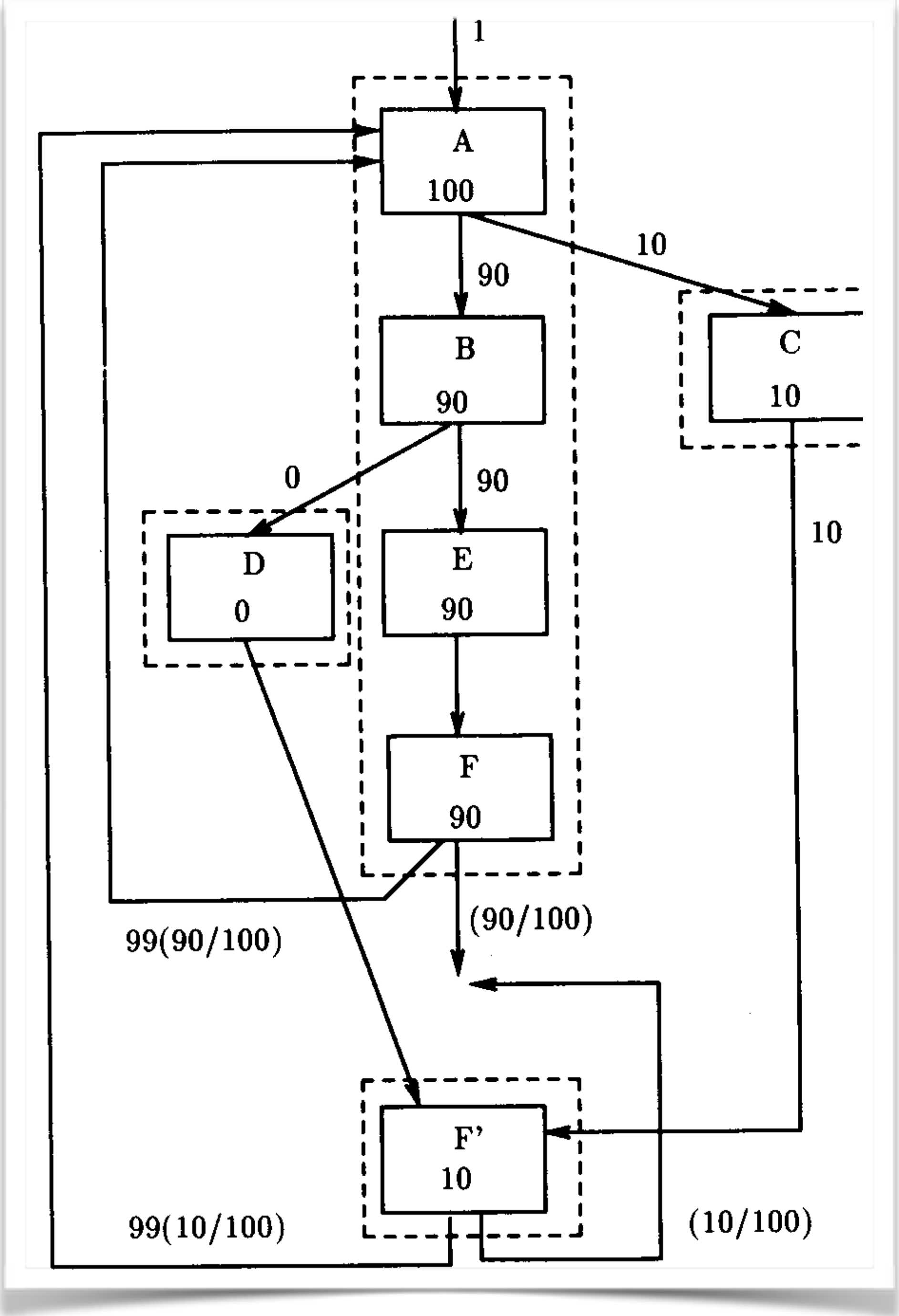
Chang et al. SPE Dec. 1991

Structure of Compiler



Bigger example





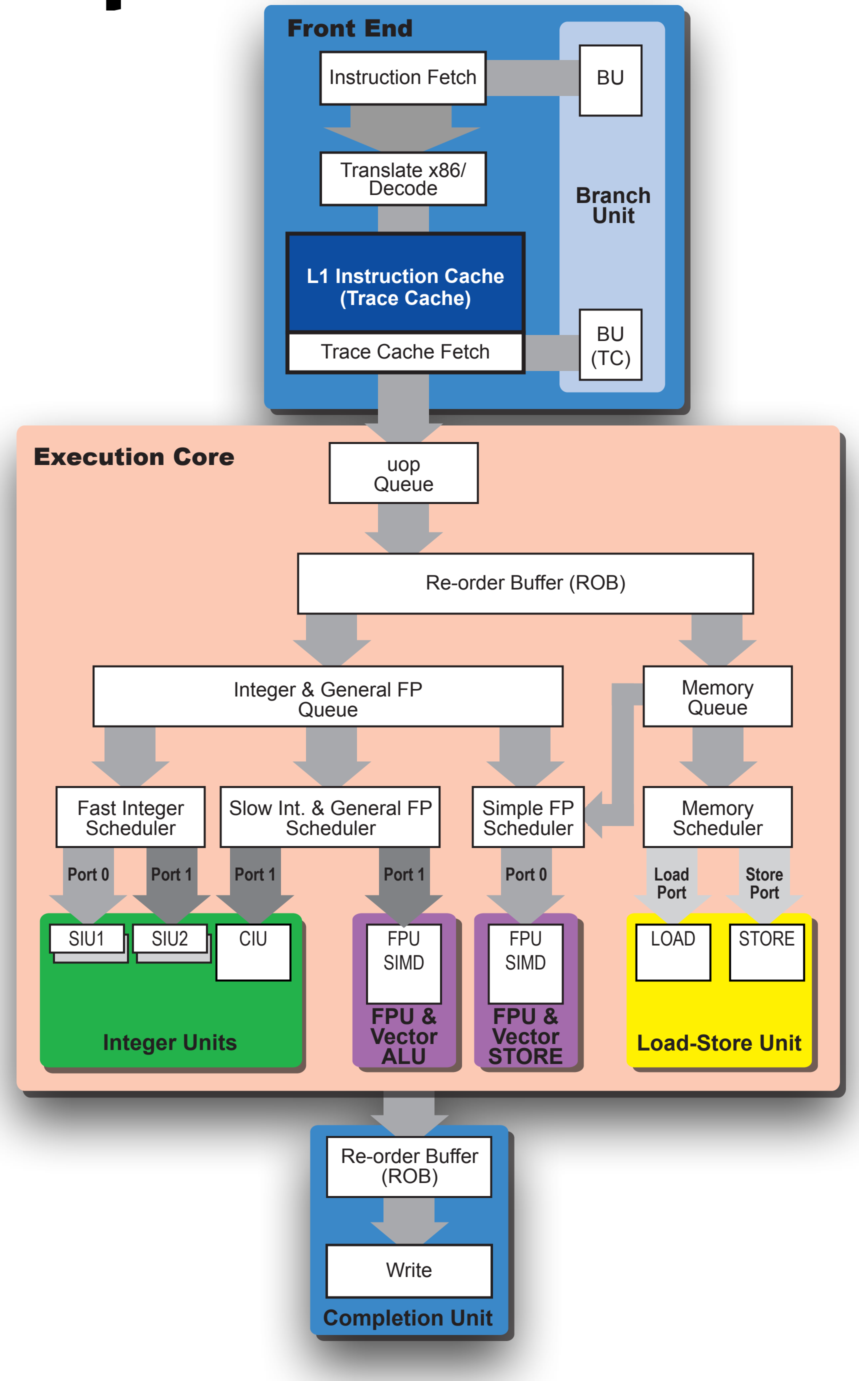
Chang et al. results

- Over SPEC and other benchmarks:
- Profile vs. global techniques: 15% better
  - MIPS 04 vs. global: 4% worse
  - gnu.0 vs. global: 12% worse
- Code size: Profile vs. global: 7% bigger

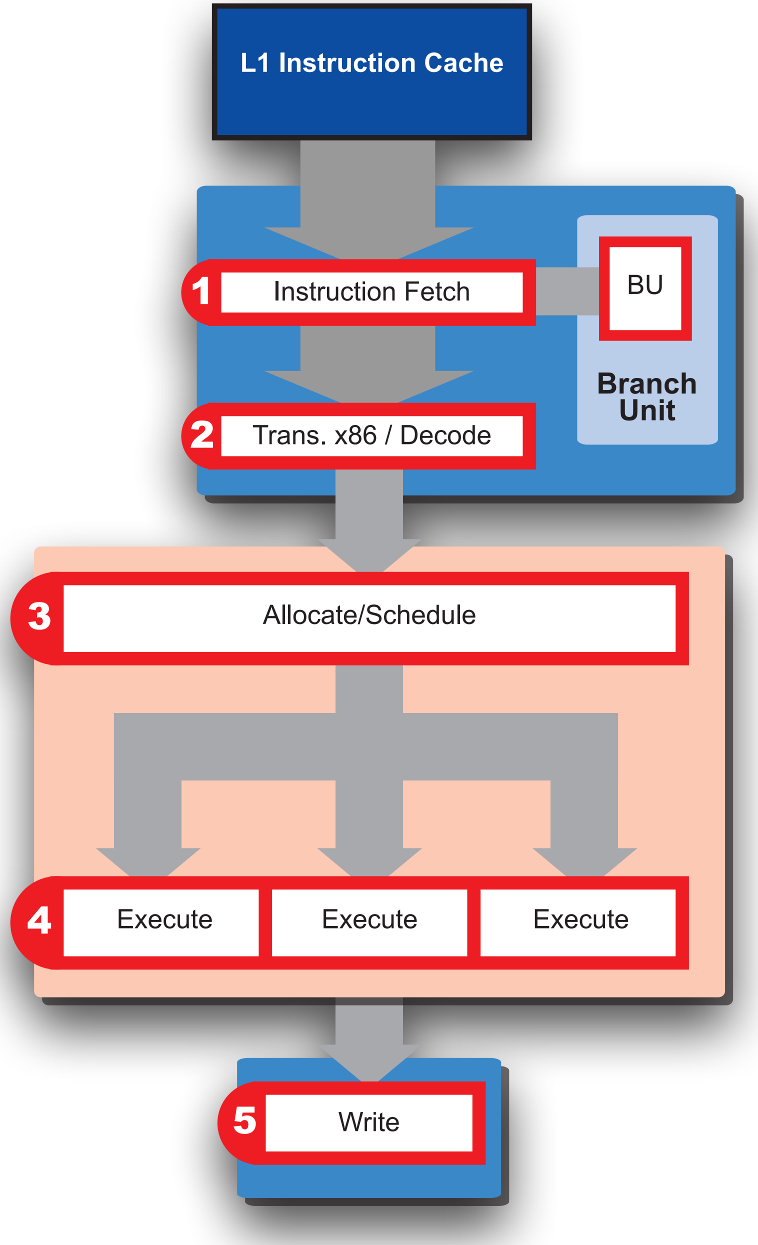
Trace Cache

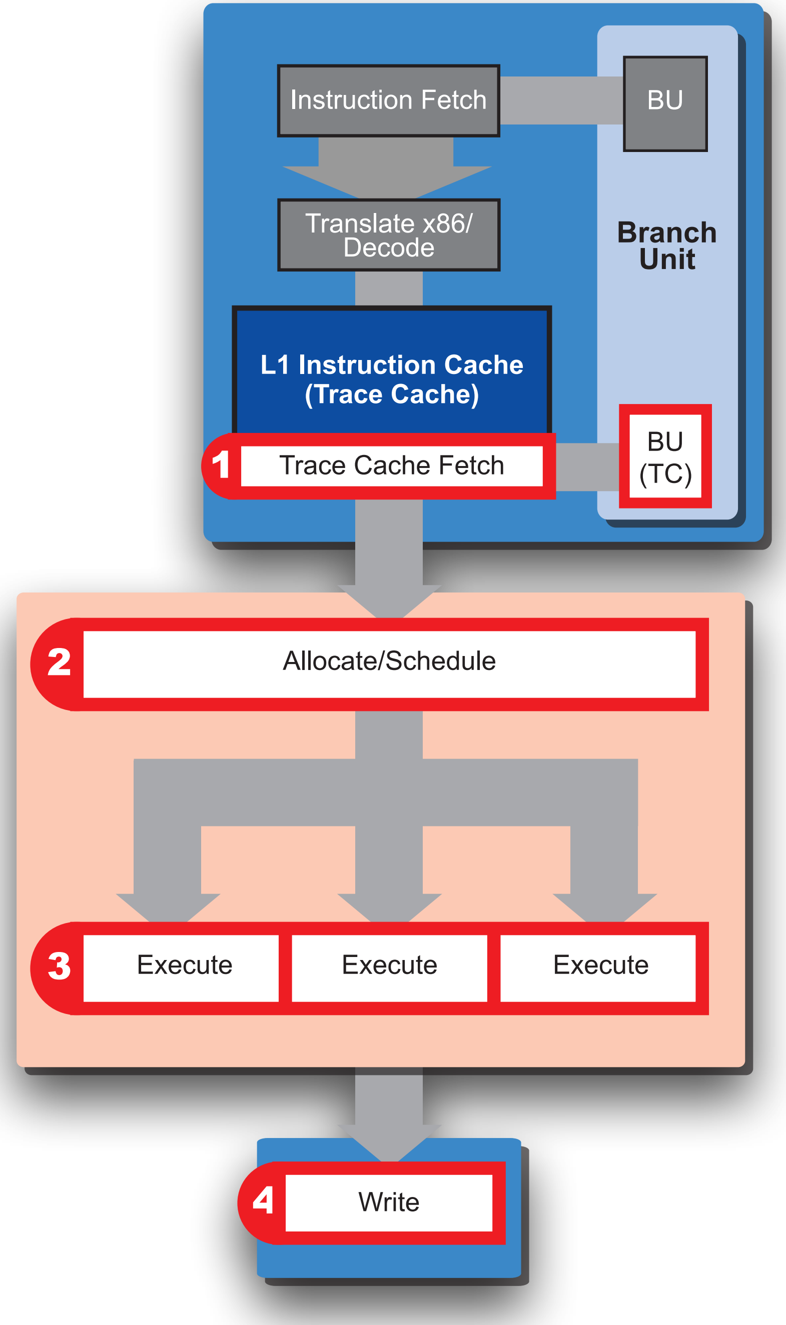
- Trace techniques are useful in software
- How about in hardware?
- Addresses 2 problems in hardware:
  - How to find more instruction level parallelism?
  - How to avoid translation from x86 to microops?
- Answer: Trace cache in Pentium 4

Pentium 4 Architecture



Pentium Pro vs. P4





Trace Cache

- Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
- Built-in branch predictor
- Cache the micro-ops vs. x86 instructions
- Decode/translate from x86 to micro-ops on trace cache miss
- P4 trace cache has unspecified size—Intel says roughly 12k μops, equivalent to 16–18 kB icache

Trace Cache Pros & Cons

- + Better utilize long blocks (don’t exit in middle of block, don’t enter at label in middle of block)
- – Complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size
  - Compiler is encouraged to produce static code like this
- – Instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

Trace Cache Operation

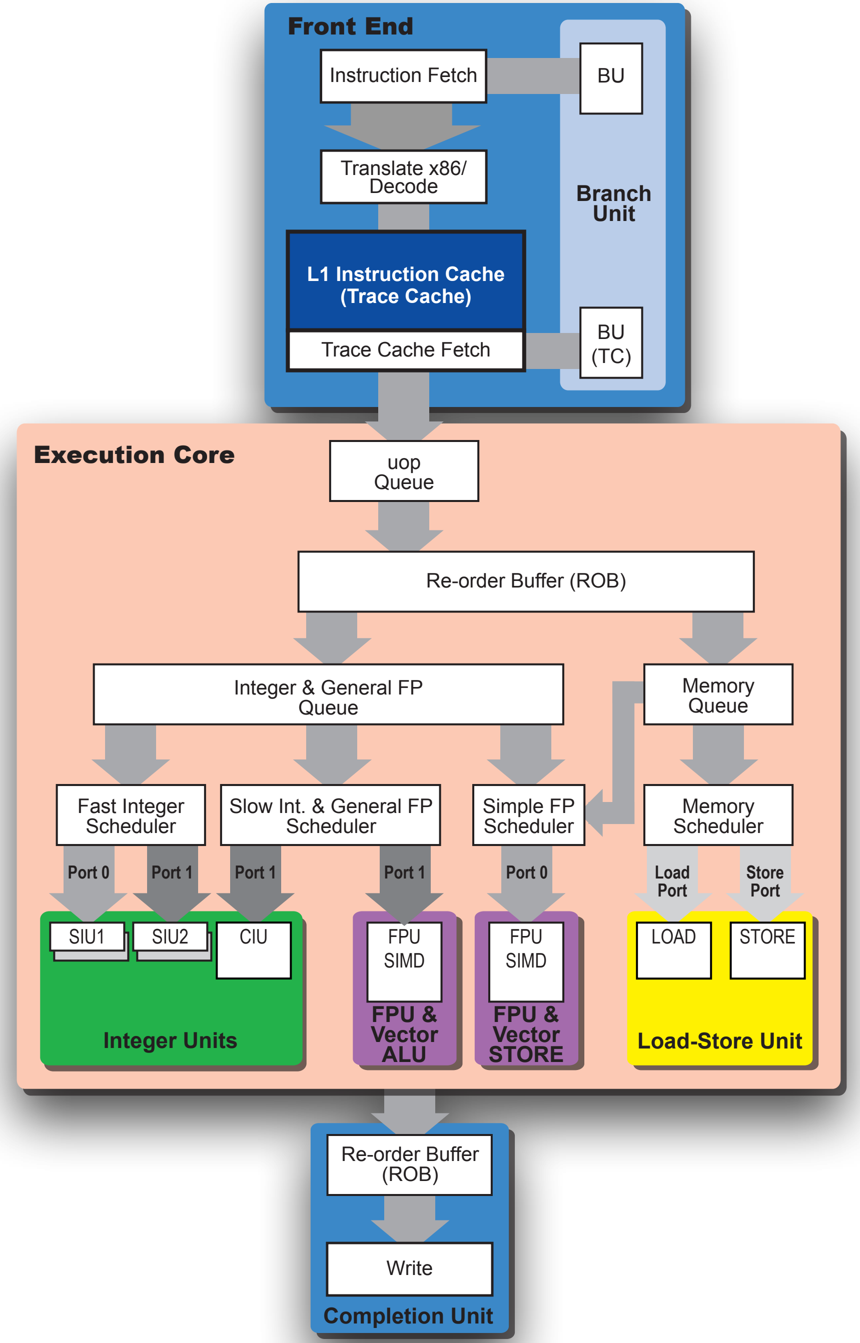
- Two modes:
  - Execute mode
    - Up to 3 uops/cycle
    - No translation or decode (saves 8 cycles)
    - Only cache misses kick in front end
  - Build mode
    - Front end fetches x86 code from L2
    - Translate, build trace segment, put into L1
    - How do we prevent giant x86 instructions from polluting trace cache? Switch control over to ROM

P4 Trace Cache Advantages

- Saves cost of branch prediction
  - Even successfully predicted branches likely put a bubble into the pipeline
  - P4 has *20* cycle minimum misprediction penalty, more if cache miss
- In standard Intel pipelines, instruction fetch of a cache line goes up to a branch and stops
  - Trace cache lines contain speculative ops after branches —no waste

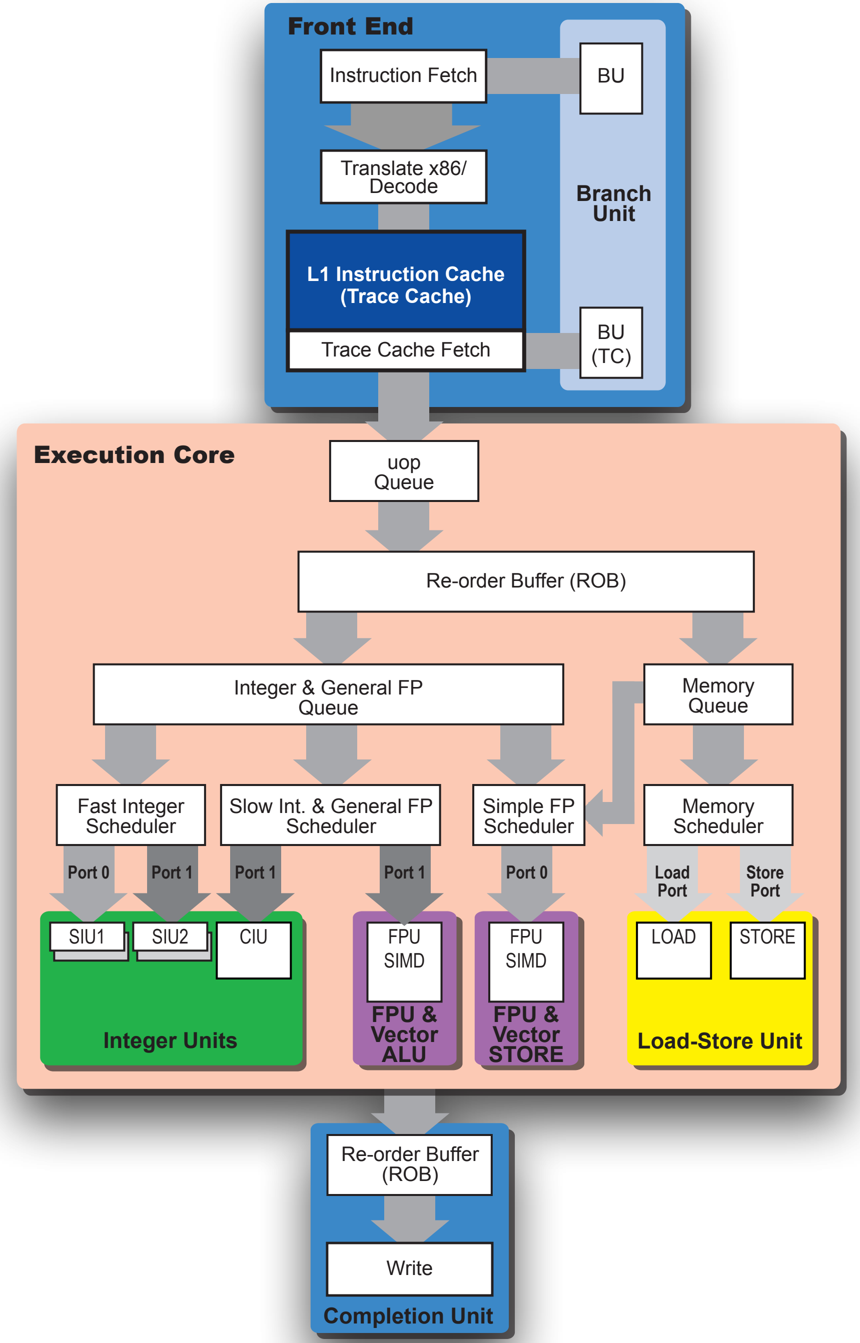
Pentium 4 Pipeline

- 1–2: Trace cache next instruction pointer
- 3–4: Trace cache fetch
- 5: Drive
  - Up to 3 μops now sent into μop queue
- 6–8: Allocate and rename
  - Up to 3 uops/cycle move from bottom of uop queue into ROB/rename registers. Rename: 8->128 F,G,V regs
- 9: Queue (int/fp, memory queues; enter in order but can leave out of order)



Pentium 4 Pipeline

- 10–12: Schedule
  - 8–12 entry mini-μop queue, arbitrates for one of 4 issue ports
- 13–14: Issue
  - Up to 6 μops/cycle thru 4 ports
  - 2 execution ports are double-clocked
- 15–16: Register files
- 17: Execute
- 18: Flags
- 19: Branch check
- 20: Drive
- 21+ : Complete and commit

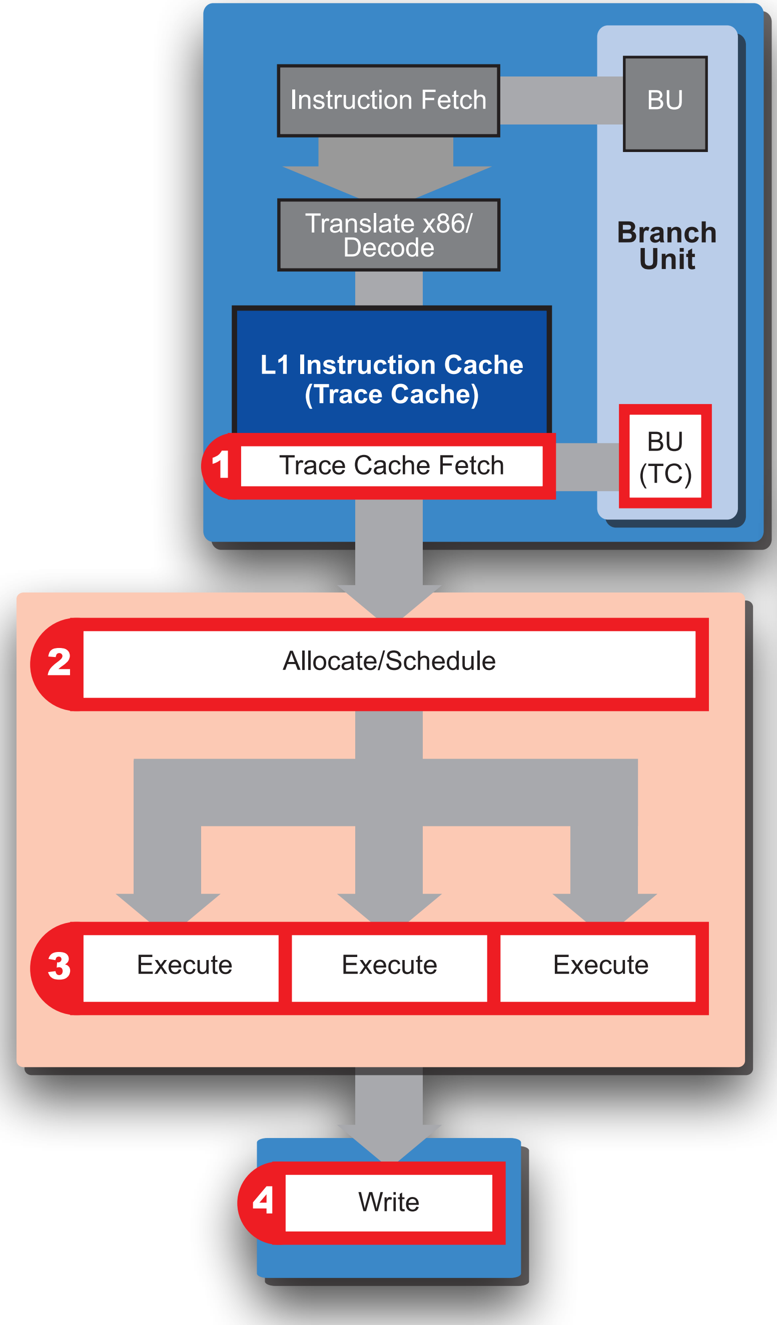


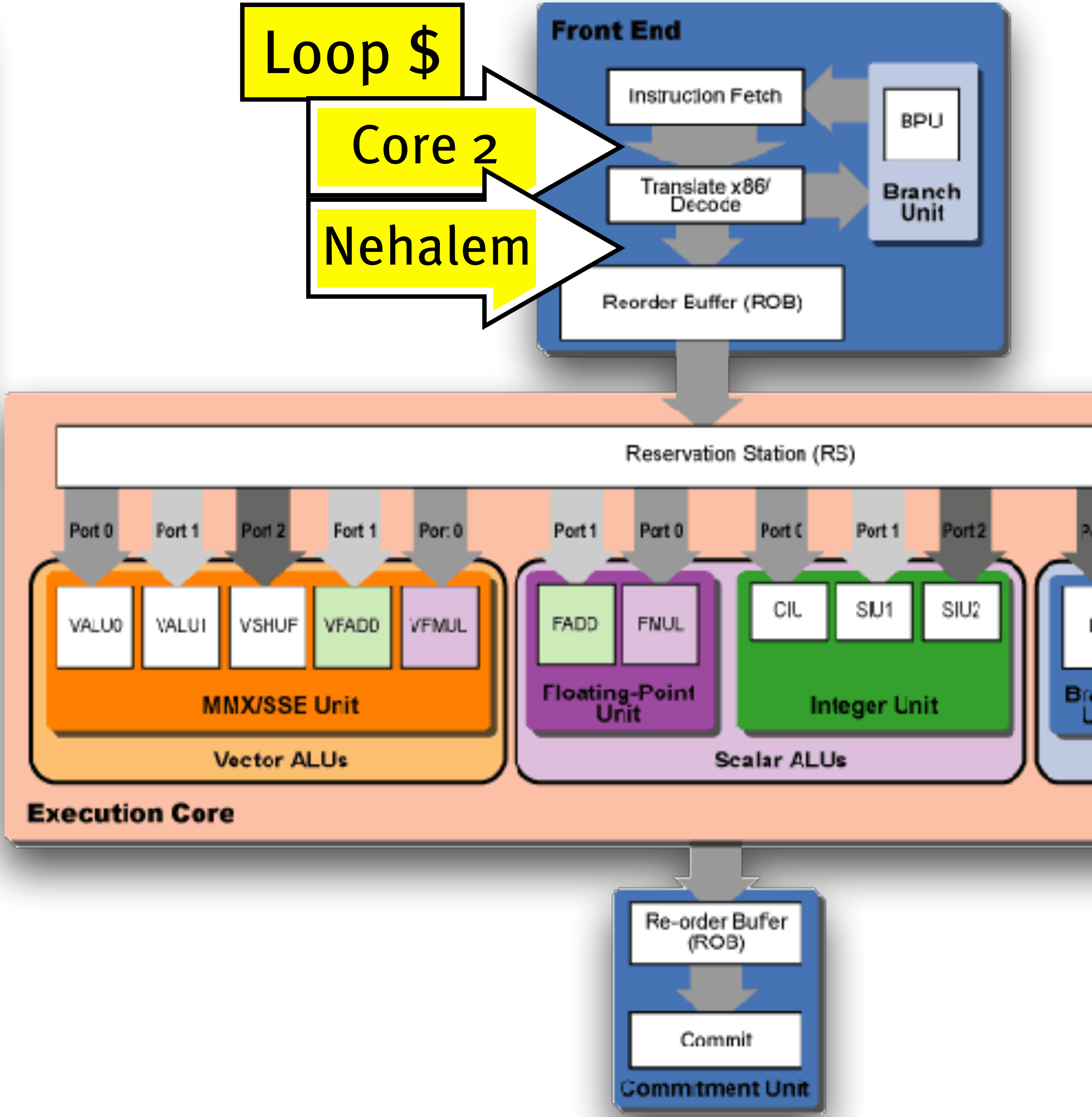
Evaluation of Trace Cache

- “The goal of the trace cache was to store decoded uops in dynamic program order, instead of the static compiler ordered x86 instructions stored in the instruction cache, thereby removing the decoder and branch predictor from the critical path and enabling multiple basic blocks to be fetched at once. The problem with the trace cache in the P4 was that it was extremely fragile; when the trace cache missed, it would decode instructions one by one. The hit rate for a normal instruction cache is well above 90%. The trace cache hit rate was extraordinarily low by those standards, rarely exceeding 80% and easily getting as low as 50–60%. In other words, 40–50% of the time, the P4 was behaving exactly like a single issue microprocessor, rather than taking full advantage of its execution resources.”—David Kanter

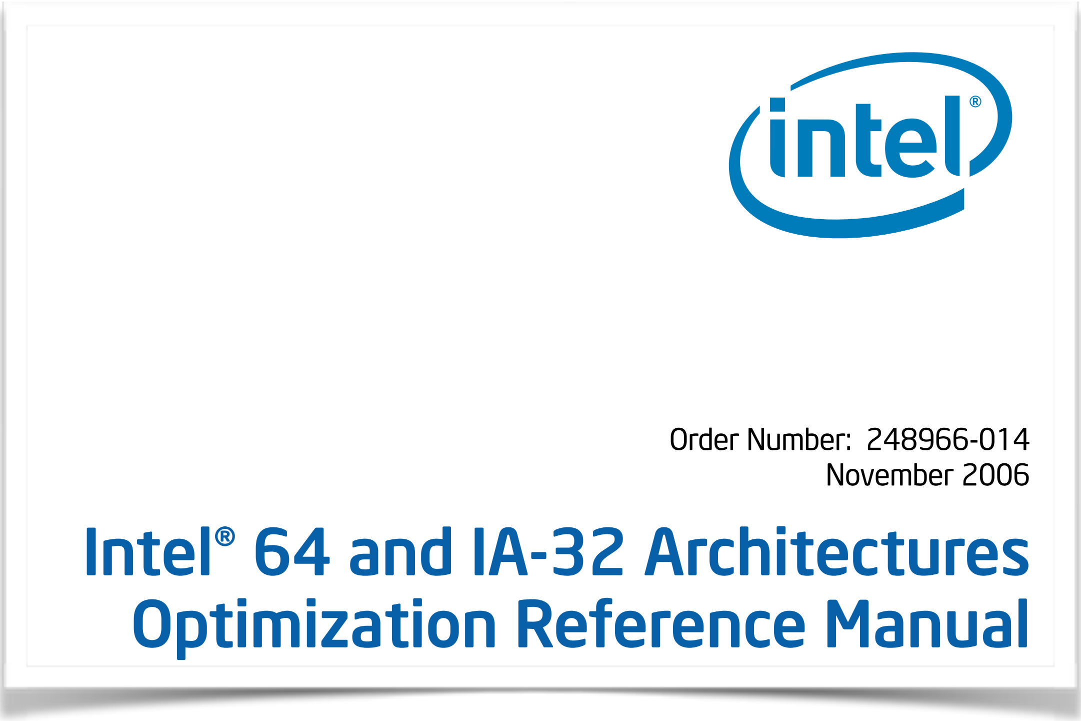
http://www.realworldtech.com/nehalem/

P4 vs. Core 2 Duo (Merom)





What does Intel say?



Order Number: 248966-014
November 2006

Intel® 64 and IA-32 Architectures
Optimization Reference Manual

All text in this section of lecture is copied straight from this manual.

Front End

- Optimizing the front end covers two aspects:
  - Maintaining steady supply of μops to the execution engine — Mispredicted branches can disrupt streams of μops, or cause the execution engine to waste execution resources on executing streams of μops in the non-architected code path. Much of the tuning in this respect focuses on working with the Branch Prediction Unit. Common techniques are covered in Section 3.4.1, "Branch Prediction Optimization."
  - Supplying streams of μops to utilize the execution bandwidth and retirement bandwidth as much as possible — For Intel Core microarchitecture and Intel Core Duo processor family, this aspect focuses maintaining high decode throughput. In Intel NetBurst microarchitecture, this aspect focuses on keeping the Trace Cache operating in stream mode. Techniques to maximize decode throughput for Intel Core microarchitecture are covered in Section 3.4.2, "Fetch and Decode Optimization."

Branch prediction

- Keep code and data on separate pages. “This is very important”. Why?
- Eliminate branches whenever possible. (next slides)
- Arrange code to be consistent with the static branch prediction algorithm. (next slides)
- Use the PAUSE instruction in spin-wait loops.
- Inline functions and pair up calls and returns.
- Unroll as necessary so that repeatedly-executed loops have sixteen or fewer iterations (unless this causes an excessive code size increase).
- Separate branches so that they occur no more frequently than every three μops where possible.

Eliminating branches

- Eliminating branches improves performance because:
  - It reduces the possibility of mispredictions.
  - It reduces the number of required branch target buffer (BTB) entries.
  - Conditional branches that are never taken do not consume BTB resources.
- There are four principal ways of eliminating branches *(next slides)*:
  - Arrange code to make basic blocks contiguous.
  - Unroll loops, as discussed in Section 3.4.1.7, “Loop Unrolling.”
  - Use the CMOV instruction.
  - Use the SETCC instruction (explained on future slide).

Assembly Rule 1

- Arrange code to make basic blocks contiguous and eliminate unnecessary branches.
  - For the Pentium M processor, every branch counts. Even correctly predicted branches have a negative effect on the amount of useful code delivered to the processor. Also, taken branches consume space in the branch prediction structures and extra branches create pressure on the capacity of the structures.

Assembly Rule 2

- Use the SETCC and CMOV instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Do not use these instructions to eliminate all unpredictable conditional branches (because using these instructions will incur execution overhead due to the requirement for executing both paths of a conditional branch). In addition, converting a conditional branch to SETCC or CMOV trades off control flow dependence for data dependence and restricts the capability of the out-of-order engine. When tuning, note that all Intel 64 and IA-32 processors usually have very high branch prediction rates. Consistently mispredicted branches are generally rare. Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.

Static Branch Predict Rules

- P4, Pentium M, Intel Core Solo and Intel Core Duo processors have similar static prediction algorithms that:
    - predict unconditional branches to be taken
    - predict indirect branches to be NOT taken
- In addition, conditional branches in processors based on the Intel NetBurst microarchitecture are predicted using the following static prediction algorithm:
    - predict backward conditional branches to be taken; rule is suitable for loops
    - predict forward conditional branches to be NOT taken

Assembly Rule 3

- Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target.
- Is it better to do:
    - if (condition) then likely() else unlikely()
    - if (condition) then unlikely() else likely()

Return address stack

- Return Stack. Returns are always taken; but since a procedure may be invoked from several call sites, a single predicted target does not suffice. The Pentium 4 processor has a Return Stack that can predict return addresses for a series of procedure calls. This increases the benefit of unrolling loops containing function calls. It also mitigates the need to put certain procedures inline since the return penalty portion of the procedure call overhead is reduced.
- Has 16 entries (P4). "If there is a chain of more than 16 nested calls and more than 16 returns in rapid succession, performance may degrade."

Assembly Rule 14

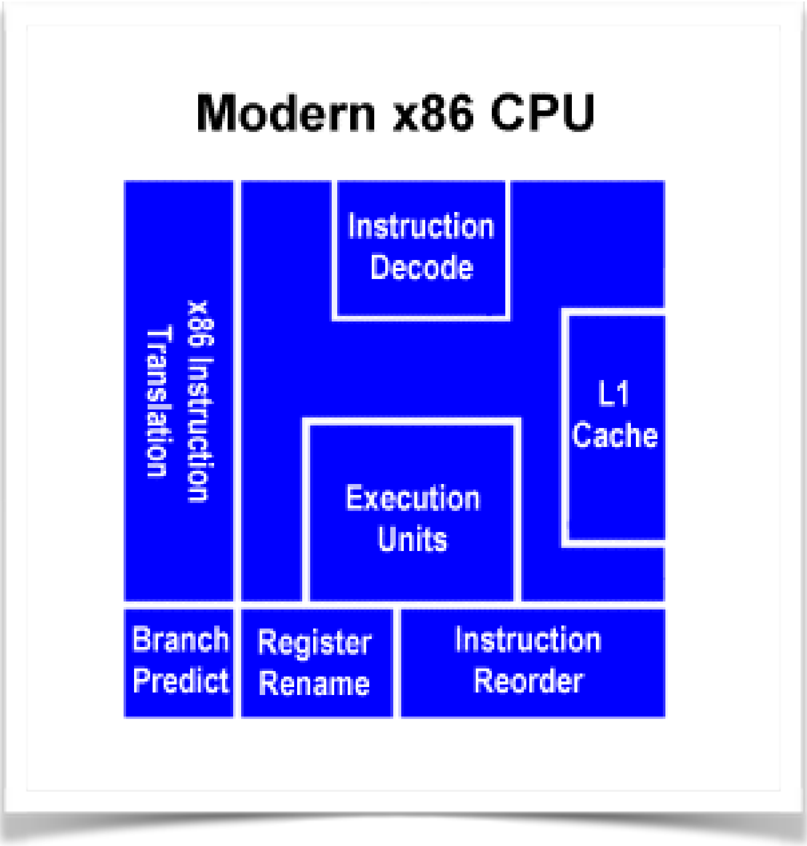
- Assembly/Compiler Coding Rule 14. (M impact, L generality) When indirect branches are present, try to put the most likely target of an indirect branch immediately following the indirect branch. Alternatively, if indirect branches are common but they cannot be predicted by branch prediction hardware, then follow the indirect branch with a UD2 instruction, which will stop the processor from decoding down the fall-through path.

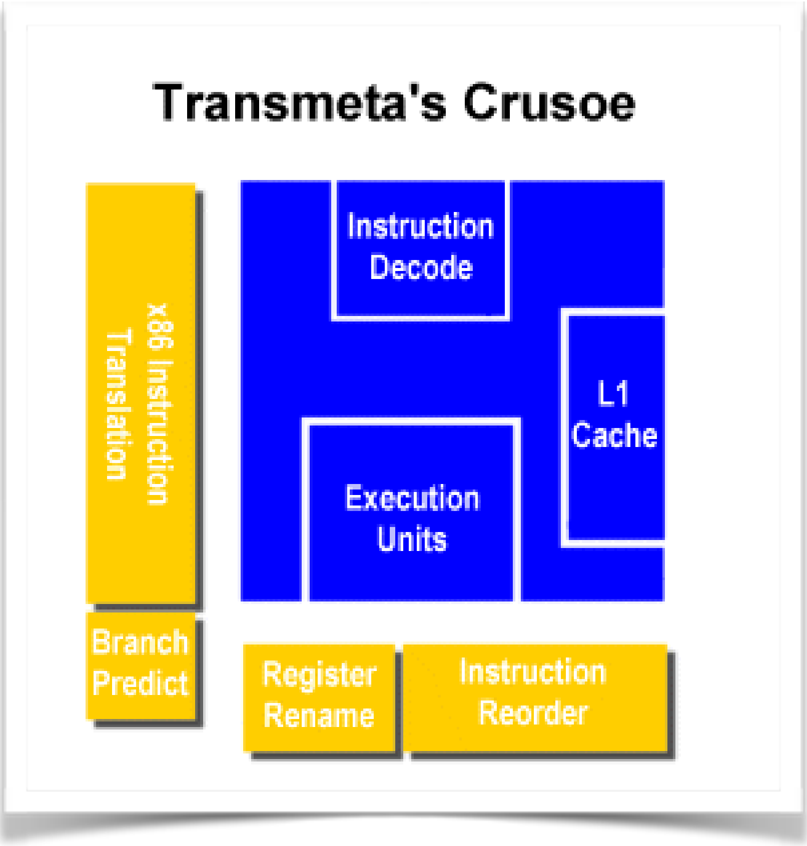
Transmeta motivation

- Intel/AMD goals:
  - x86 compatibility
  - Fastest performance
- Transmeta goals:
  - x86 compatibility
  - lowest possible power consumption
  - reasonable performance

http://arstechnica.com/articles/paedia/cpu/crusoe.ars

HW vs. SW approaches





Crusoe is VLIW

- Functional units
  - 1 FPU
  - 2 ALUs
  - 1 load-store unit
  - 1 branch unit
- 64 registers

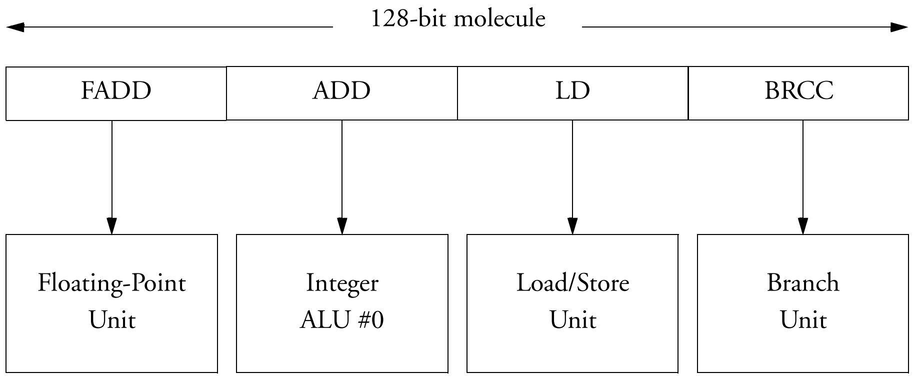
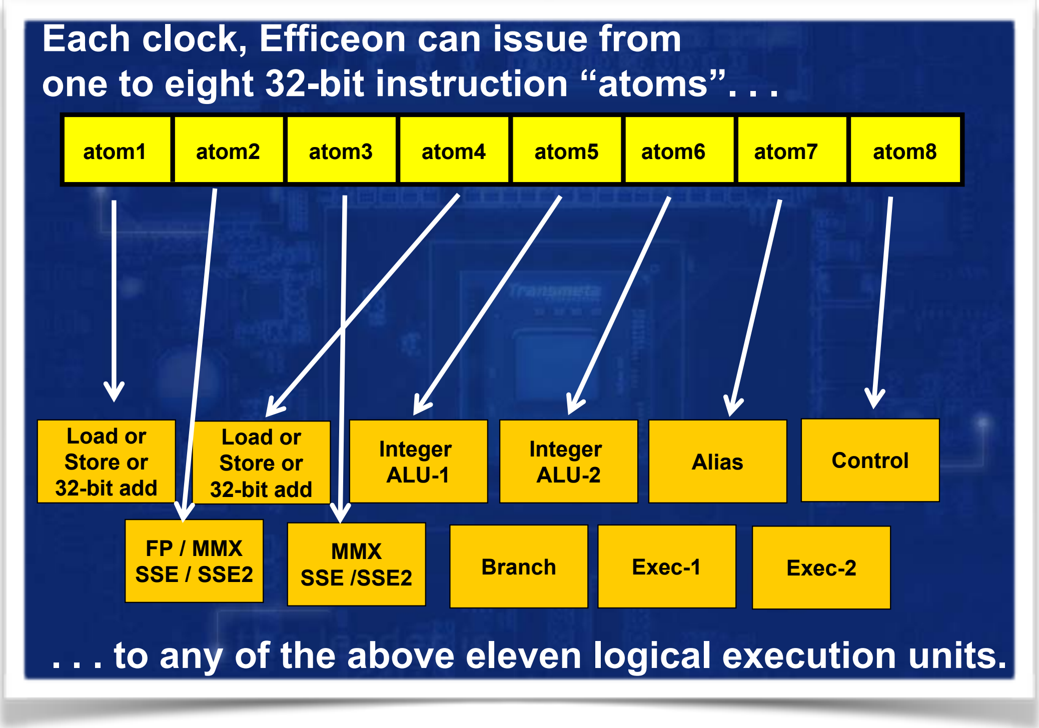


Figure 1. A molecule can contain up to four atoms, which are executed in parallel.

Efficion is VLIW

- Functional units
  - 2 FPUs
  - 2 ALUs
  - 2 load-store units
  - 2 “execute” units
  - 1 branch unit
  - 1 control unit
  - 1 alias unit
- 256b wide

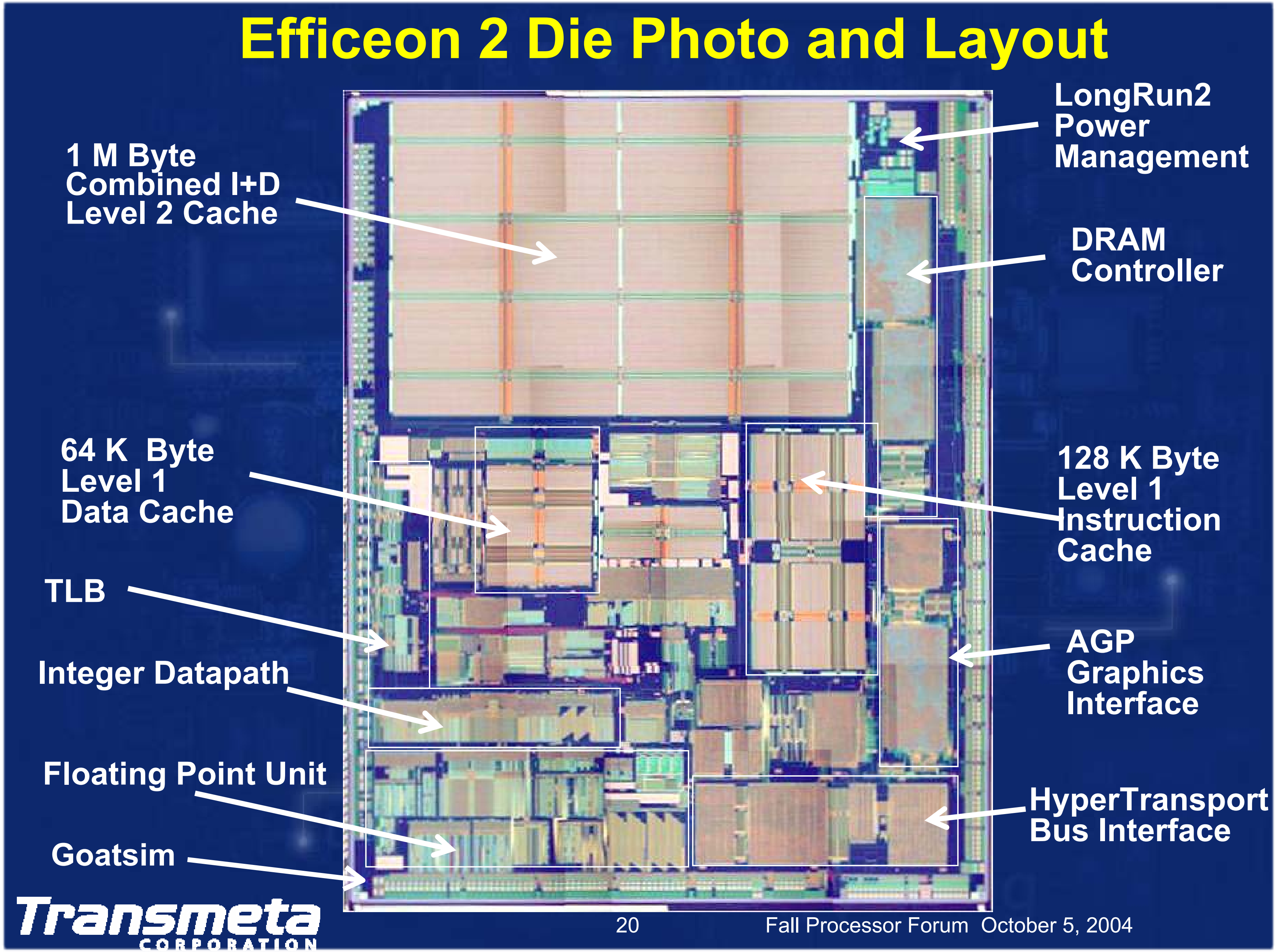


Each clock, Efficeon can issue from one to eight 32-bit instruction “atoms”. . .

. . . to any of the above eleven logical execution units.

David Ditzel, 2004 Fall Processor Forum

Efficeon 2 Die Photo



Efficeon 2 Die Photo and Layout

1 M Byte
Combined I+D
Level 2 Cache

64 K Byte
Level 1
Data Cache

TLB

Integer Datapath

Floating Point Unit

Goatsim

LongRun2
Power
Management

DRAM
Controller

128 K Byte
Level 1
Instruction
Cache

AGP
Graphics
Interface

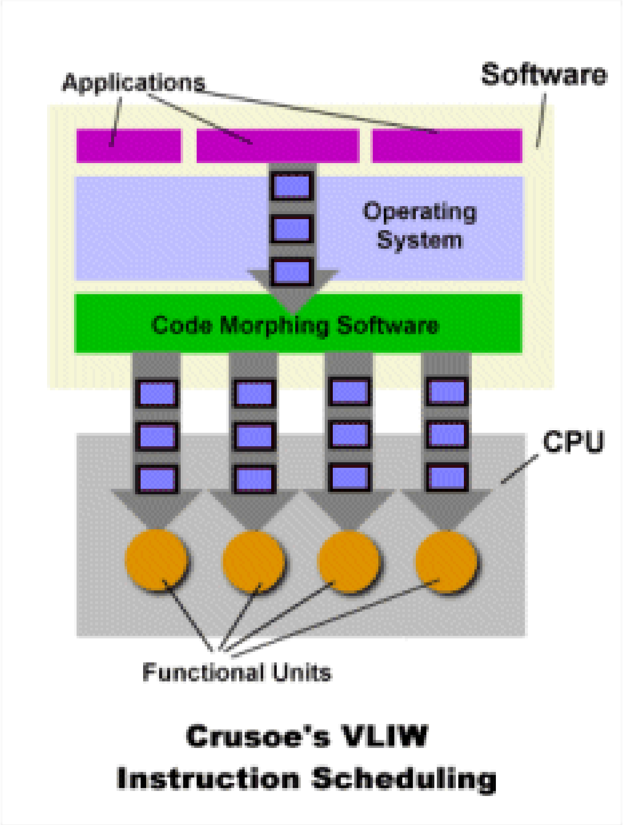
HyperTransport
Bus Interface

Transmeta
CORPORATION

20 Fall Processor Forum October 5, 2004

Code Morphing

- x86 fed to Code Morphing layer
- CM translates chunk of x86 to VLIW
    - Output stored in translation cache
- CM watches execution:
    - Frequently used chunks are more heavily optimized
    - Watches branches, can tailor speculation to history
- Some CM support in hardware



Crusoe's VLIW Instruction Scheduling

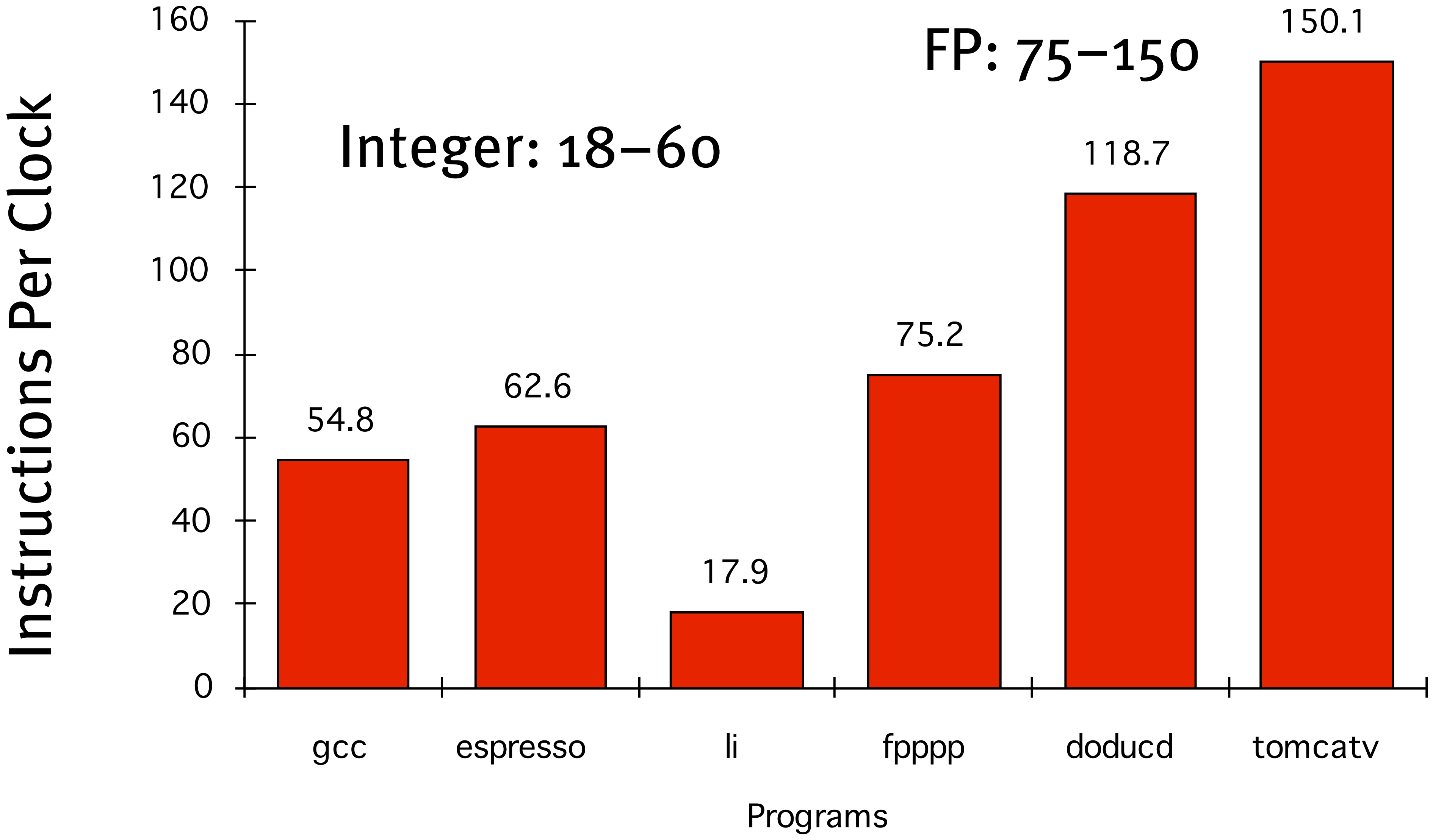
Limits to ILP

- Conflicting studies of amount
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication
- How much ILP is available using existing mechanisms with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola AltiVec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.

Overcoming Limits

- Advances in compiler technology + significantly new and different hardware techniques may be able to overcome limitations assumed in studies
- However, unlikely such advances when coupled with realistic hardware will overcome these limits in near future

Upper Limit to ILP: Ideal Machine



Limits to ILP

- Most techniques for increasing performance increase power consumption
- The key question is whether a technique is energy efficient: does it increase power consumption faster than it increases performance?
- Multiple issue processor techniques all are energy inefficient:
  - Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
  - Growing gap between peak issue rates and sustained performance
- Number of transistors switching = *f*(peak issue rate), and performance = *f*(sustained rate):
  Growing gap between peak and sustained performance
  ⇒ increasing energy per unit of performance

Limits to ILP

- Doubling issue rates above today's 3–6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to
  - Issue 3 or 4 data memory accesses per cycle,
  - Resolve 2 or 3 branches per cycle,
  - Rename and access more than 20 registers per cycle, and
  - Fetch 12 to 24 instructions per cycle.
- Complexities of implementing these capabilities likely means sacrifices in maximum clock rate
  - E.g, widest issue processor is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power!

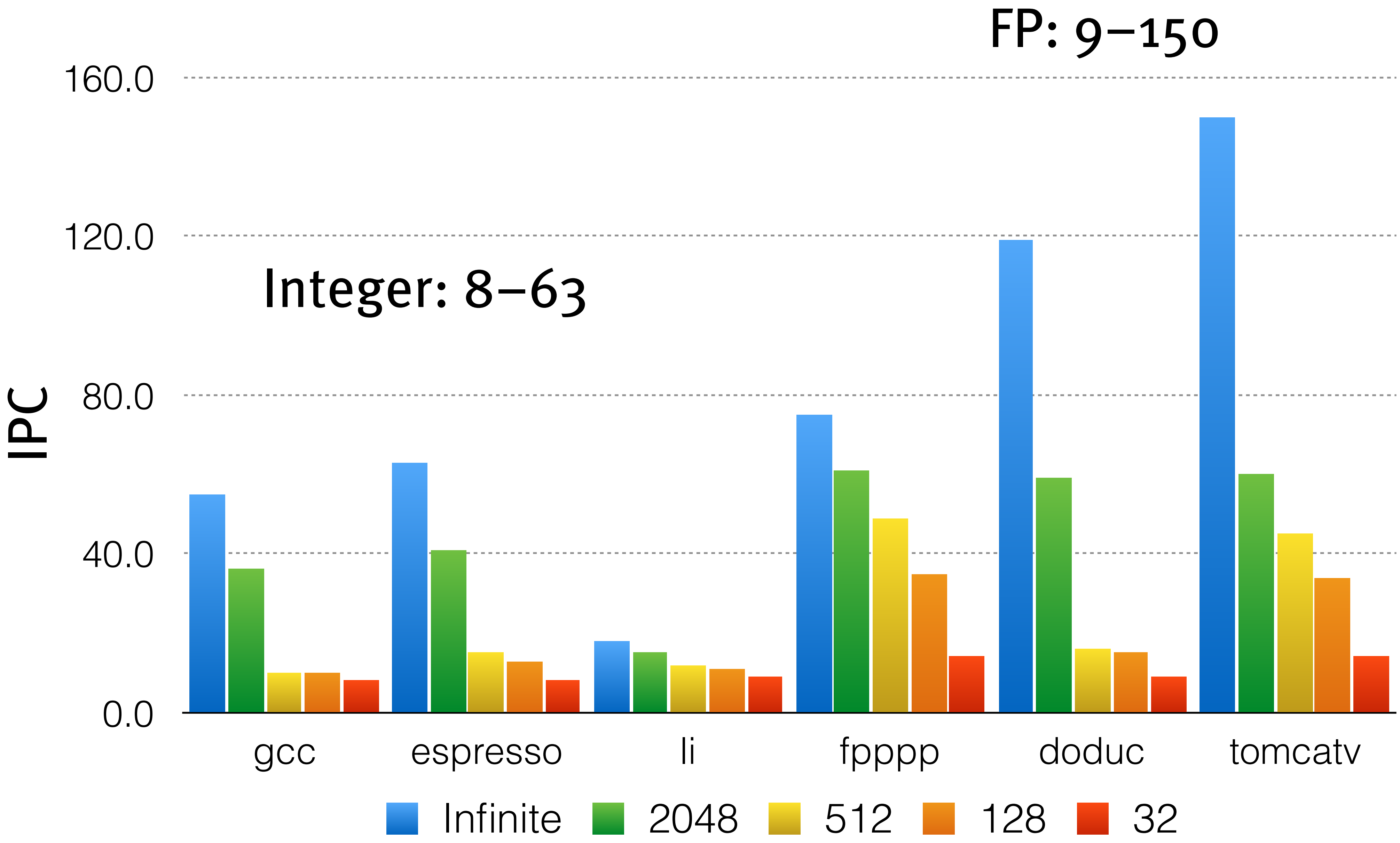
Limits to ILP

- Initial HW Model here; MIPS compilers.
- Assumptions for ideal/perfect machine to start: [this comes up on exams]
  - 1. Register renaming – infinite virtual registers
    => all register WAW & WAR hazards are avoided
  - 2. Branch prediction – perfect; no mispredictions
  - 3. Jump prediction – all jumps perfectly predicted (returns, case statements)
    2 & 3 ⇒ no control dependencies; perfect speculation & an unbounded buffer of instructions available
  - 4. Memory-address alias analysis – addresses known & a load can be moved before a store provided addresses not equal; 1&4 eliminates all but RAW
- Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;

Limits to ILP HW Model comparison

	New Model	Ideal	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	256 Int + 256 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect v. Stack v. Inspect v. none	Perfect	Perfect

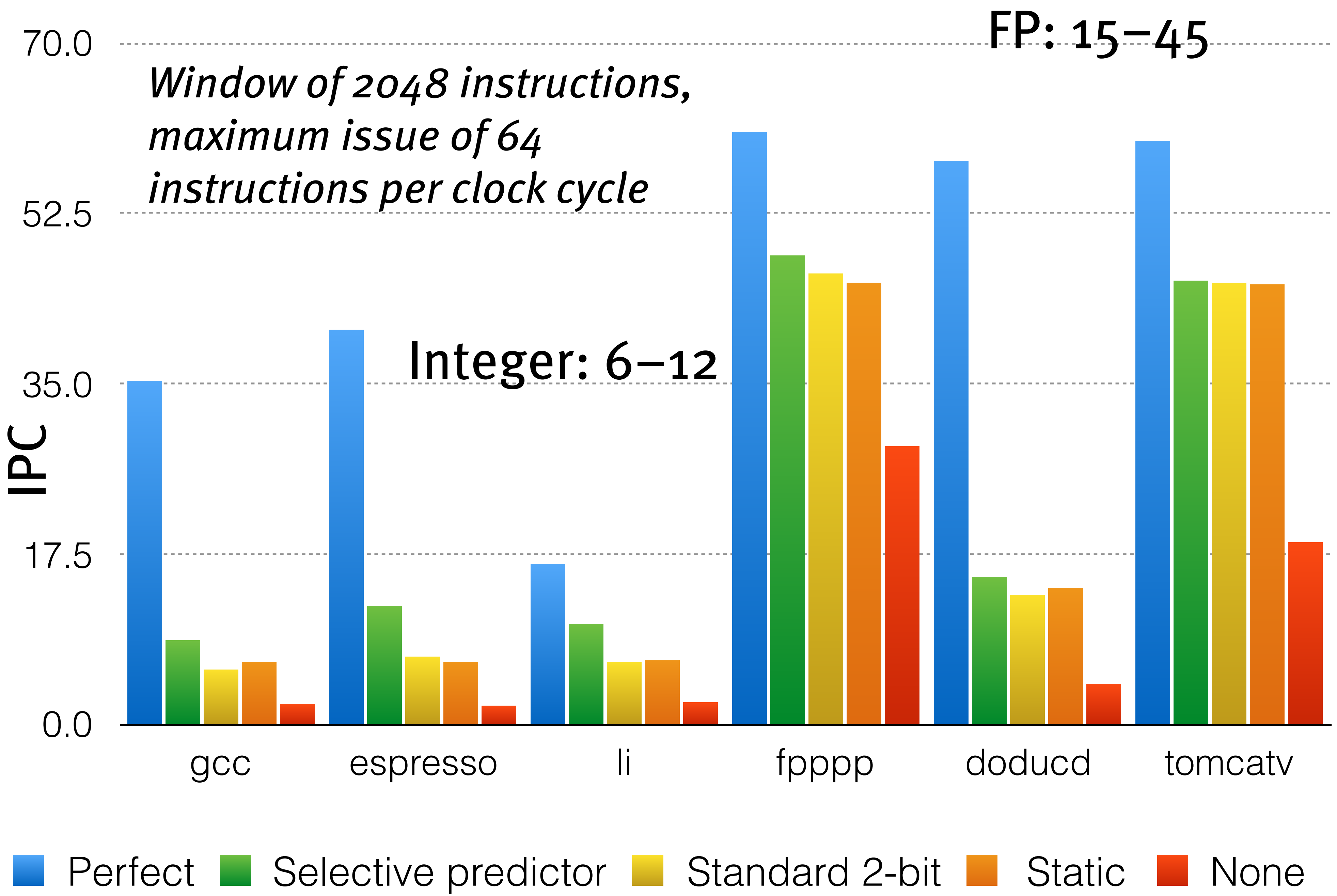
More Realistic HW: Window Impact



FP: 9–150

Integer: 8–63

More Realistic HW: Branch Impact

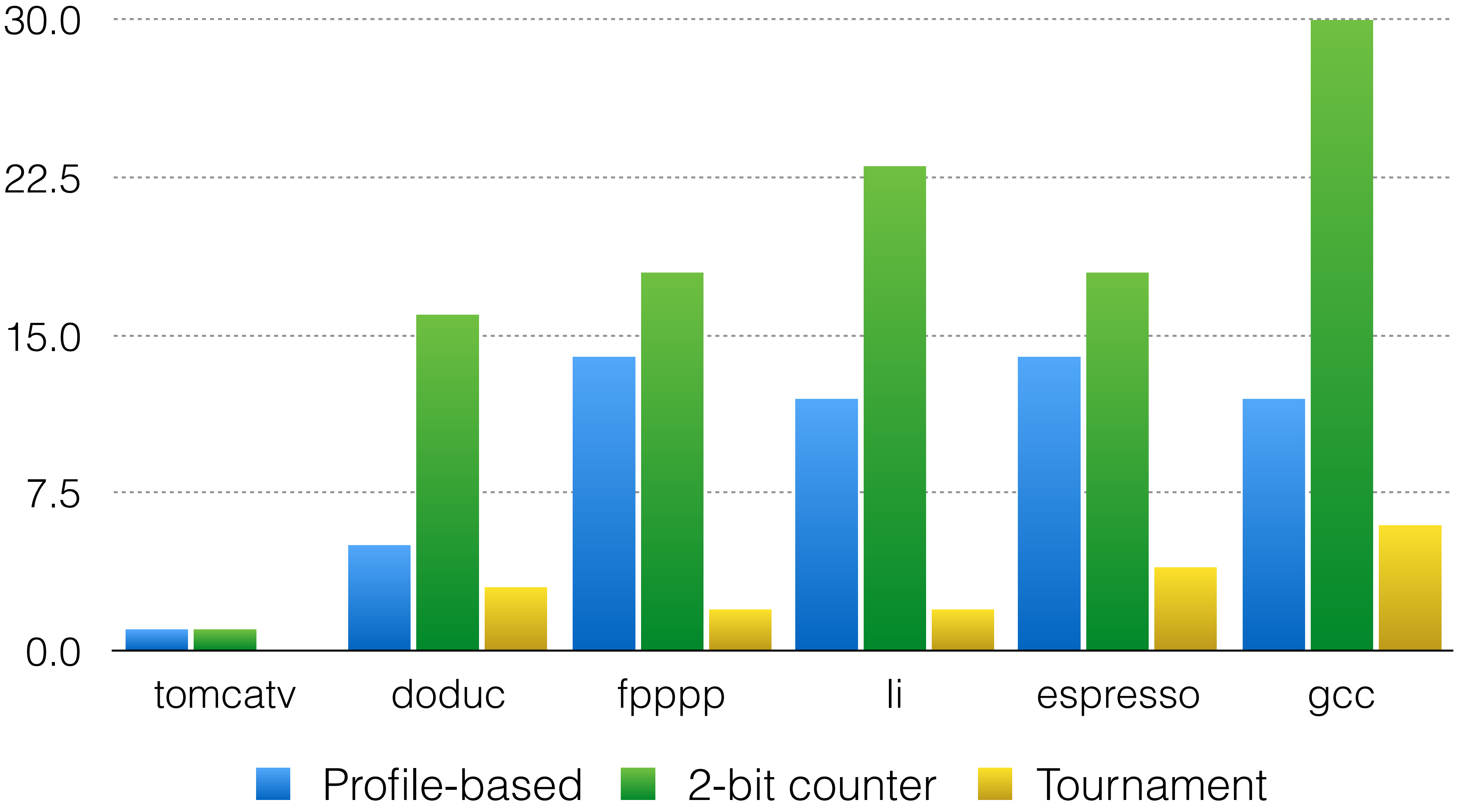


Window of 2048 instructions, maximum issue of 64 instructions per clock cycle

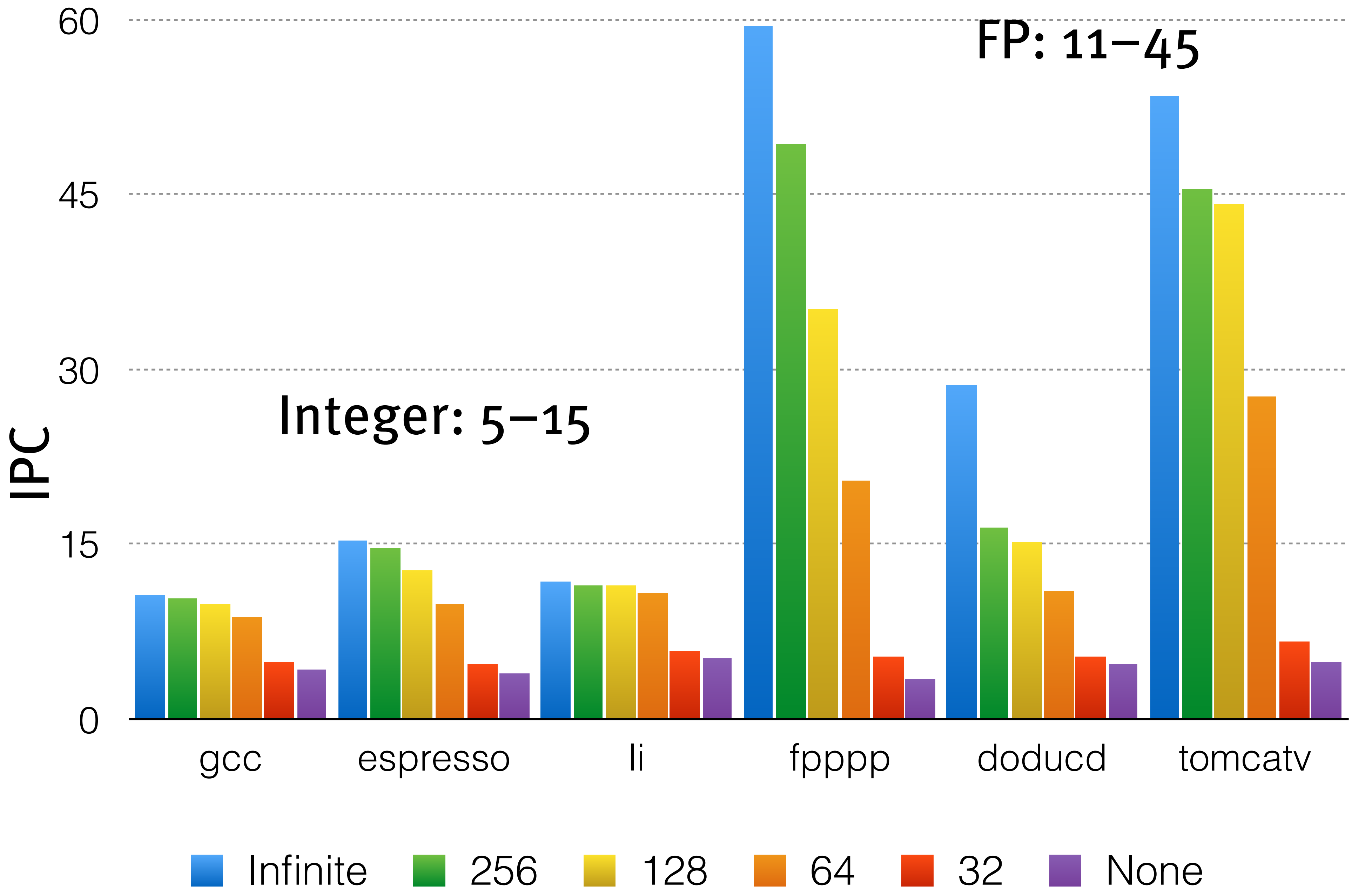
Integer: 6–12

FP: 15–45

Misprediction Rates (%)



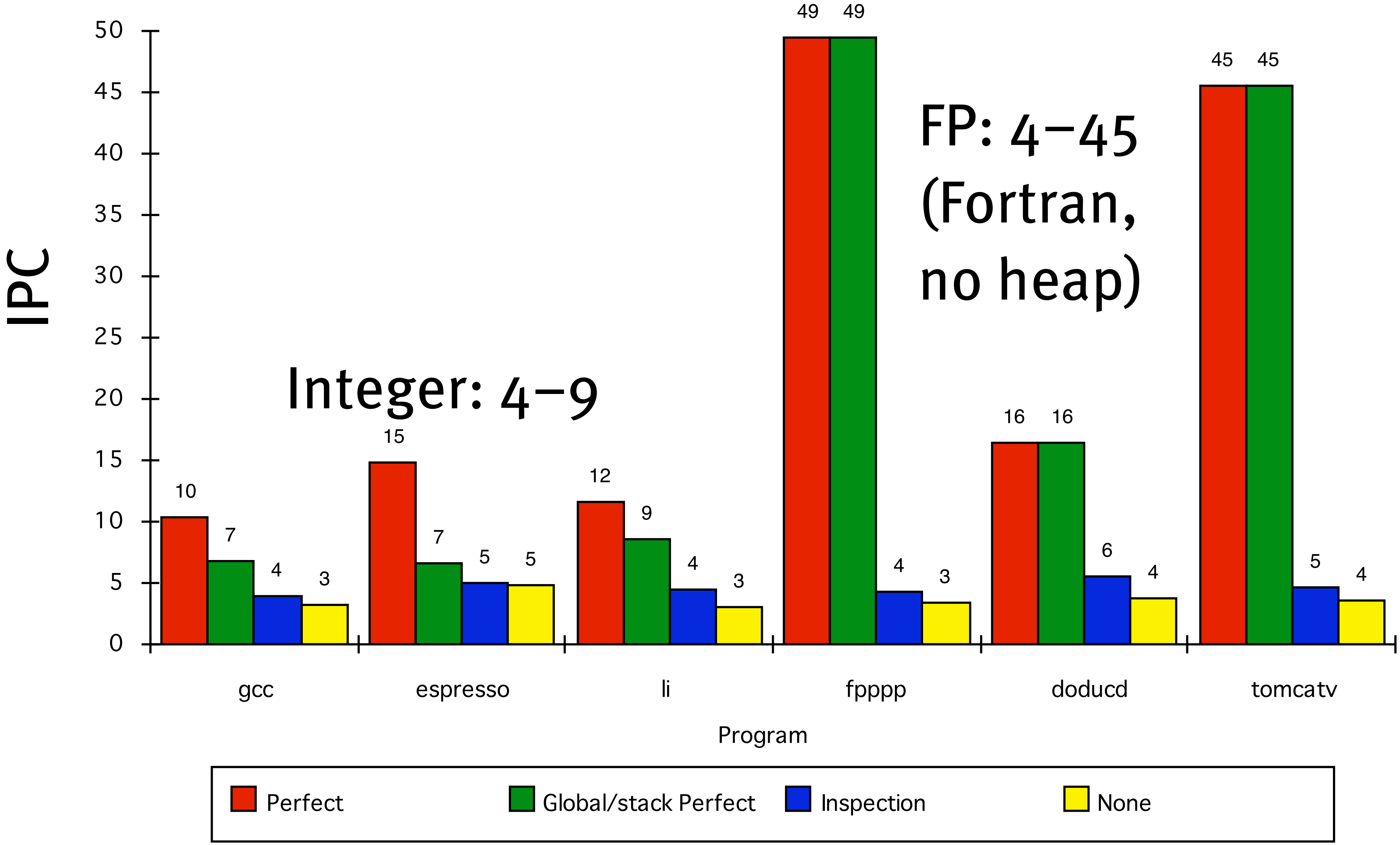
Renaming Register Impact



Integer: 5–15

FP: 11–45

Memory Address Alias Impact



Integer: 4–9

FP: 4–45
(Fortran,
no heap)

HW vs. SW to increase ILP

- Memory disambiguation: HW best
- Speculation:
  - HW best when dynamic branch prediction better than compile time prediction
  - Exceptions easier for HW
  - HW doesn’t need bookkeeping code or compensation code
  - Very complicated to get right
- Scheduling: SW can look ahead to schedule better
- Compiler independence: does not require new compiler, recompilation to run well

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

ILP Summary

- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
  - Chief advantage: Accelerates unmodified serial instruction stream
- Loop unrolling by compiler to increase ILP
- Branch prediction (speculation) + predication to increase ILP
- Dynamic HW exploiting ILP
  - Works when can’t know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another