Beyond Superscalar

Virendra Singh

Associate Professor

Computer Architecture and Dependable Systems Lab

Department of Electrical Engineering

Indian Institute of Technology Bombay

http://www.ee.iitb.ac.in/~viren/

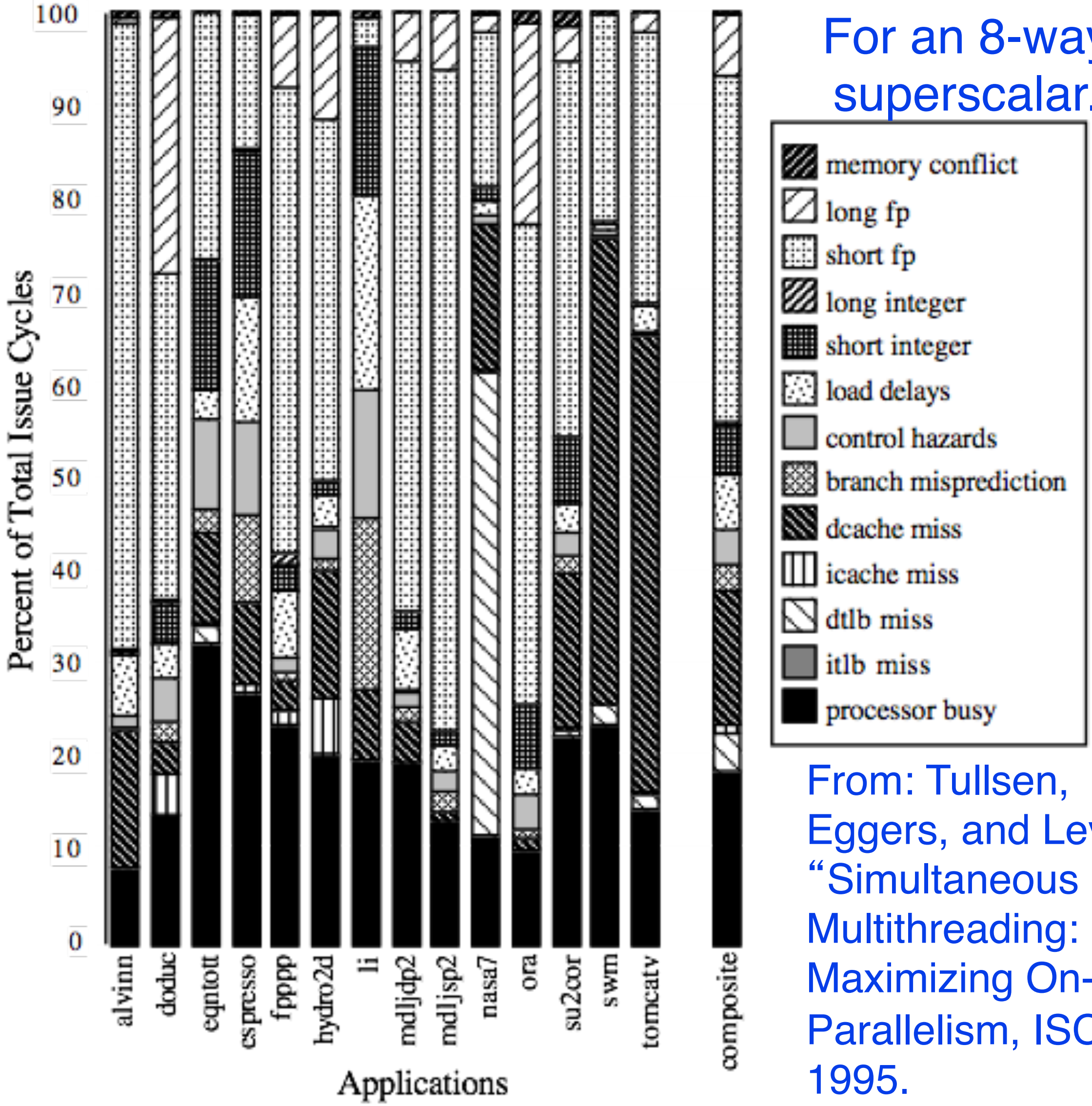
E-mail: viren@ee.iitb.ac.in

EE-739: Processor Design

Lecture 10 (12 Feb 2015)

CADSL

For most apps, most execution units lie idle



For an 8-way superscalar.

From: Tullsen, Eggers, and Levy, “Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

Superscalar Scenario

- Interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Conservative in ideas, just faster clock and bigger
- Processors of last 15 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units → performance 8 to 16X
- Peak v. delivered performance gap increasing

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 AltaVec: 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

Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

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

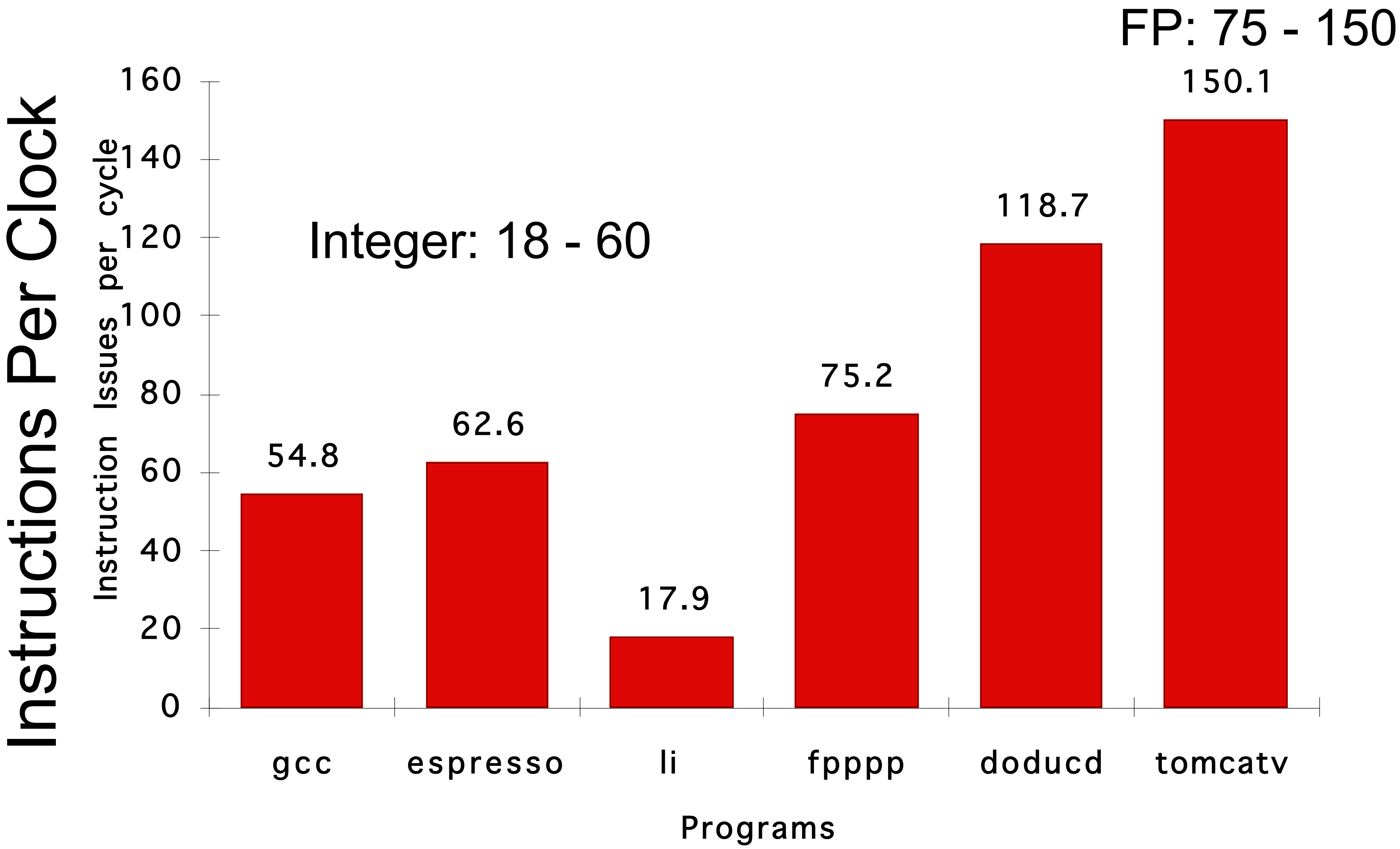
4. *Memory-address alias analysis* – addresses known & a load can be moved before a store provided addresses not equal;
1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;

Limits to ILP HW Model comparison

	Model	Power 5
Instructions Issued per clock	Infinite	4
Instruction Window Size	Infinite	200
Renaming Registers	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias Analysis	Perfect	??

Upper Limit to ILP: Ideal Machine



Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	Infinite	Infinite	4
Instruction Window Size	Infinite, 2K, 512, 128, 32	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??

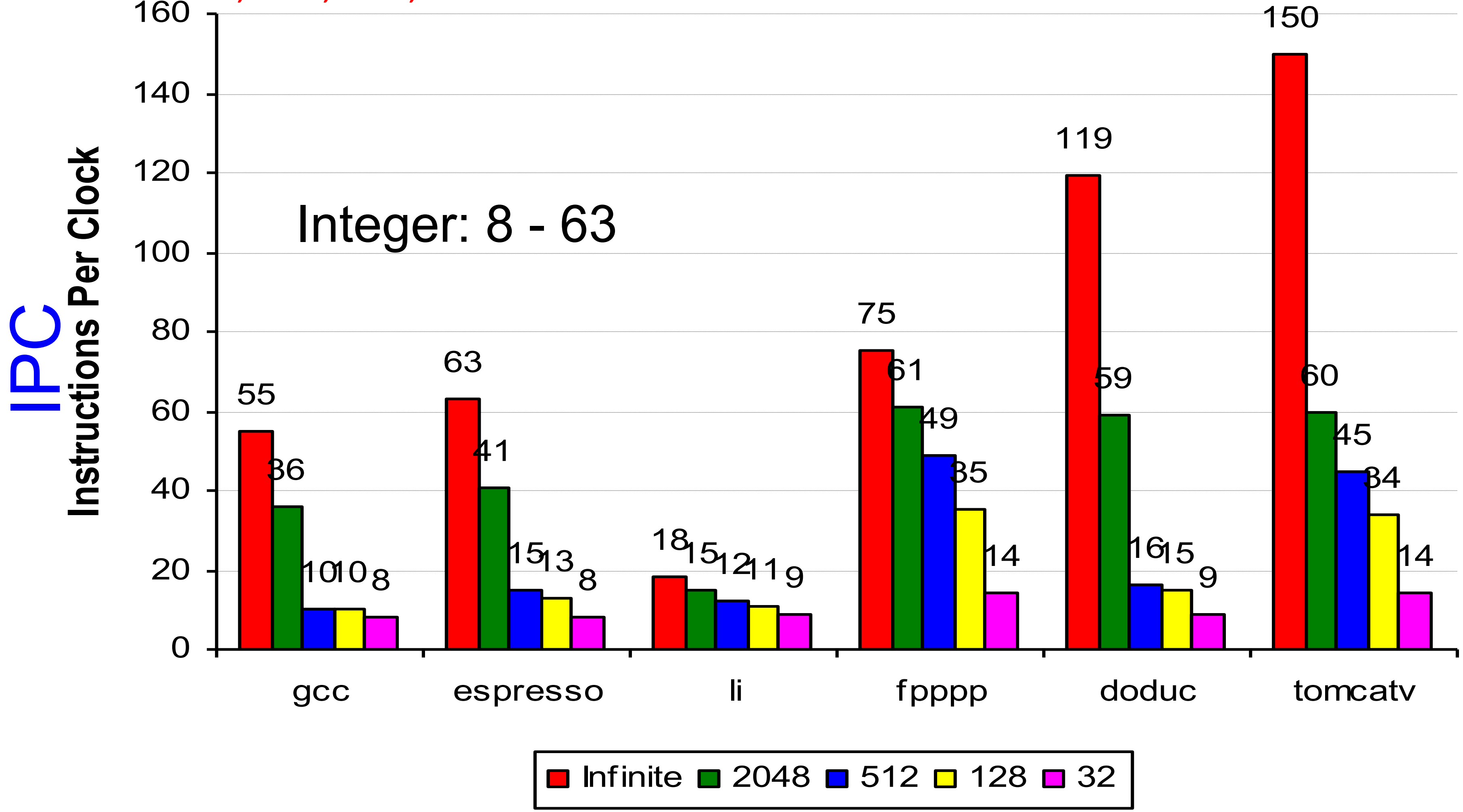
More Realistic HW: Window Impact

Change from Infinite window

2048, 512, 128, 32

FP: 9 - 150

Integer: 8 - 63



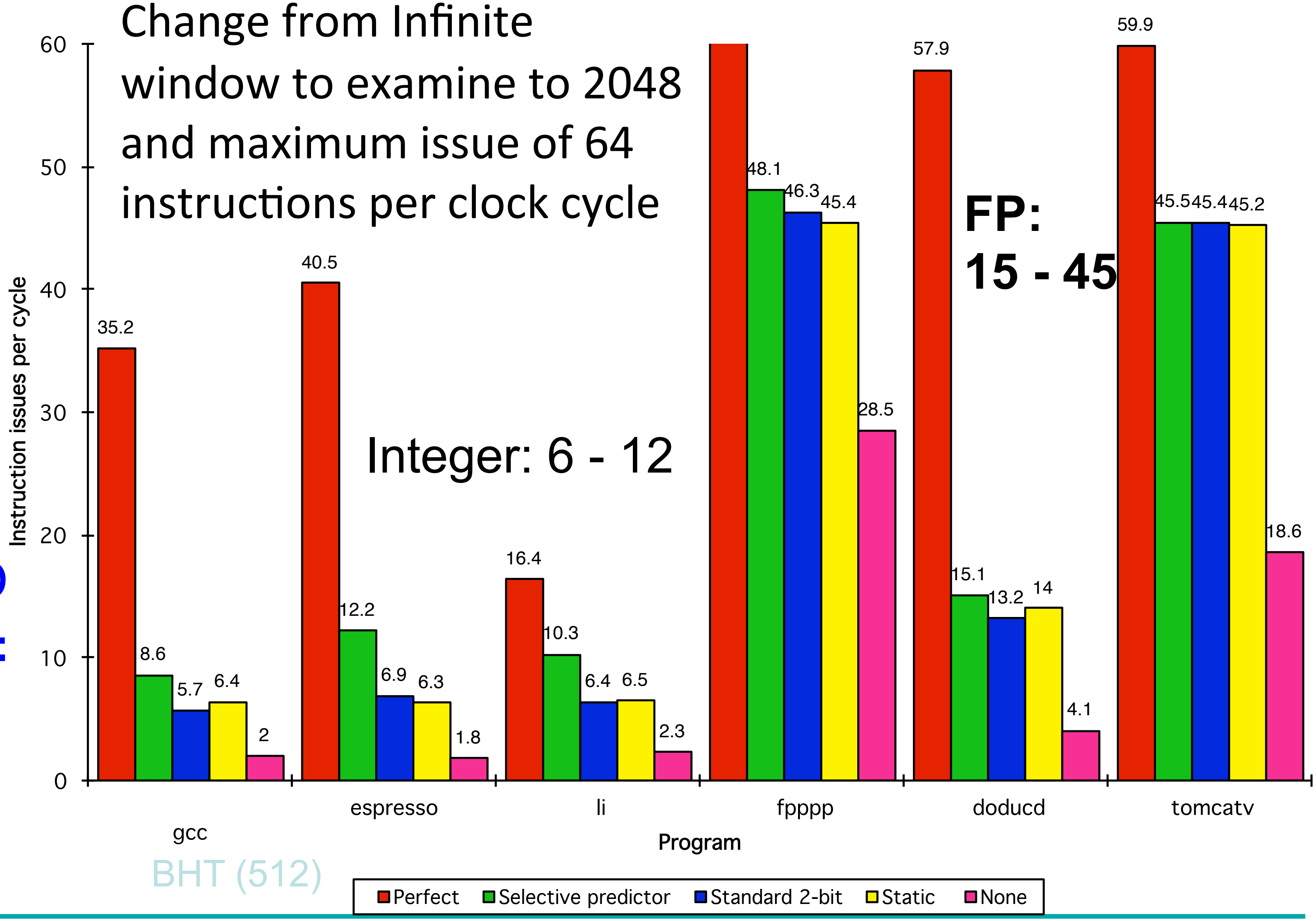
CADSL

Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??

More Realistic HW: Branch Impact

Change from Infinite window to examine to 2048 and maximum issue of 64 instructions per clock cycle



IPC

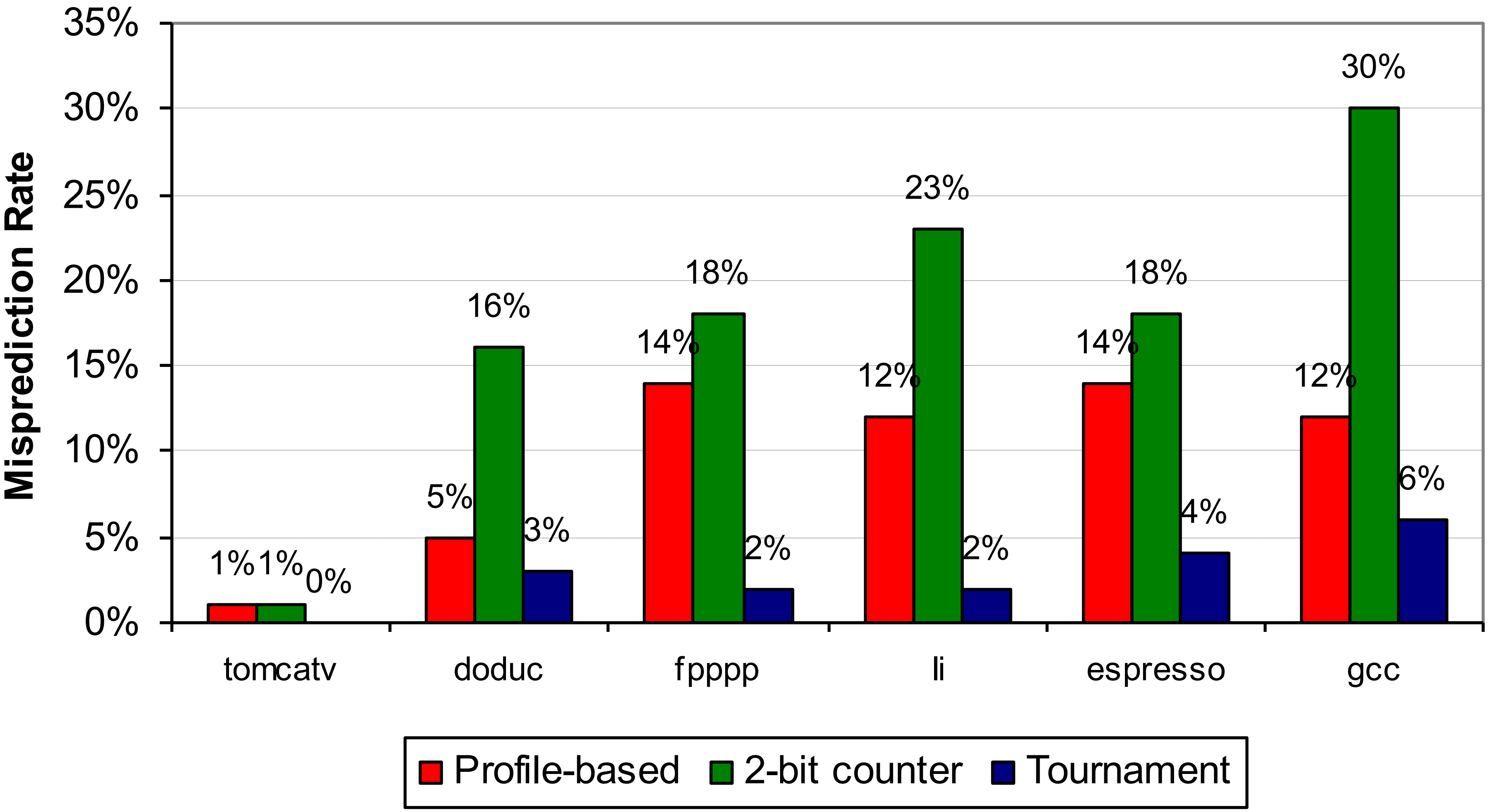
Integer: 6 - 12

FP: 15 - 45

BHT (512)

Profile

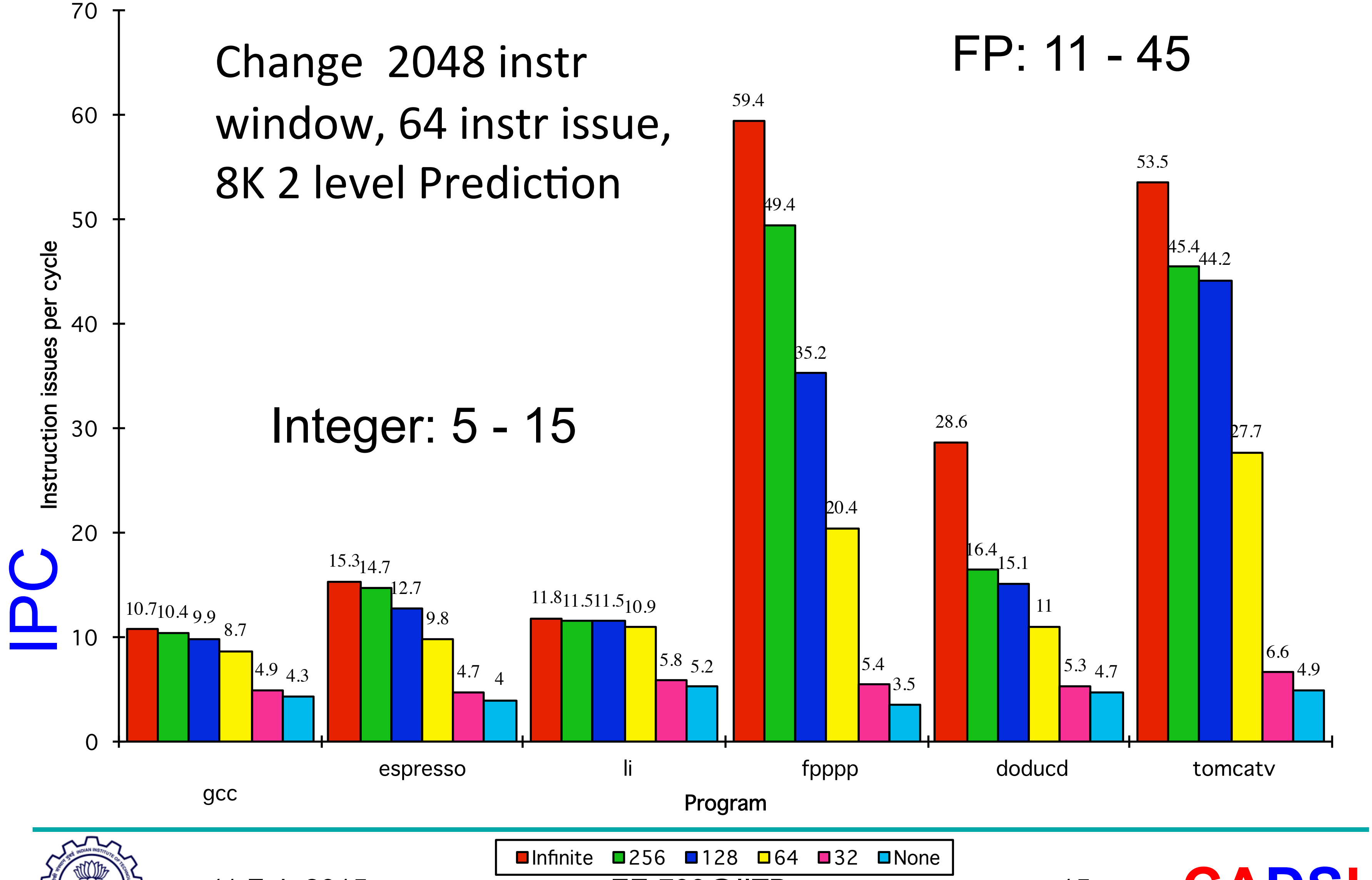
Misprediction Rates



Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite v. 256, 128, 64, 32, none	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament Branch Predictor
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	Perfect

More Realistic HW: Renaming Register Impact (N int + N fp)



Change 2048 instr window, 64 instr issue, 8K 2 level Prediction

FP: 11 - 45

Integer: 5 - 15

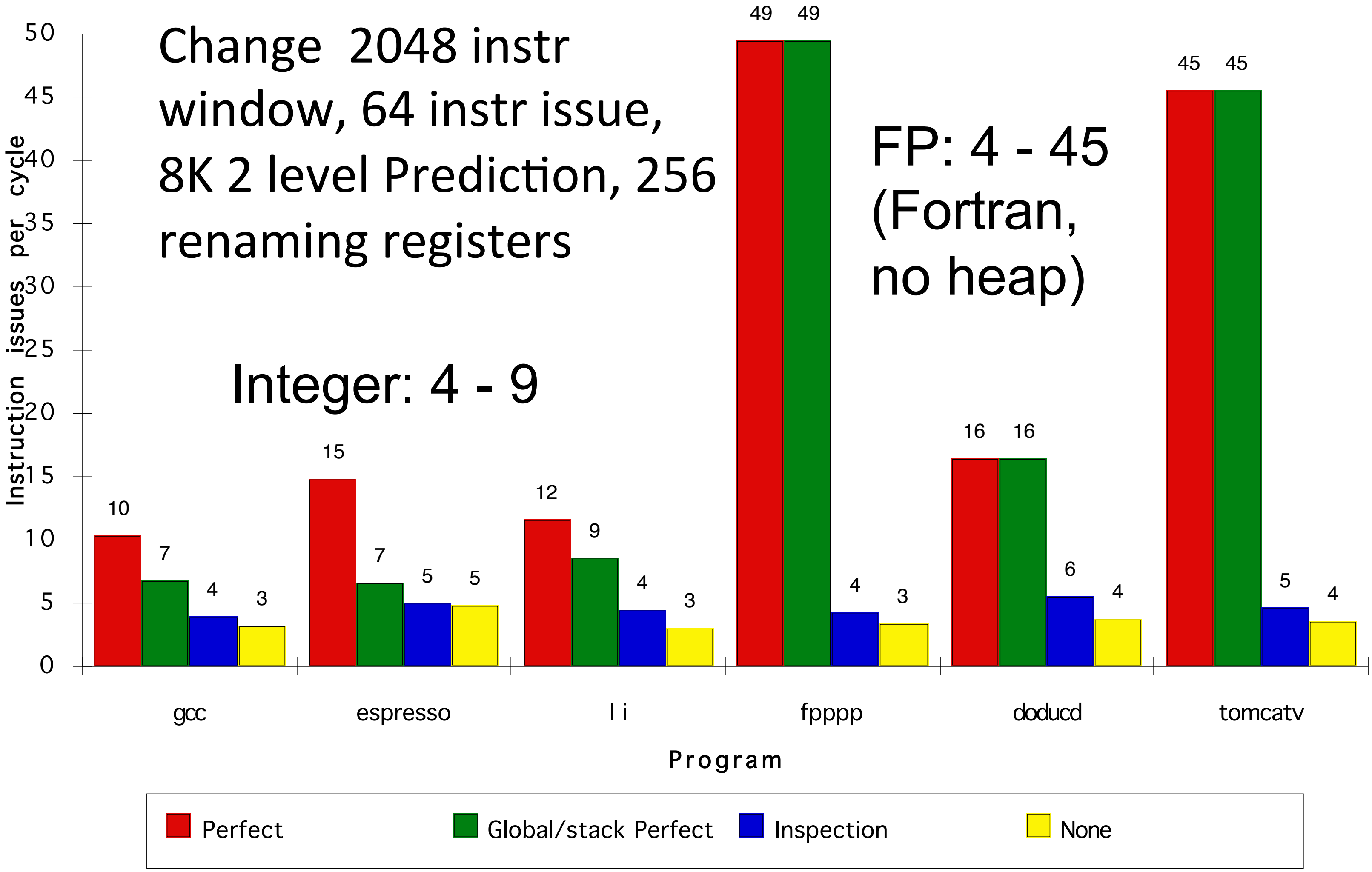
IPC

Limits to ILP HW Model comparison

	New Model	Model	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: Memory Address Alias Impact

IPC



Change 2048 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

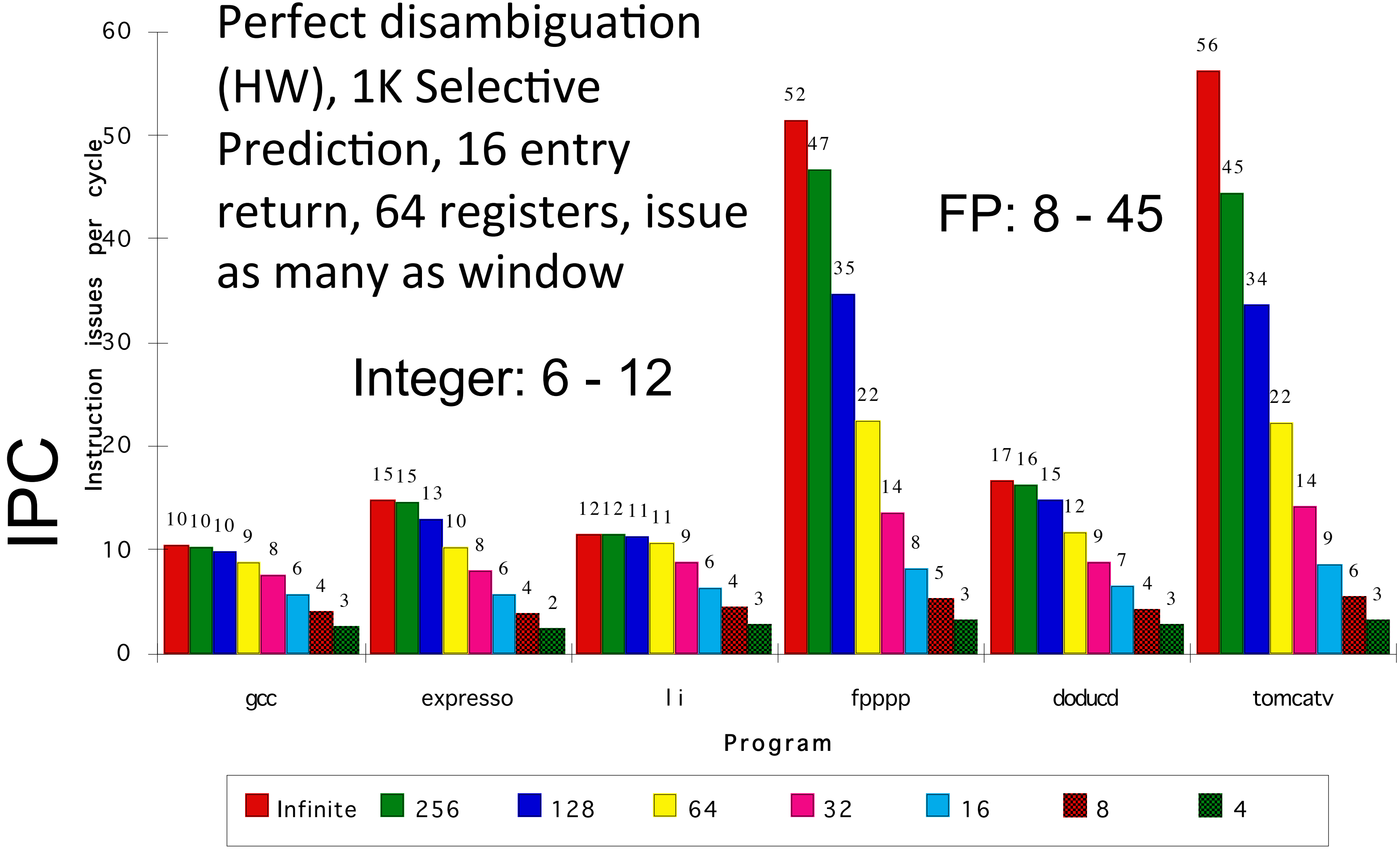
Integer: 4 - 9

FP: 4 - 45 (Fortran, no heap)

Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64 (no restrictions)	Infinite	4
Instruction Window Size	Infinite vs. 256, 128, 64, 32	Infinite	200
Renaming Registers	64 Int + 64 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	1K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	HW disambiguation	Perfect	Perfect

Realistic HW: Window Impact



Perfect disambiguation (HW), 1K Selective Prediction, 16 entry return, 64 registers, issue as many as window

Integer: 6 - 12

FP: 8 - 45

How to Exceed ILP Limits of this study?

- These are not laws of physics; just practical limits for today, and perhaps overcome via research
- Compiler and ISA advances could change results
- WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage

HW v. 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

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
- Goal: Use multiple instruction streams to improve
  1. Throughput of computers that run many programs
  2. Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

Beyond ILP: Multithreading

- Basic idea:
  - CPU resources are expensive and should not be idle
- 1960's: Virtual memory and multiprogramming
  - Virtual memory/multiprogramming invented to tolerate latency to secondary storage (disk/tape/etc.)
  - Processor-disk speed mismatch:
    - microseconds to tens of milliseconds (1:10000 or more)
  - OS context switch used to bring in other useful work while waiting for page fault or explicit read/write
  - Cost of context switch must be much less than I/O latency (easy)

Beyond ILP: Multithreading

- 1990's: Memory wall and multithreading
  - Processor-DRAM speed mismatch:
    - nanosecond to fractions of a microsecond (1:500)
  - H/W task switch used to bring in other useful work while waiting for cache miss
  - Cost of context switch must be much less than cache miss latency
- Very attractive for applications with abundant thread-level parallelism
  - Commercial multi-user workloads

Program vs Process

- Program is a passive entity which specifies the logic of data manipulation and IO action
- Process is an active entity which performs the actions specified in a program
- Multiple execution of a program process leads to concurrent processes

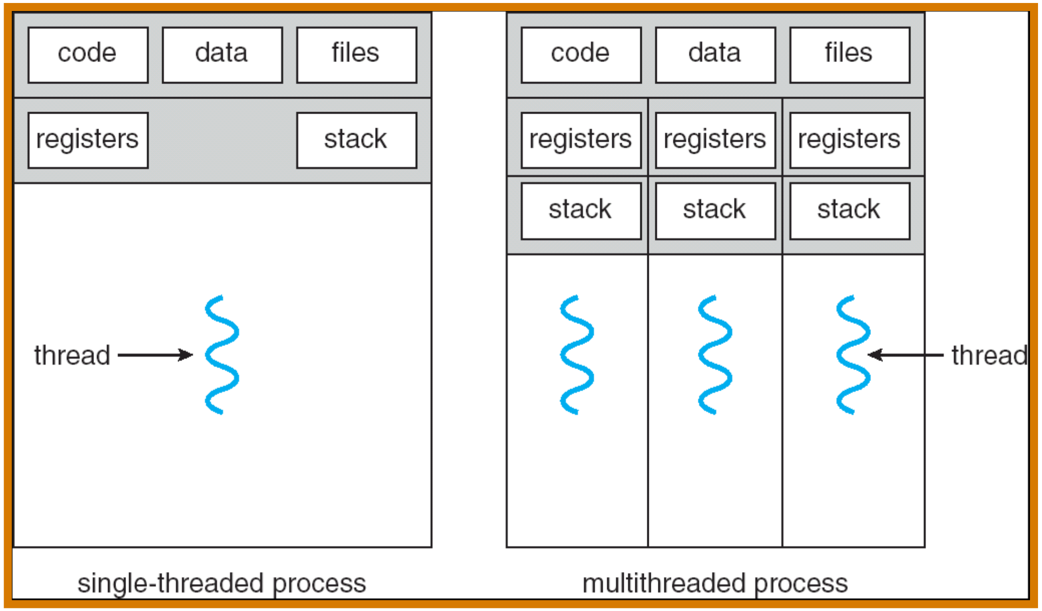
Process

- Process is a program in execution that can be in a number of states
  - New, running, waiting, ready, terminated
- Process creation
  - fork() and exec() system calls
- Inter-process communications
  - Shared memory, and message passing
- Client-server communication
  - Socket, RPC, RMI

Threads

- A thread (a lightweight process) is a basic unit of CPU utilization.
- A thread has a single sequential flow of control.
- A thread is comprised of: A thread ID, a program counter, a register set and a stack.
- A process is the execution environment in which threads run.
  - (Recall previous definition of process: program in execution).
- The process has the code section, data section, OS resources (e.g. open files and signals).
- Traditional processes have a single thread of control
- Multi-threaded processes have multiple threads of control
  - The threads share the address space and resources of the process that owns them.

Single and Multithreaded Processes



Threads encapsulate concurrency: "Active" component

Address spaces encapsulate protection: "Passive" part

Keeps buggy program from trashing the system

Processes vs. Threads

Which of the following belong to the process and which to the thread?

Program code:	Process
local or temporary data:	Thread
global data:	Process
allocated resources:	Process
execution stack:	Thread
memory management info:	Process
Program counter:	Thread
Parent identification:	Process
Thread state:	Thread
Registers:	Thread

Control Blocks

- The thread control block (TCB) contains:
  - Thread state, Program Counter, Registers

- PCB' = everything else (e.g. process id, open files, etc.)

- The process control block (PCB) = PCB' U TCB

Why use threads?

- Because threads have minimal internal state, it takes less time to create a thread than a process (10x speedup in UNIX).
- It takes less time to terminate a thread.
- It takes less time to switch to a different thread.
- A multi-threaded process is much cheaper than multiple (redundant) processes.

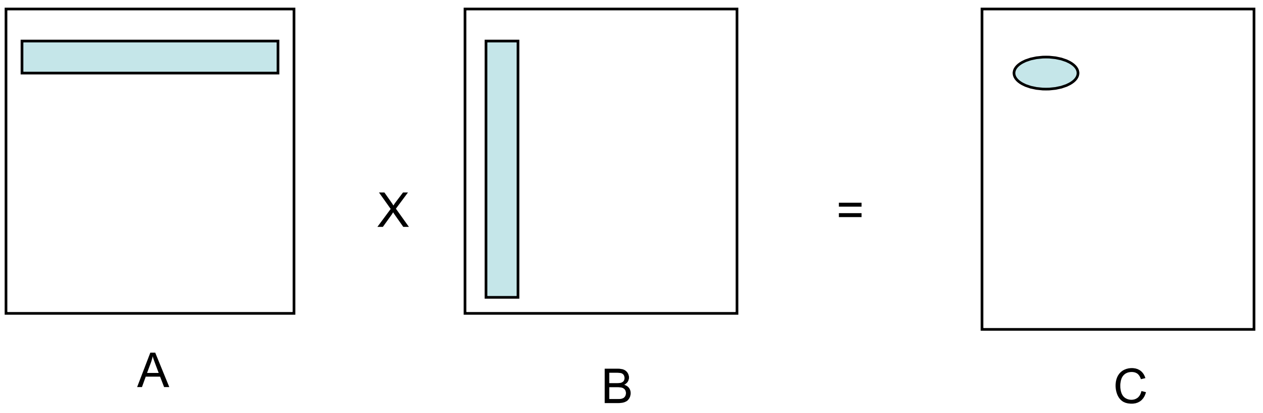
Examples of Using Threads

- Threads are useful for any application with multiple tasks that can be run with separate threads of control.
- A Word processor may have separate threads for:
  - User input
  - Spell and grammar check
  - displaying graphics
  - document layout
- A web server may spawn a thread for each client
  - Can serve clients concurrently with multiple threads.
  - It takes less overhead to use multiple threads than to use multiple processes.

Examples of multithreaded programs

- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism. This is called Multiprocessing

Multithreaded Matrix Multiply...



C[1,1] = A[1,1]*B[1,1]+A[1,2]*B[2,1]..

….

C[m,n]=sum of product of corresponding elements in row of A and column of B.

Each resultant element can be computed independently.

Multithreaded Matrix Multiply

```
typedef struct {
int id; int size;
int row, column;
matrix *MA, *MB, *MC;
} matrix_work_order_t;
main()
{
    int size = ARRAY_SIZE, row, column;
    matrix_t MA, MB,MC;
    matrix_work_order *work_orderp;
    pthread_t peer[size*size];
    ...
```

Multithreaded Matrix Multiply

```
/* process matrix, by row, column */
for( row = 0; row < size; row++ )
  for( column = 0; column < size; column++)
  {
     id = column + row * ARRAY_SIZE;
     work_orderp = malloc( sizeof(matrix_work_order_t));
    /* initialize all members if wirk_orderp */
    pthread_create(peer[id], NULL, peer_mult, work_orderp);
} }
/* wait for all peers to exist*/ for( i =0; i < size*size;i++)
                                              pthread_join( peer[i], NULL );
}
```

Benefits

- Responsiveness:
  - Threads allow a program to continue running even if part is blocked.
  - For example, a web browser can allow user input while loading an image.
- Resource Sharing:
  - Threads share memory and resources of the process to which they belong.
- Economy:
  - Allocating memory and resources to a process is costly.
  - Threads are faster to create and faster to switch between.
- Utilization of Multiprocessor Architectures:
  - Threads can run in parallel on different processors.
  - A single threaded process can run only on one processor no matter how many are available.

Thank You