

Master Informatics Eng.

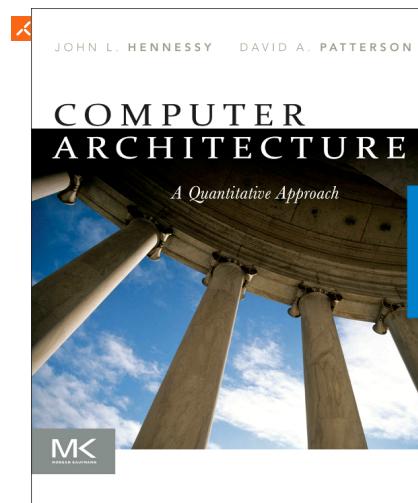
2016/17

A.J.Proen  a

Concepts from undegrad Computer Systems (2) (some slides are borrowed)

AJProen  a, Advanced Architectures, MiEI, UMinho, 2016/17

1



AJProen  a, Advanced Architectures, MiEI, UMinho, 2016/17

2

Computer Architecture, 5th Edition

Hennessy & Patterson

Table of Contents

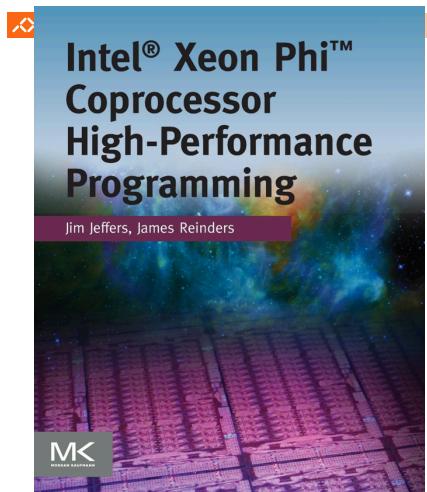
Printed Text

- Chap 1: Fundamentals of Quantitative Design and Analysis
- Chap 2: Memory Hierarchy Design
- Chap 3: Instruction-Level Parallelism and Its Exploitation
- Chap 4: Data-Level Parallelism in Vector, SIMD, and GPU Architectures
- Chap 5: Multiprocessors and Thread-Level Parallelism
- Chap 6: The Warehouse-Scale Computer
- App A: Instruction Set Principles
- App B: Review of Memory Hierarchy
- App C: Pipelining: Basic and Intermediate Concepts

Online

- App D: Storage Systems
- App E: Embedded Systems
- App F: Interconnection Networks
- App G: Vector Processors
- App H: Hardware and Software for VLIW and EPIC
- App I: Large-Scale Multiprocessors and Scientific Applications
- App J: Computer Arithmetic
- App K: Survey of Instruction Set Architectures
- App L: Historical Perspectives

Recommended textbook (1)



Contents

1. Introduction
2. High Performance examples
3. Benchmarking Apps
4. Real-world Situations
5. Lots of Data (Vectors)
6. Lots of Tasks (not Threads)
7. Processing Parallelism
8. Coprocessor Architecture
9. Coprocessor System Software
10. Linux on the Coprocessor
11. Math Library
12. MPI
13. Profiling
14. Summary

AJProen  a, Advanced Architectures, MiEI, UMinho, 2016/17

3

Recommended textbook (2)

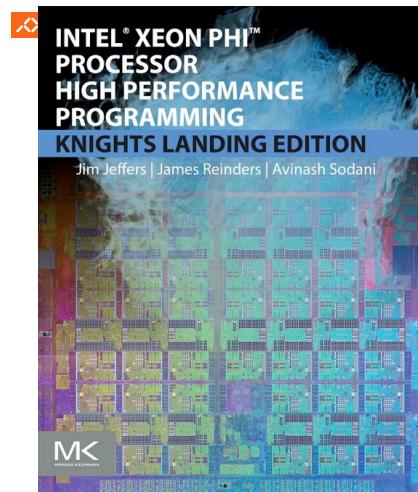


Table of Contents

Section I: Knights Landing.

- Chapter 1: Introduction
- Chapter 2: Knights Landing Overview
- Chapter 3: Programming MCDRAM and Cluster Modes
- Chapter 4: Knights Landing Architecture
- Chapter 5: Intel Omni-Path Fabric
- Chapter 6: *mpack* Optimization Advice

Section II: Parallel Programming

- Chapter 7: Programming Overview for Knights Landing
- Chapter 8: Tasks and Threads
- Chapter 9: Vectorization
- Chapter 10: Vectorization Advisor
- Chapter 11: Vectorization with SDLT
- Chapter 12: Vectorization with AVX-512 *Intrinsics*
- Chapter 13: Performance Libraries
- Chapter 14: Profiling and Timing
- Chapter 15: MPI
- Chapter 16: PGAS Programming Models
- Chapter 17: Software Defined Visualization
- Chapter 18: Offload to Knights Landing
- Chapter 19: Power Analysis

Section III: Pearls

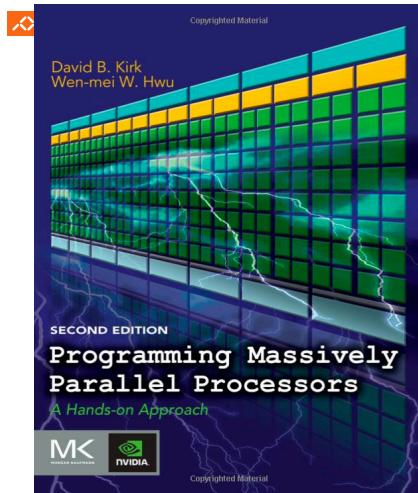
- Chapters 20-26: Results on LAMMPS, SeisSol, WRF, N-Body Simulations, Machine Learning, Trinity mini-applications and QCD are discussed.

AJProen  a, Advanced Architectures, MiEI, UMinho, 2016/17

4

Recommended textbook (3)

Understanding Performance



Contents

- 1 Introduction
- 2 History of GPU Computing
- 3 Introduction to Data Parallelism and CUDA C
- 4 Data-Parallel Execution Model
- 5 CUDA Memories
- 6 Performance Considerations
- 7 Floating-Point Considerations
- 8 Parallel Patterns: Convolution
- 9 Parallel Patterns: Prefix Sum
- 10 Parallel Patterns: Sparse Matrix-Vector Multiplication
- 11 Application Case Study: Advanced MRI Reconstruction
- 12 Application Case Study: Molecular Visualization and Analysis
- 13 Parallel Programming and Computational Thinking
- 14 An Introduction to OpenCL
- 15 Parallel Programming with OpenACC
- 16 Thrust: A Productivity-Oriented Library for CUDA
- 17 CUDA FORTRAN
- 18 An Introduction to C11 AMP
- 19 Programming a Heterogeneous Computing Cluster
- 20 CUDA Dynamic Parallelism
- 21 Conclusion and Future Outlook

AJProen , Advanced Architectures, MiEI, UMinho, 2016/17

5



Algorithm + Data Structures

- Determines number of operations executed
- Determines how efficient data is assessed
- Programming language, compiler, architecture
 - Determine number of machine instructions executed per operation
- Processor and memory system
 - Determine how fast instructions are executed
- I/O system (including OS)
 - Determines how fast I/O operations are executed

AJProen , Advanced Architectures, MiEI, UMinho, 2016/17

6

COD: Chapter 1 — Computer Abstractions and Technology

Response Time and Throughput

- Response time
 - How long it takes to do a task
- Throughput
 - Total work done per unit time
 - e.g., tasks/transactions/... per hour
- How are response time and throughput affected by
 - Replacing the processor with a faster version?
 - Adding more processors?
- We'll focus on response time for now...

COD: Chapter 1 — Computer Abstractions and Technology

CPU Time

(single-core)

$$\begin{aligned} \text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \end{aligned}$$

- Performance improved by
 - Reducing number of clock cycles
 - Increasing clock rate
 - Hardware designer must often trade off clock rate against cycle count

AJProen , Advanced Architectures, MiEI, UMinho, 2016/17

8

COD: Chapter 1 — Computer Abstractions and Technology

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count, **IC**, for a program
 - Determined by program, ISA and compiler
- Average cycles per instruction (**CPI**)
 - Determined by CPU hardware
 - If different instructions have different CPI
 - Average CPI affected by instruction mix

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} \times T_c$$

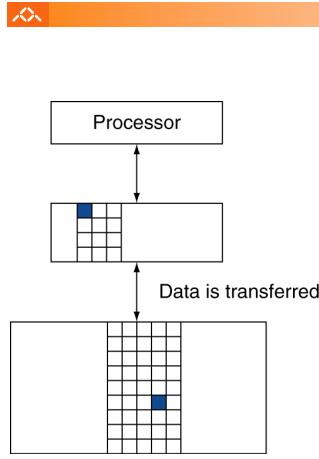
- Performance depends on
 - Algorithm: affects IC, possibly CPI
 - Programming language: affects IC, CPI
 - Compiler: affects IC, CPI
 - Instruction set architecture: affects IC, CPI, T_c
 - Processor design: ILP, memory hierarchy, ...

Pipeline Summary**Does Multiple Issue Work?****The BIG Picture**

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
 - e.g., pointer aliasing
- Some parallelism is hard to expose
 - Limited window size during instruction issue
- Memory delays and limited bandwidth
 - Hard to keep pipelines full
- Speculation can help if done well



- Block (aka line): unit of copying
 - May be multiple words
- If accessed data is present in upper level
 - Hit: access satisfied by upper level
 - Hit ratio: hits/accesses
- If accessed data is absent
 - Miss: block copied from lower level
 - Time taken: miss penalty
 - Miss ratio: misses/accesses = 1 – hit ratio
 - Then accessed data supplied from lower level

The BIG Picture

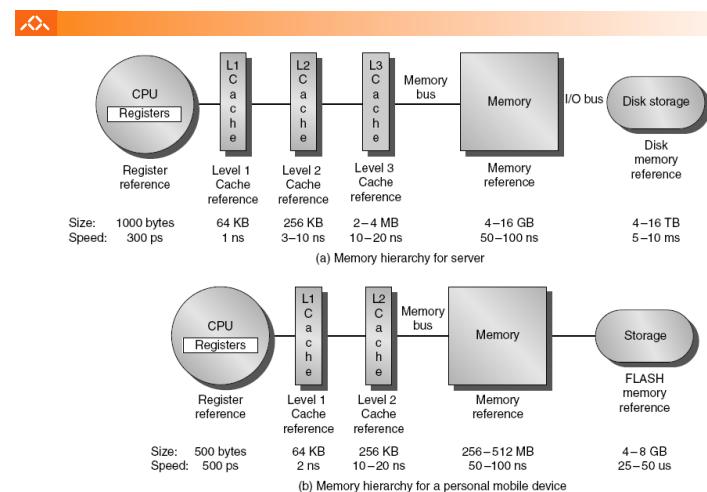
- Common principles apply at all levels of the memory hierarchy
 - Based on notions of caching
- Decisions at each level in the hierarchy
 - Block placement
 - Finding a block
 - Replacement on a miss
 - Write policy

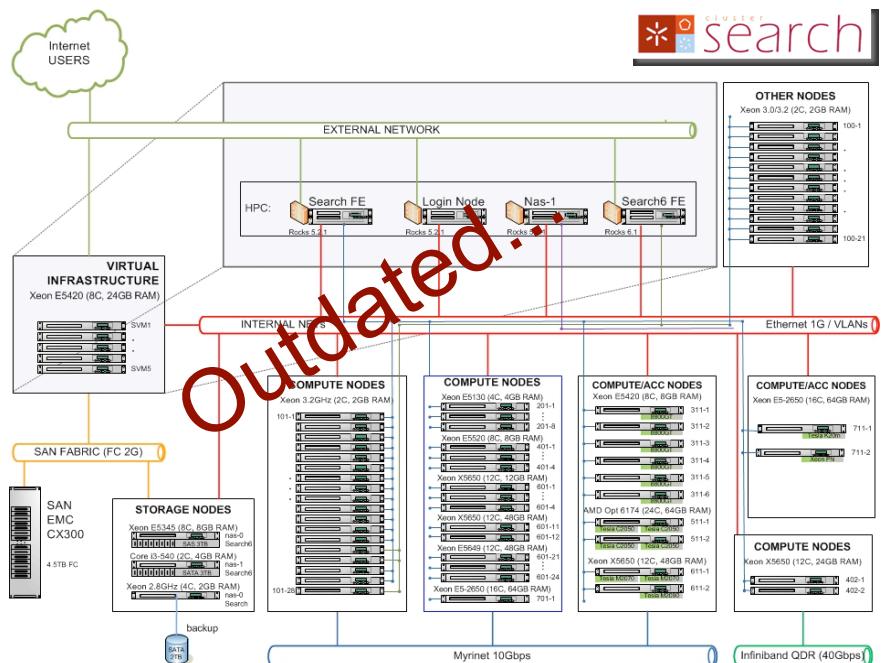
§5.5 A Common Framework for Memory Hierarchies

Multilevel Caches

- Primary cache private to CPU/core
 - Small, but fast
- Level-2 cache services misses from primary cache
 - Larger, slower, but still faster than main memory
- High-end systems include L3 cache
- Main memory services L2/3 cache misses

Memory Hierarchy

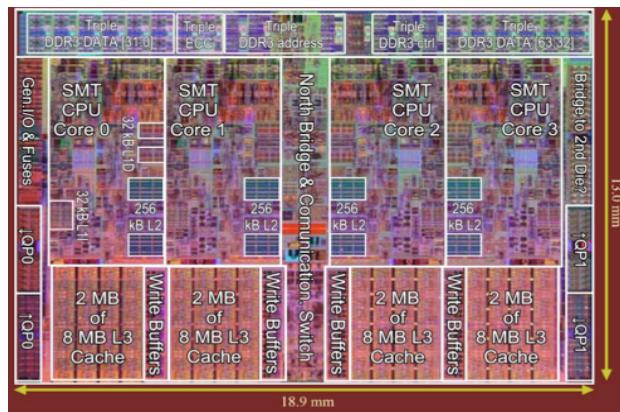




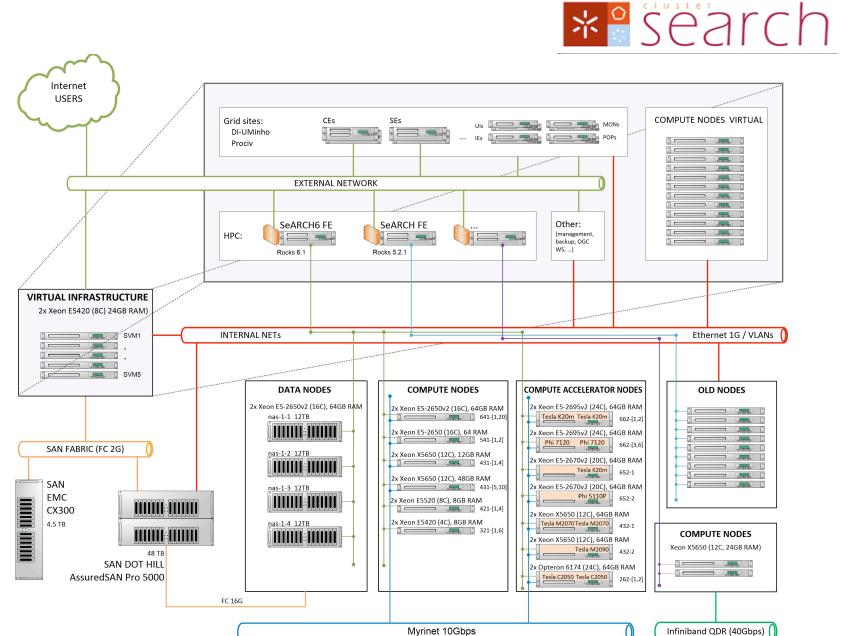
The diagram illustrates the internal network architecture. It shows two racks of compute nodes, each containing four server units. The racks are connected to an internal network switch labeled "INTERNAL NET". A red box highlights the connection between the racks and the switch.

Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache



Internal x86 roadmap

CCOD: Chapter 5 — Large and Fast: Exploiting Memory Hierarchy

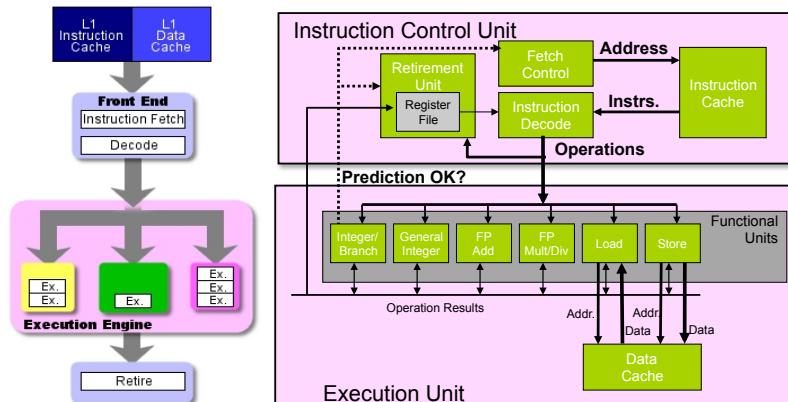
NetBurst														
Willamette	Northwood	Prescott	Tejas	Nehalem (NetBurst)										
			✓	Cedar Mill (Tejas)										
			✓	Prescott-2M	Cedar Mill									
				Smithfield	Presler									
P6					Core	Nehalem		Sandy Bridge		Haswell	Skylake			
Coppermine	Tualatin	Banias	Dothan	Yonah	Core	Penryn	Nehalem	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake	Cannonlake
180 nm	130 nm	90 nm	65 nm		45 nm	32 nm		22 nm		14 nm	10 nm			
P5					Atom	Bonnell		Saltwell		Silvermont	Airmont	Goldmont		
Xeon Phi					{	Bonnel		Saltwell		Silvermont	Airmont	Goldmont		
					Xeon Phi					Knights Corner	Knights Landing			



Internal architecture of Intel P6 processors



Note: "Intel P6" is the common pArch name for PentiumPro, Pentium II & Pentium III, which inspired Core, Nehalem and Phi



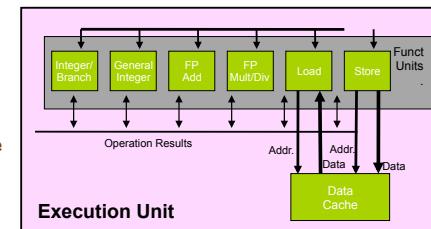
AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

21

Some capabilities of Intel P6



- Parallel execution of several instructions
 - 2 integer (1 can be branch)
 - 1 FP Add
 - 1 FP Multiply or Divide
 - 1 load
 - 1 store



- Some instructions require > 1 cycle, but can be pipelined:

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

22

A detailed example: generic & abstract form of combine



```
void abstract_combine4(vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t t = IDENT;
    for (i = 0; i < length; i++)
        t = t OP data[i];
    *dest = t;
}
```

- Procedure to perform addition (w/ some improvements)
 - compute the sum of all vector elements
 - store the result in a given memory location
 - structure and operations on the vector defined by ADT
- Metrics
 - Clock-cycles Per Element, CPE

AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

23

Converting instructions with registers into operations with tags



- Assembly version for `combine4`
 - data type: `integer`; operation: `multiplication`

```
.L24:          # Loop:
    imull (%eax,%edx,4),%ecx # t *= data[i]
    incl %edx                 # i++
    cmpl %esi,%edx           # i:length
    jl .L24                  # if < goto Loop
```

- Translating 1st iteration

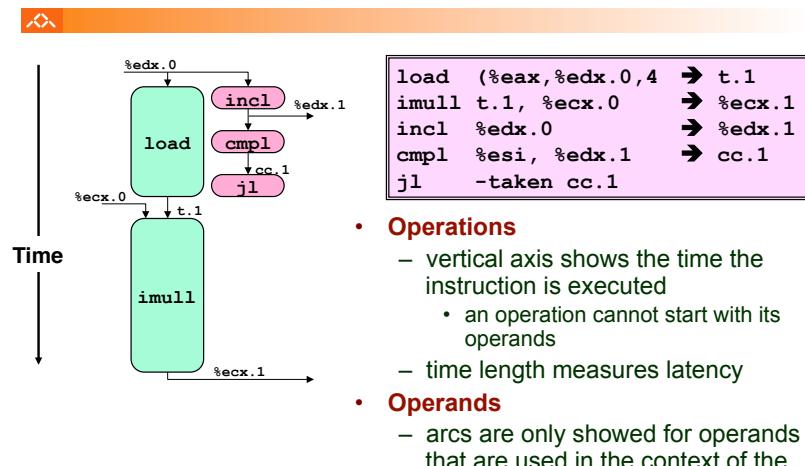
```
.L24:
    imull (%eax,%edx,4),%ecx
    incl %edx
    cmpl %esi,%edx
    jl .L24
```

```
load (%eax,%edx.0,4)  t.1
imull t.1, %ecx.0      %ecx.1
incl %edx.0             %edx.1
cmpl %esi, %edx.1      cc.1
jl -taken cc.1
```

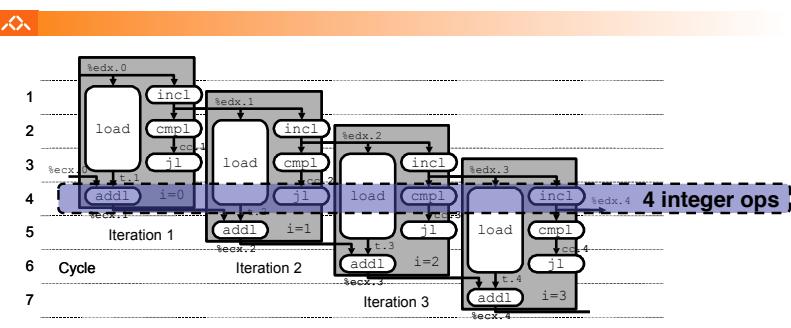
AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

24

Visualizing instruction execution in P6: 1 iteration of the multiplication cycle on combine



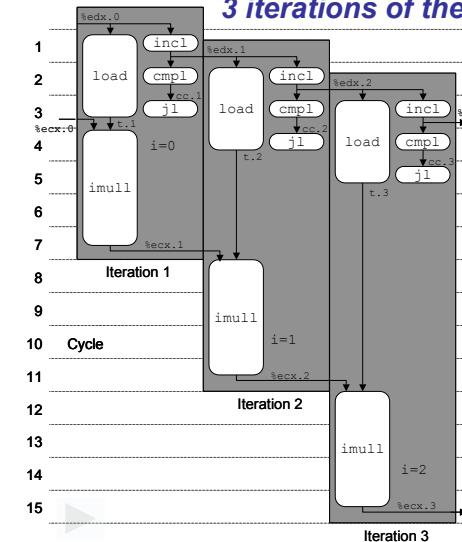
Visualizing instruction execution in P6: 4 iterations of the addition cycle on combine



With unlimited resources

Performance

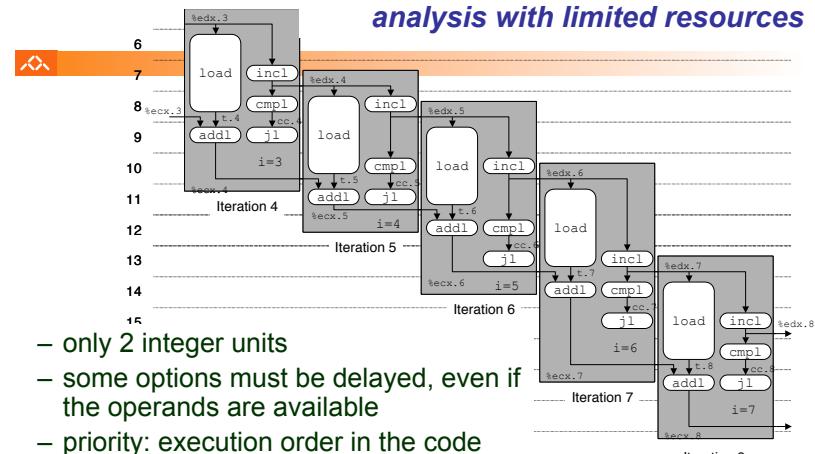
Visualizing instruction execution in P6: 3 iterations of the same cycle on combine



With unlimited resources

Performance

Iterations of the addition cycles: analysis with limited resources



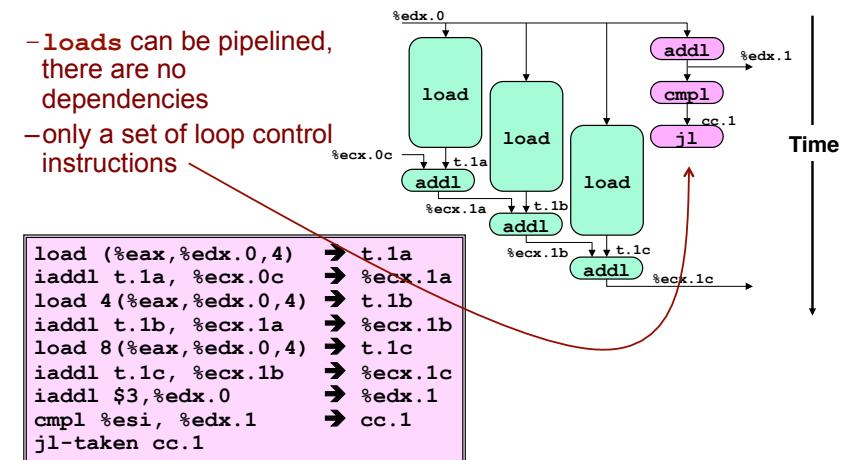
Machine dependent optimization techniques: loop unroll (1)

```
void combine5(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-2;
    int *data = get_vec_start(v);
    int sum = 0;
    int i;
    /* junta 3 elem's no mesmo ciclo */
    for (i = 0; i < limit; i+=3) {
        sum += data[i] + data[i+1]
            + data[i+2];
    }
    /* completa os restantes elem's */
    for (; i < length; i++) {
        sum += data[i];
    }
    *dest = sum;
}
```

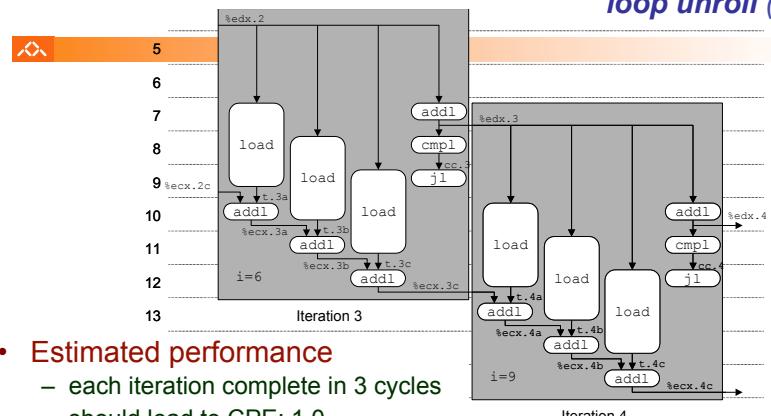
Optimization 4:

- merges several (3) iterations in a single loop cycle
- reduces cycle overhead in loop iterations
- runs the extra work at the end
- CPE: 1.33

Machine dependent optimization techniques: loop unroll (2)



Machine dependent optimization techniques: loop unroll (3)



- Estimated performance
 - each iteration complete in 3 cycles
 - should lead to CPE: 1.0
- Measured performance
 - CPE: 1.33
 - 1 iteration for each 4 cycles

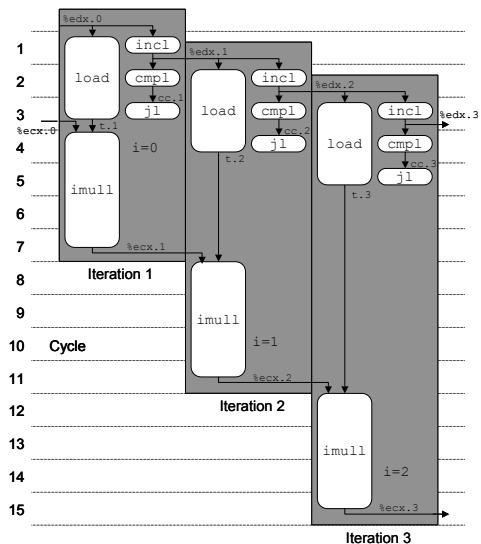
Machine dependent optimization techniques: loop unroll (4)

CPE value for several cases of loop unroll:

Degree of Unroll	1	2	3	4	8	16
Integer Addition	2.00	1.50	1.33	1.50	1.25	1.06
Integer Product					4.00	
fp Addition					3.00	
fp Product					5.00	

- only improves the integer addition
 - remaining cases are limited to the unit latency
- result does not linearly improve with the degree of unroll
 - subtle effects determine the exact allocation of operations

What else can be done?



AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

33

Machine dependent optimization techniques: loop unroll with parallelism (1)

Sequential ... versus parallel!

Optimization 5:

- accumulate in 2 different products
 - can be in parallel, if OP is associative!
- merge at the end
- Performance
 - CPE: 2.0
 - improvement 2x

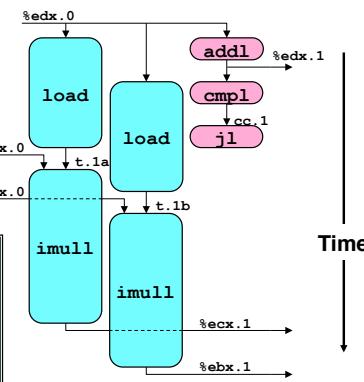
```
void combine6(vec_ptr v, int *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    int *data = get_vec_start(v);
    int x0 = 1;
    int x1 = 1;
    /* junta 2 elem's de cada vez */
    for (i = 0; i < limit; i+=2) {
        x0 *= data[i];
        x1 *= data[i+1];
    }
    /* completa os restantes elem's */
    for (; i < length; i++) {
        x0 *= data[i];
    }
    *dest = x0 * x1;
}
```

AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

34

Machine dependent optimization techniques: loop unroll with parallelism (2)

- each product at the inner cycle does not depend from the other one...
- so, they can be pipelined
- known as iteration splitting

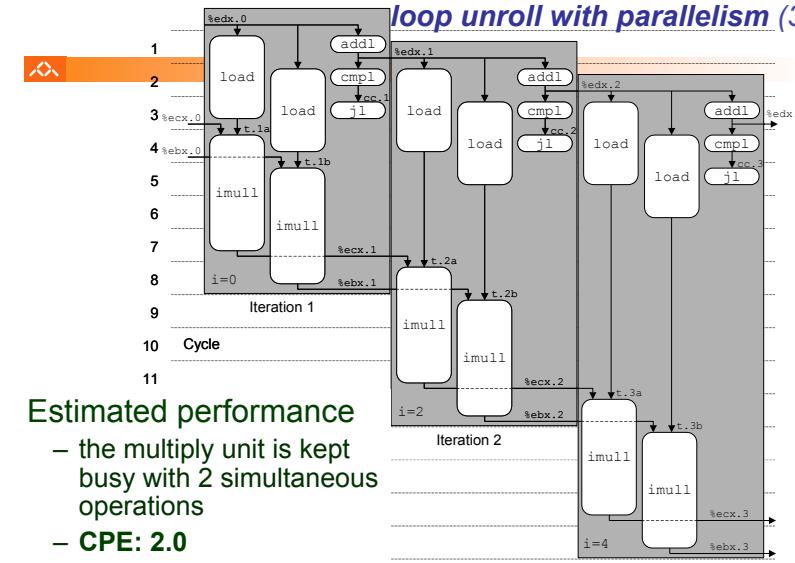


```
load (%eax,%edx.0,4)    ➔ t.1a
imull t.1a, %ecx.0       ➔ %ecx.1
load 4(%eax,%edx.0,4)   ➔ t.1b
imull t.1b, %ebx.0       ➔ %ebx.1
iaddl $2,%edx.0          ➔ %edx.1
cmp1 %esi, %edx.1        ➔ cc.1
jl-taken cc.1
```

AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

35

Machine dependent optimization techniques: loop unroll with parallelism (3)



Estimated performance

- the multiply unit is kept busy with 2 simultaneous operations
- CPE: 2.0

AJProen  , Advanced Architectures, MiEI, UMinho, 2016/17

36

Method	Integer		Real (single precision)	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
Access to data	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00
Unroll 4x	1.50	4.00	3.00	5.00
Unroll 16x	1.06	4.00	3.00	5.00
Unroll 2x, paral. 2x	1.50	2.00	2.00	2.50
Unroll 4x, paral. 4x	1.50	2.00	1.50	2.50
Unroll 8x, paral. 4x	1.25	1.25	1.50	2.00
Theoretical Optimiz	1.00	1.00	1.00	2.00
Worst : Best	39.7	33.5	27.6	80.0

It requires a lot of registers!

- to save results from add/multip
- only 6 integer registers in IA32
 - also used as pointers, loop control, ...
- 8 fp registers
- when registers aren't enough, temp's are pushed to the stack
 - cuts performance gains
(see assembly in integer product with 8x unroll & 8x parallelism)
- re-naming registers is not enough
 - it is not possible to reference more operands than those at the instruction set
 - ... main drawback at the IA32 instruction set

Operations to parallelize must be associative!

- fp add & multipl in a computer is not associative!
 - $(3.14+1e20)-1e20$ not always the same as $3.14+(1e20-1e20)$...

Limitation of parallelism: not enough registers

Homework

combine

- integer multiplication
- 8x unroll & 8x parallelism
- 7 local variables share 1 register (%edi)
 - note the stack accesses
 - performance improvement is compromised...
 - consequence: register spilling

```
.L165:
imull (%eax),%ecx
movl -4(%ebp),%edi
imull 4(%eax),%edi
movl %edi,-4(%ebp)
movl -8(%ebp),%edi
imull 8(%eax),%edi
movl %edi,-8(%ebp)
movl -12(%ebp),%edi
imull 12(%eax),%edi
movl %edi,-12(%ebp)
movl -16(%ebp),%edi
imull 16(%eax),%edi
movl %edi,-16(%ebp)
...
addl $32,%eax
addl $8,%edx
cmpl -32(%ebp),%edx
jl .L165
```

Identify the 4 more recent Intel Xeon processor microarchitectures in the SeARCH cluster and build a table with the following column headings:

- chip ID (e.g., Xeon E5345)
-   architecture (e.g., Haswell)
- #pipeline stages (e.g., 14 pipe stages)
- #simultaneous_threads per core (e.g., 2-way SMT/HyperThreading)
- degree of superscalarity (e.g., 3-issue wide)
- vector support (e.g., 128-bit SIMD)
- #cores
- #cache levels on-chip & size (e.g., (32KB-Intrs + 32KB Data)L1, 256KB L3, 4x2MB L3)
- type/speed to connect CPU-chips