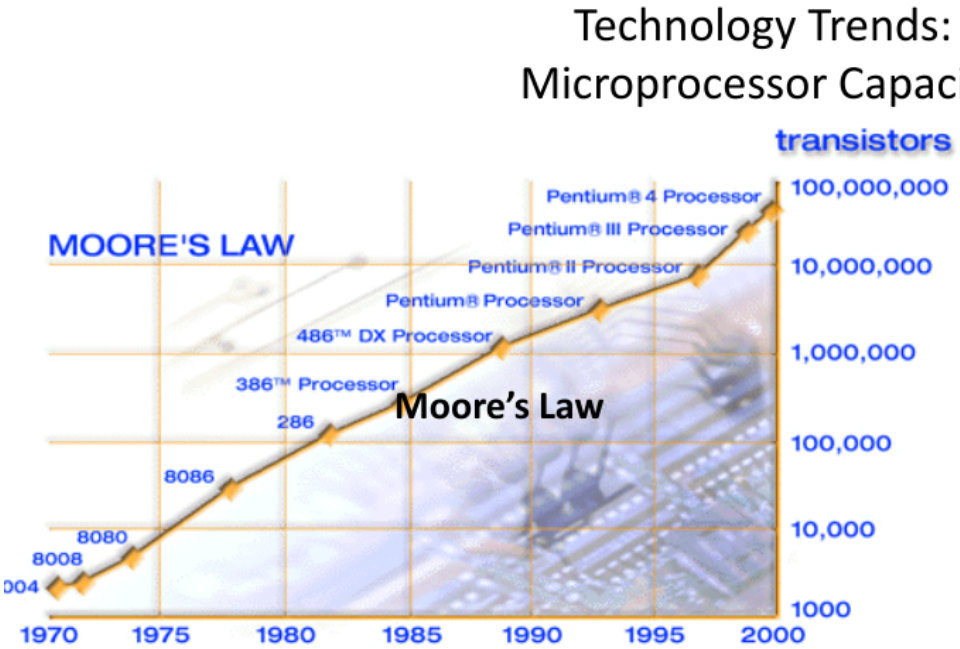
Computer architecture techniques and power dissipation

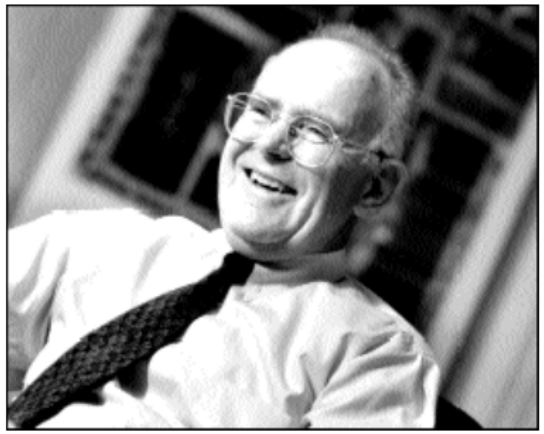
Oscar Palomar

Barcelona Supercomputing Center

July 2014

Technology Trends





2X transistors/Chip Every 1.5 years

Called "Moore's Law"

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Microprocessors have become smaller, denser, and more powerful. Not just processors, bandwidth, storage, etc

"Runtime Aware Architectures", Mateo Valero, HiPEAC CSW 2014, Barcelona.

Design goals

- **Performance**
- Area
- Power/energy
- *Football is like a short blanket, if you cover your head, you uncover your feet, and if you cover your feet, you uncover your head* (Elba de Pádua)
- Priorities depend on purpose:
  server, desktop, laptop, mobile, embedded
- But computer architecture has a historical strong bias towards performance...
  until forced to do otherwise (power wall)

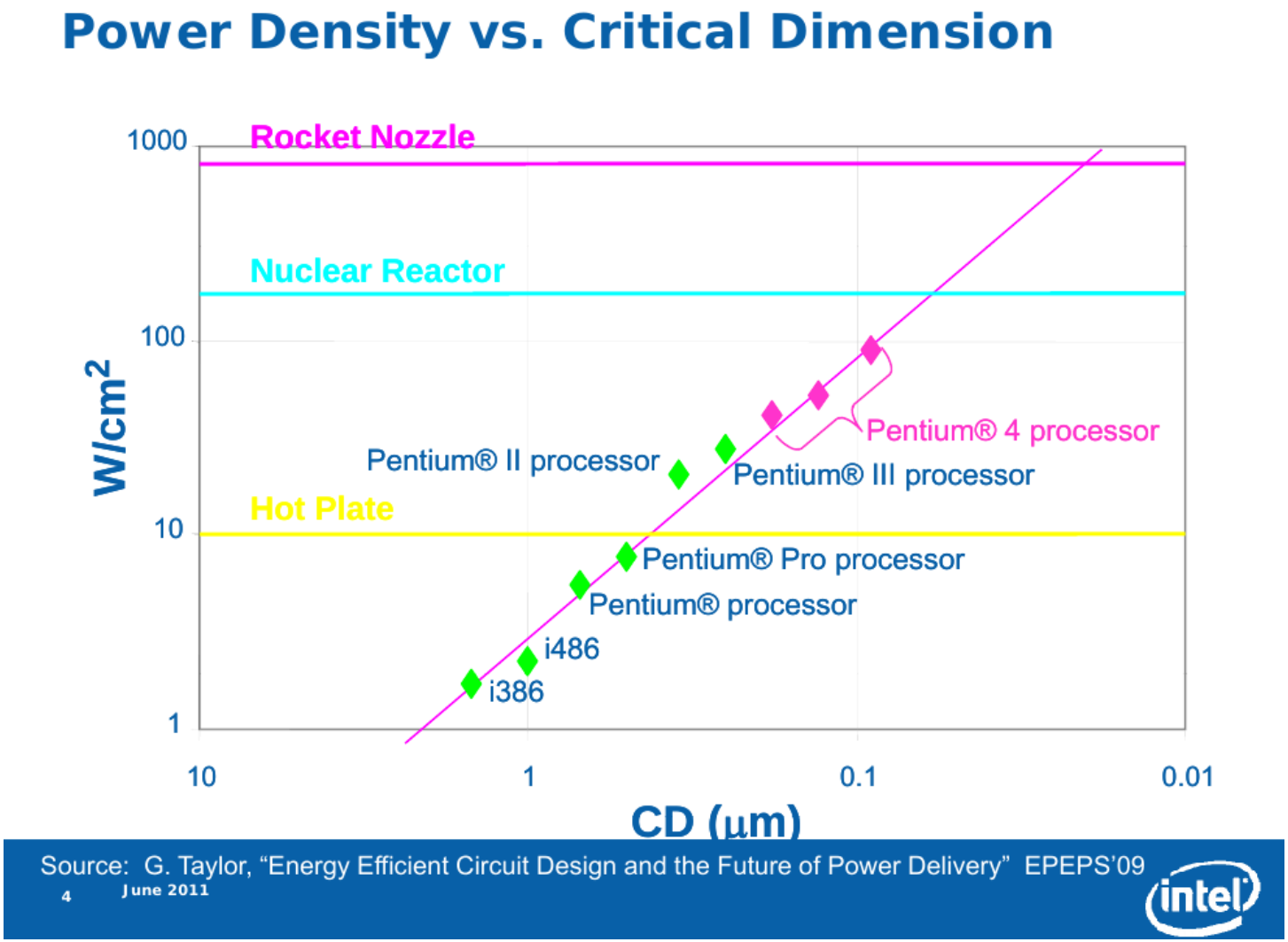
What does performance mean?

- Latency: time until task completion
  - $\mathrm{T_{cpu}} = \mathrm{cycles} * \mathrm{T_{cycle}} = \mathrm{inst} * \mathrm{CPI} * \mathrm{T_{cycle}}$
  - Dynamic instruction count
  - CPI: Cycles per instruction
- Throughput: amount of work made per time unit (e.g. bandwidth)
  - IPC: Instructions per cycle
  - $CPI = 1/IPC$
  - Simultaneous Multi-threading (SMT): improves throughput, not latency
  - Memory bandwidth, network bandwidth
  - Application specific: webpages/second, queries/second...

Latency vs. Throughput, Peak vs. Average

- User: latency
- System: throughput
- Better latency -> better throughput
- Better throughput with equal latency
- Peak IPC does not matter, we want high average IPC
- We want low power in avg, but peak DOES matter
  -> temp cooling, power supply requirements

The Power Wall



Power and Energy

- $\text{Power}_{\text{dyn}} = 1/2 * \text{Capacitive load} * \text{Voltage}^2 * \text{Freq}$
- $\text{Power}_{\text{static}} = \text{Current}_{\text{static}} * \text{Voltage}$
- $\text{Energy}_{\text{dyn}} = \text{Capacitive load} * \text{Voltage}^2$
- Frequency scales roughly linearly with voltage
- $\text{Power} \propto \text{V}^3$
- 1% extra performance translates 3% extra power
- This means that power-saving techniques must reduce 3x power w.r.t. performance degradation

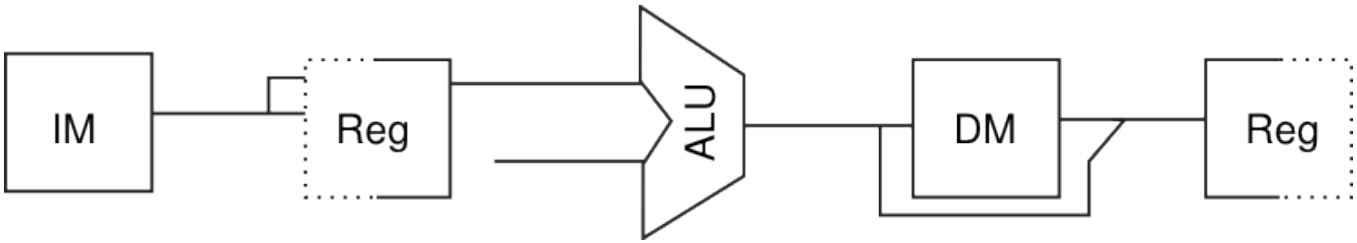
The rest of the talk

- Some classic computer architecture techniques with their implications w.r.t power
  - Pipelining
  - Cache hierarchy
  - Speculation
  - Out-of-order execution
- The ParaDIME project and current C.A. trends
  - Multi/many-core processors
  - Heterogeneous computing
  - Reduced precision

Pipelining

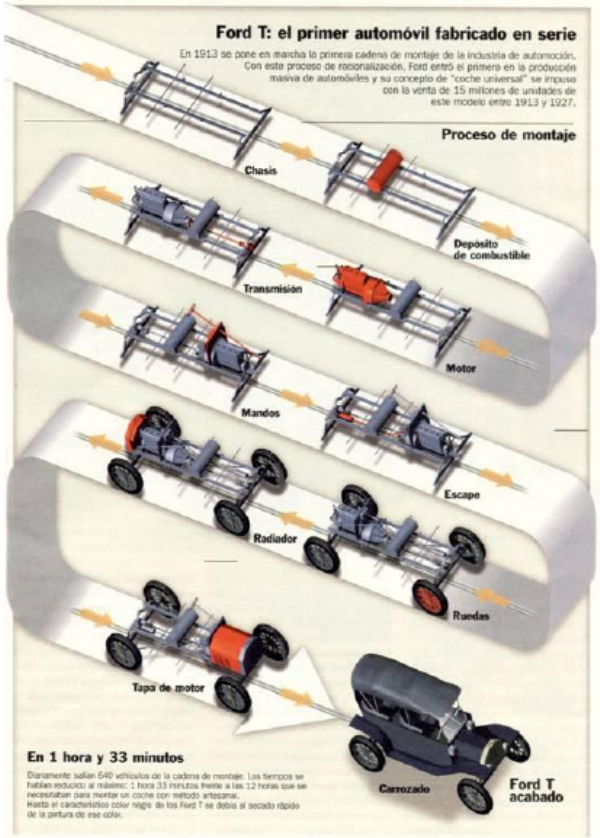
Motivation

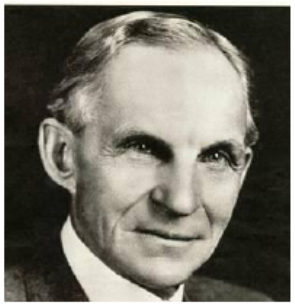
- Programmer assumes sequential execution of each inst
- Instruction execution: sequential use of proc. logic
  - Read instruction from memory
  - Decode instruction
  - Read registers
  - Operate data
  - Write result
- When a given structure is used, the others are idle
- If inst must complete before executing the next one, the resources of the processor are underutilized

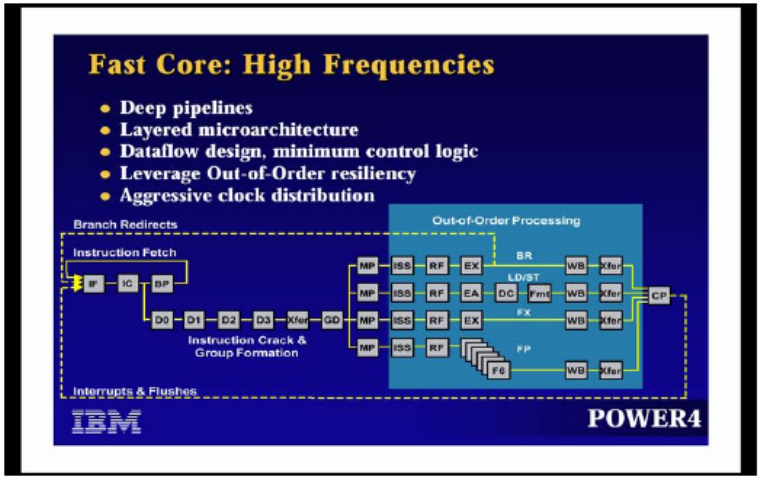


"Computer Architecture: A Quantitative Approach", H. and P., pp. A-8, 4th Ed, 2006.

Historical perspective







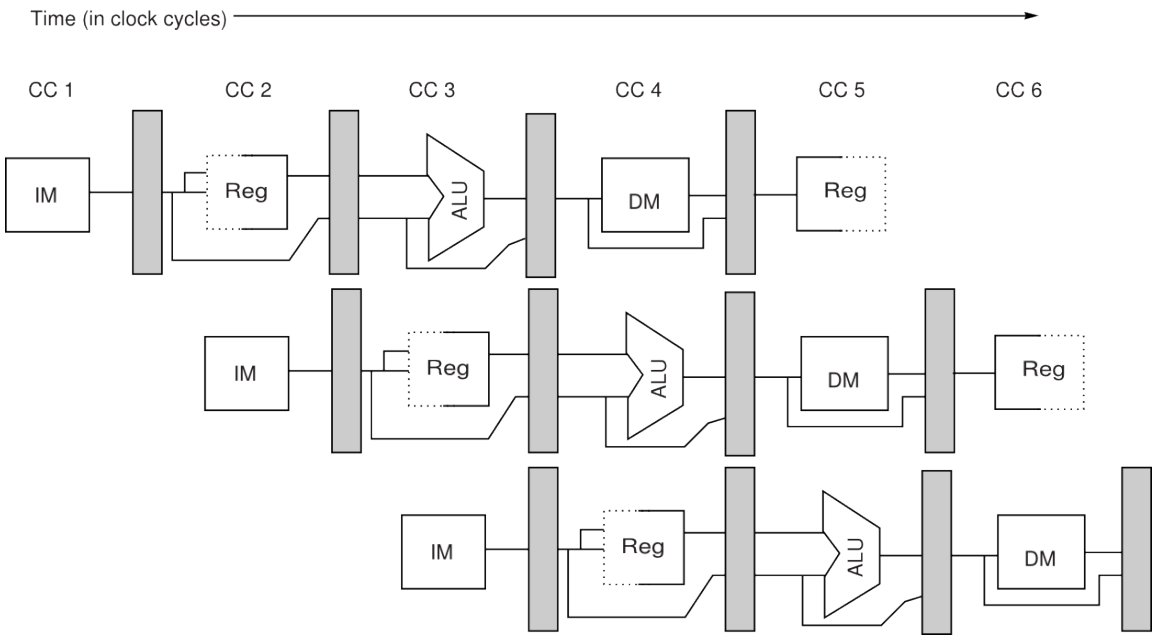
"Runtime Aware Architectures", Mateo Valero, HiPEAC CSW 2014, Barcelona.

Implementation: Divide and Conquer

- Divide execution in several stages (pipeline)
- Instructions progress through the pipeline
- So overlapped execution of several instructions
- Ideal IPC = 1 (improves throughput,
  1 inst same latency, overall time improves)
- Control logic is more complex
- Multiple control (PC, Instruction), operands
  -> stage latches

Pipeline stages

- Fetch stage: read inst and update PC
- Decode stage: decode inst and read reg
- Execute stage: perform operation
- Memory stage: access memory if needed
- Write stage: update dst register



"Computer Architecture: A Quantitative Approach", H. and P., pp. A-9, 4th Ed, 2006.

Hazards

- Data: inst reads the result of previous inst. Example:

```
ADD R1, R2, R3; F|D|X|M|W
SUB R3, R4, R5;   F|D|X|M|W
```

- Control: Control inst evaluated (condition, next PC) before fetching next inst
- Structural: Two inst need to use the same structure
- Simplest solution: stall (wait) until hazard disappears: result is written, next PC is known, structure is free
- Stall -> low IPC, so more complex solutions are used

```
ADD R1, R2, R3; F|D|X|M|W
SUB R3, R4, R5;   F|S|S|D|X|M|W
```

Bypasses

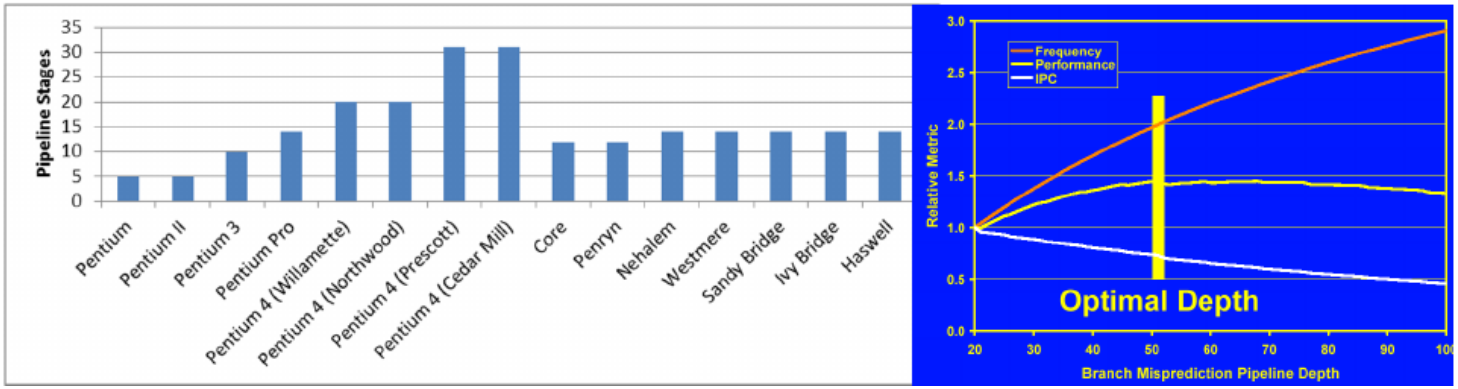
- Avoid stalls in data hazards, a.k.a. Forwarding
- The result is available after stage X
  but we must wait to update and read registers
- The result is needed before beginning of stage X
- Bypass: data sent from producer to consumer ASAP

```
ADD R1, R2, R3; F|D|X|M|W
SUB R3, R4, R5;   F|D|X|M|W
```

- Introduces muxes to select the source of the operand

Pipelining++

- Superpipelining: more stages. Higher frequency. Several fetch stages, etc. Pentium 4: 20 stages (31 stages in Prescott)
- Drawbacks: latch overhead, tight loops
- Superscalar: process several inst per cycle. Higher IPC. Ideal IPC > 1.0
- Drawbacks: more complex logic, hazards, bypasses



"Runtime Aware Architectures", Mateo Valero, HiPEAC CSW 2014, Barcelona.

Power

- Higher power as a direct consequence of higher clock frequency. Pentium 4: 20-31 stages
- Overhead of pipeline stage latches
- Overall: quite simple additional hardware

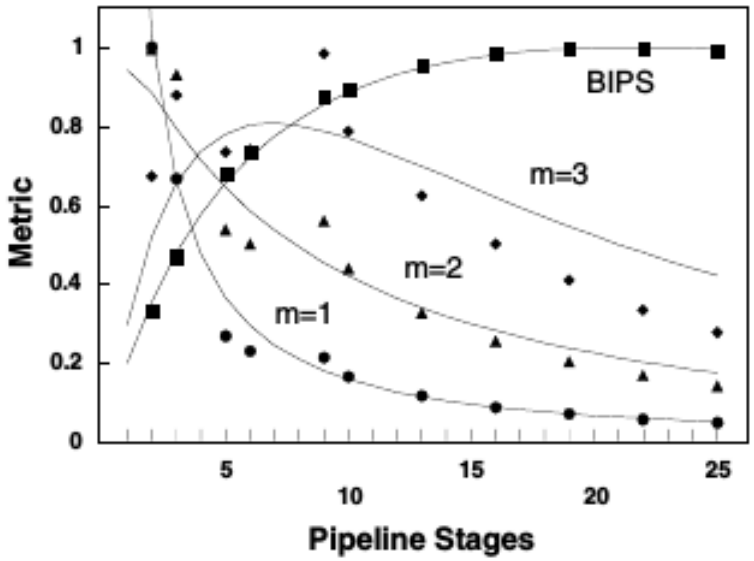


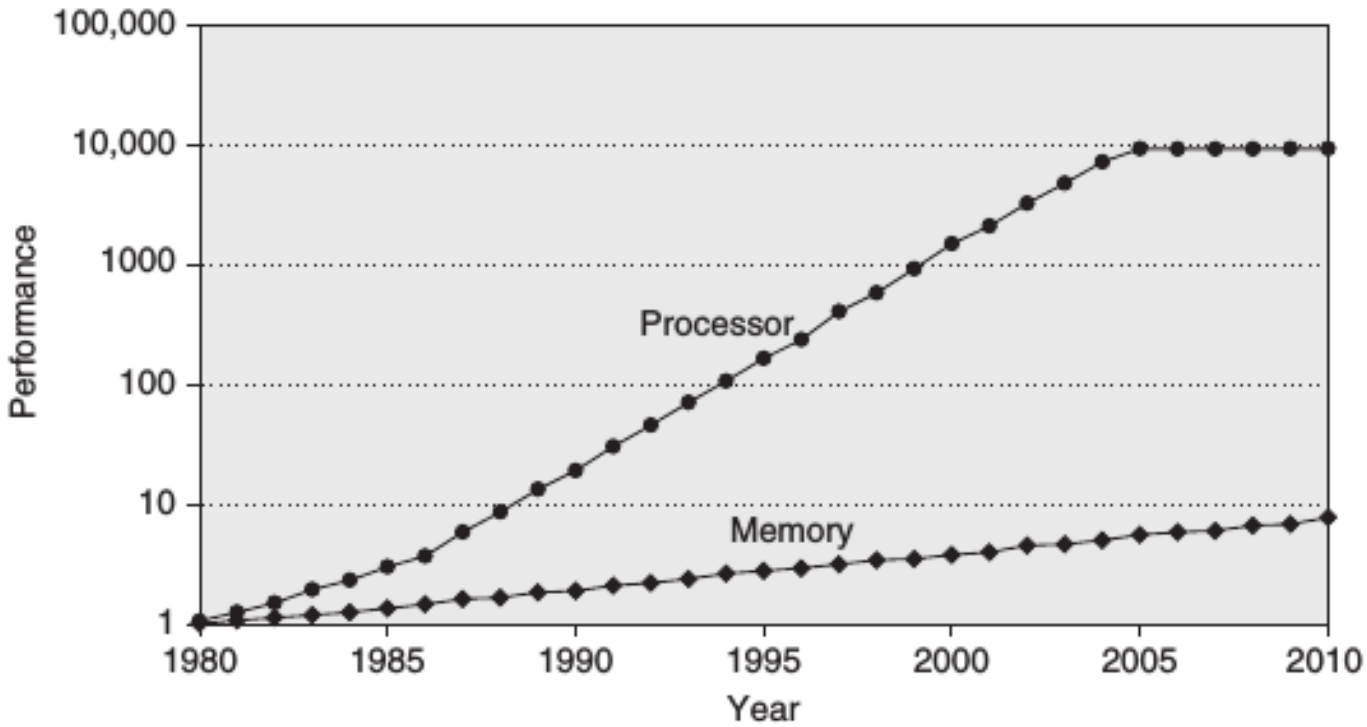
Fig. 5 shows four different metrics as a function of pipeline depth. Both theory and simulation are shown.

"Optimum Power/Performance Pipeline Depth", A. Hartstein and Thomas R. Puzak, MICRO-36 2003.

The cache hierarchy

The memory wall

- Increasing gap in latency, many cycles to do an access
- $\text{Tcycle}_{\text{cpu}} << \text{Taccess}_{\text{mem}}$
- We want large memories, which have higher latencies



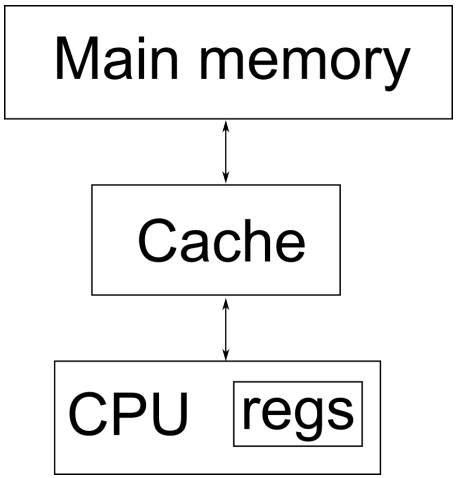
"Computer Architecture: A Quantitative Approach", H. and P., 5th Ed, 2012.

Locality

- Memory instructions amongst the most used
- But some @ are accessed more
- For instance, inst in a loop
- This property is called *locality*
- Temporal locality: reuse data
  - After accessing @x, it is likely to access @x+1 soon
- Spatial locality: use neighbor data
  - After accessing @x, it is likely to access @x again soon
- 90/10 rule of thumb: 90% of accesses to 10% of @

Benefit from locality

- Put most accessed data in fast (but small) SRAM (cache)
- Place the rest in large (but slow) DRAM (main memory)
- Detection of most accessed data? Locality
- Temp loc. -> on first access to @ copy data to cache
- Spatial locality -> store also neighbor data
  Memory is divided in blocks of consecutive words
- Two possibilities when accessing the cache
  - Hit: desired block is in cache -> read data
  - Miss: otherwise -> bring it from next level



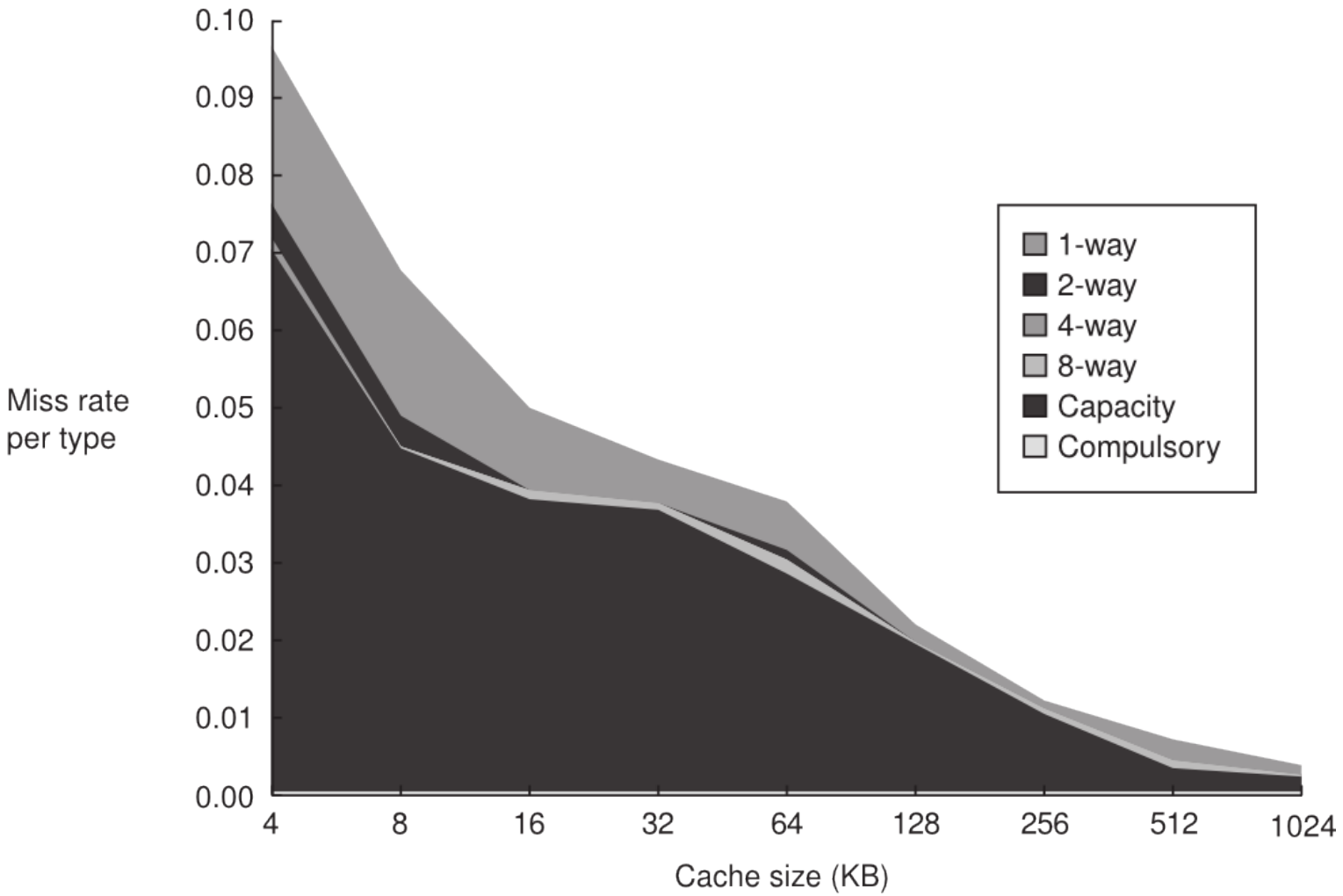
Reducing Misses

- $\mathrm{MissRate} = \mathrm{Misses}/\mathrm{Accesses}$: as low as possible
- Different techniques depending on miss class
  - Larger cache to store more blocks
  - Pre-fetch: access blocks before actually needed (SW/HW)
  - Change software (e.g. blocking)
  - Associativity: allow placing block in different lines

Placement policies

- Direct Mapped: lower bits of block @ index cache line
  Simple logic, many conflict misses.
- Fully Associative: block can be placed in any cache line
  Compare each cache line's tag
  (CAM: Content Addressable Memory)
  Complex (slow or small) logic, fewer misses.
- k-Way Set-Associative: lower bits of block @
  index a set of k cache lines. F.A. inside the set.
  8-Way near same miss rate as F.A.
- Higher associativity increases power. Can be limited by restricting parallel lookups

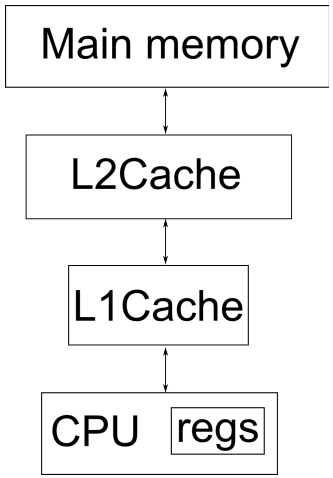
Miss Rate

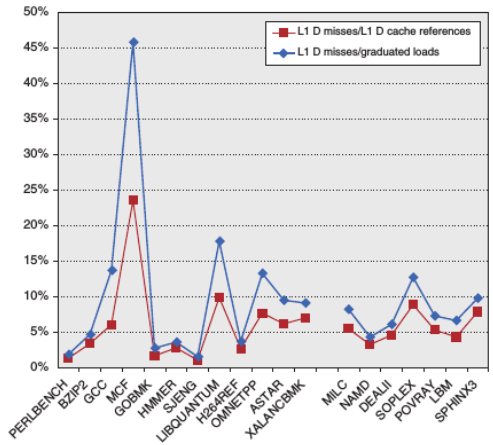


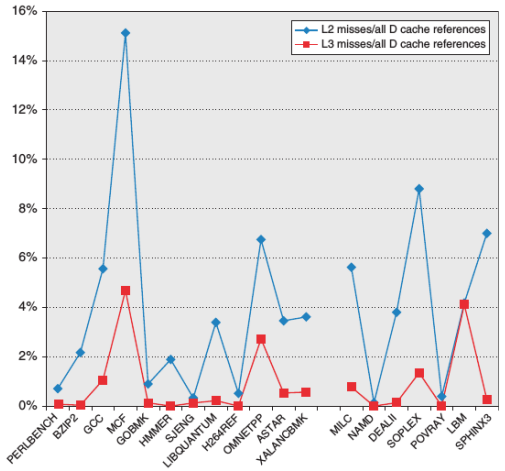
"Computer Architecture: A Quantitative Approach", H. and P., pp. C-24, 4th Ed, 2006.

Improvements: Multi-level Caches

- Memory wall: too much penalty if cache miss
- Place another cache level (L2)
- Larger and slower than L1
- But still much faster than memory
- Inclusive vs. exclusive
- In multi-core processors: Repeat idea -> L3







"Computer Architecture: A Quantitative Approach", H. and P., 5th Ed, 2012.

Improvements: Separate Instruction and Data Caches

- Inst and data have different access patterns
- Inst have lower miss rate but cost is higher
- Separate Inst and Data Caches

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorihtm

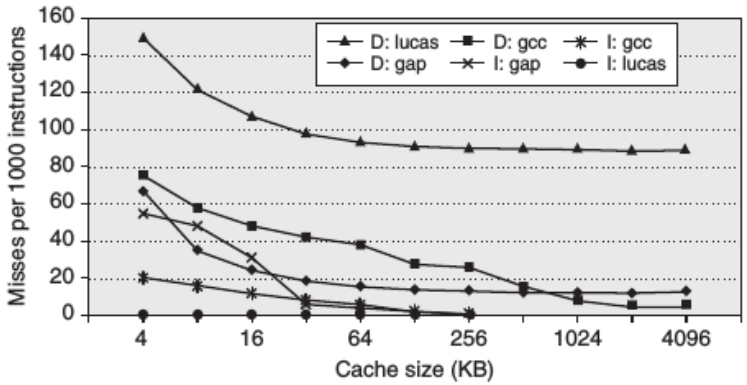


Figure 2.26 Instruction and data misses per 1000 instructions as cache size varies from 4 KB to 4096 KB. Instruction misses for gcc are 30,000 to 40,000 times larger than lucas, and, conversely, data misses for lucas are 2 to 60 times larger than gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite.

"Computer Architecture: A Quantitative Approach", H. and P., 5th Ed, 2012.

Power breakdown (1)

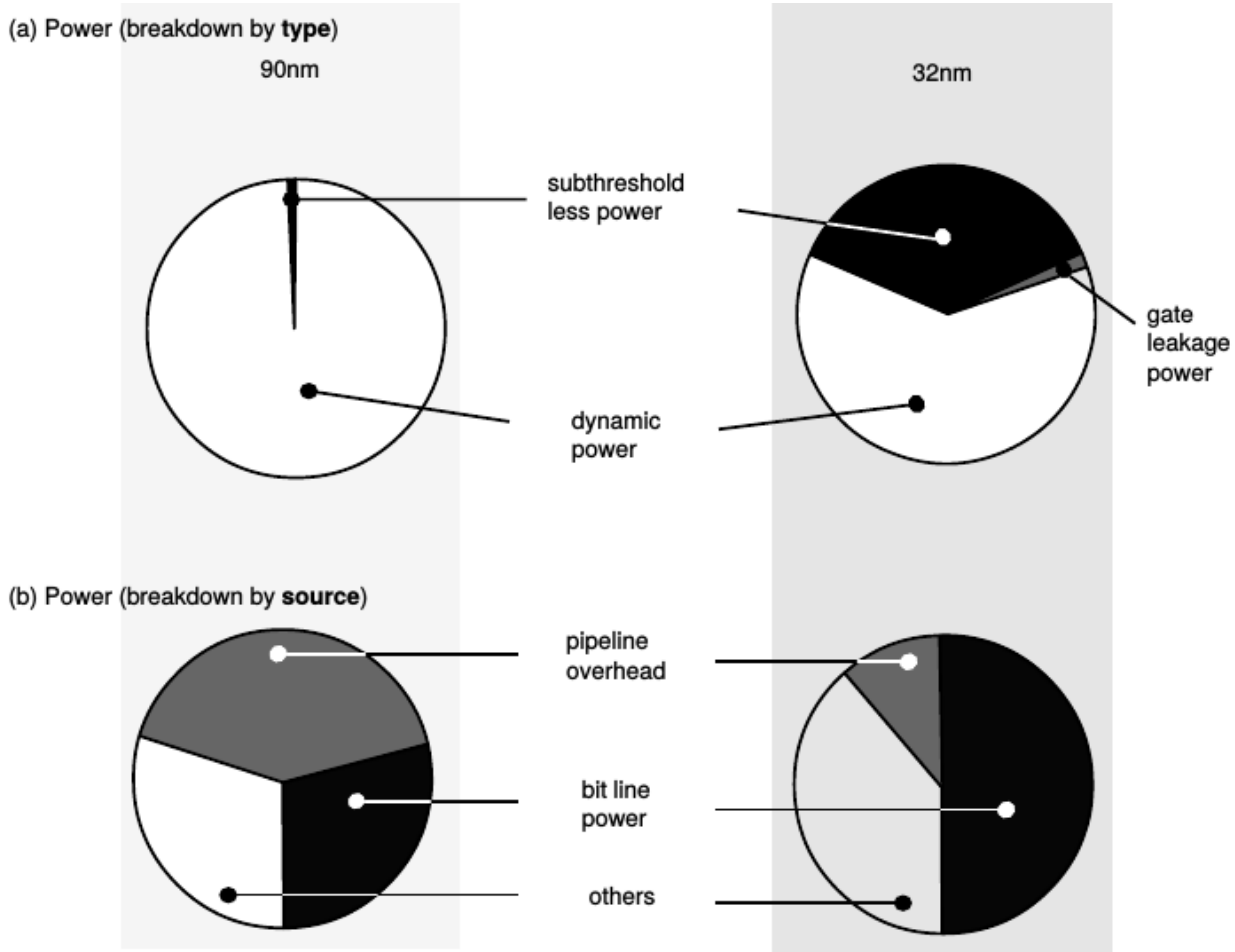


FIGURE 29.6: Overview of cache power. (a) Dynamic and static power dissipation components for a 64 kB-4W cache in 90 nm and 32 nm, (b) major components of power dissipation for a 64 kB-4 way, 90-nm and 32-nm pipelined cache.

"Memory systems: Cache, DRAM, Disk", Bruce Jacob et al., Ed. Morgan Kaufmann, 2008.

Power breakdown (2)

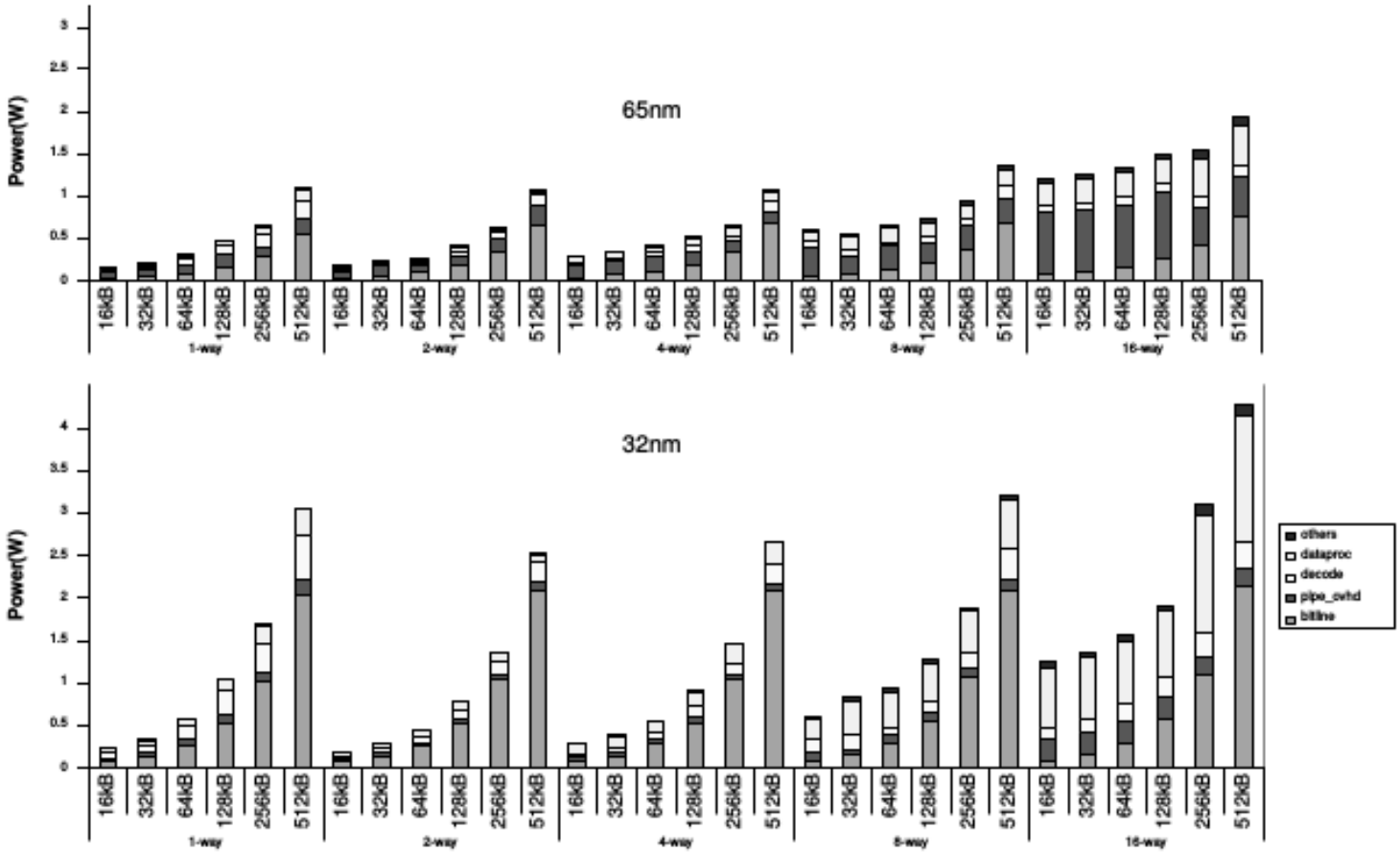


FIGURE 29.13: Detailed breakdown. Power breakdown showing major cache power contributors for the 65- and 32-nm technology nodes for different cache sizes and associativities. Cache operating frequency increases with technology node. (Note that the *y*-axis for both plots uses the same scale.)

"Memory systems: Cache, DRAM, Disk", Bruce Jacob et al., Ed. Morgan Kaufmann, 2008.

Data movement

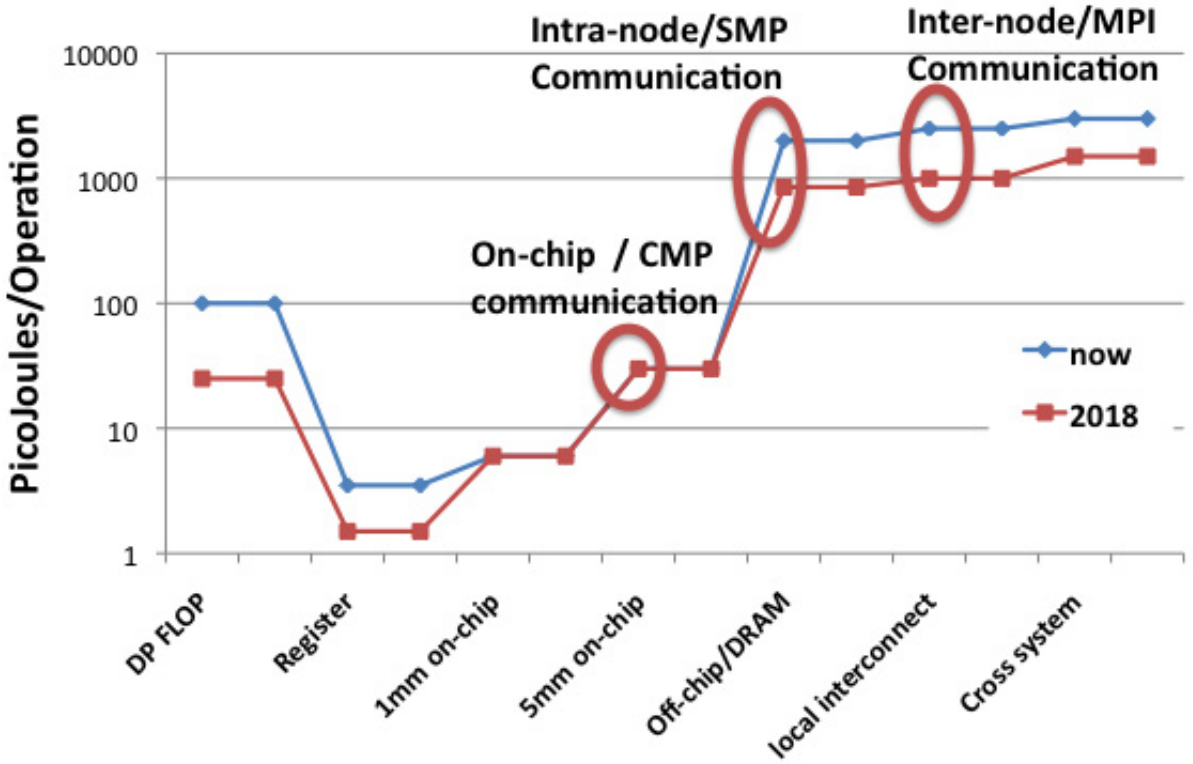


Fig. 2. Energy cost of data movement relative to the cost of a flop for current and 2018 systems (the 2018 estimate is conservative and doesn't account for the development of an advanced memory part). The biggest change in energy cost is moving data off-chip. Therefore, future programming environments must support the ability of algorithms and applications to exploit locality which will, in turn, be necessary to achieve performance and energy efficiency.

"Exascale Computing Technology Challenges", J. Shalf et al., VECPAR 2010.

Speculation

Motivation

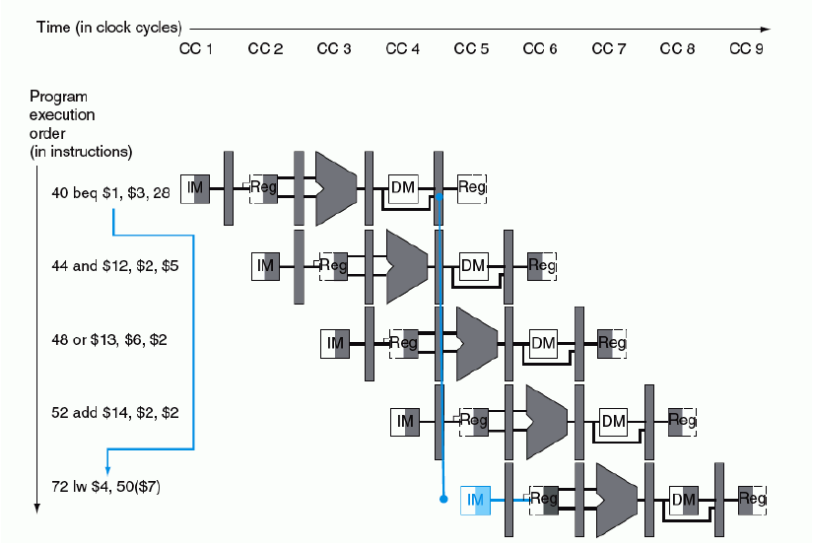
- Waiting to resolve a dependency may take many cycles
- This results in significant pipeline bubbles, which seriously hurt performance
- A solution: speculate on the outcome and continue execution based on the assumption
- If wrong, correct the results
- Significant performance speedups, but if frequently misspeculated, a lot of energy is wasted on useless computation
- Two important cases: branches and load-store aliasing

Load-store aliasing

- Memory accesses to the same address must be serialized
- Accesses to different addresses can proceed in parallel
- The problem is that it may take a number of cycles to calculate the address
- Solution: assume they are independent
- If misspeculation, correct and record the case (StWait, Store Sets)
- Next time, stall until address is calculated

Branch prediction: motivation

- Non-control inst: PC++
- Until control inst completes, next PC is not known
- Conditional branches: Condition must be evaluated to know if taken or not taken
- Indirect branches: Read next PC
- Just stall fetch -> many wasted cycles



"Computer Organization and Design: The Hardware/Software Interface", P. and H., 3th Ed., 2004.

Can we speculate?

- Actual next PC not known until branch completes, but sometimes is predictable
- Many cond. br. are predictable with high confidence

```
sum=0;                  //    MOVI R0, 0;
i=0;                    //    MOVI R1, 0;
do {                    // loop:
  sum+=a[i];            //    LD R2, a(R1);
                        //    ADD R0, R2, R0
  i++;                  //    ADDI R1, R1, 1
} while (i<1000);       //    BNEQ R1, 1000, loop
```

The branch is taken 1000 times
and changes to not taken on last iteration

Branch prediction

- What is predicted?
  - Direction of conditional branches (Taken/Not Taken)
  - Target @
- The prediction is either correct...
  - It is called a Hit
  - Both direction and target prediction must match to hit
- ...or incorrect
  - Known as Miss
  - On a miss: flush pipeline, undo wrong-path changes

Predicting direction

- PC-indexed table of predictors
- But it has no tag (aliasing)
- Simple predictor: remember last direction of branch
- Poor HitRate. Ex: last, first iter. of loop-branch misses
- Better HitRate if pred resists to changes
- 2-bit saturating counter:
  must miss twice to change predicted direction

Improving the predictor: history

- Following code exposes problem of PC-indexed pred

```
for(i=0;i<100;i++)
  if (i%2) a[i]++;
```

- The if-branch alternates Taken and Not Taken -> many misses!
- But follows a regular pattern -> it can be predicted: if prev branch T-> current NT; else current T;

Further improvements

- Hash (combine, xor) PC and hist register bits
- Use several predictors (global, local, hist sizes, hashes) and choose (majority, predictor of predictors)
- Neural-based predictors (perceptron)

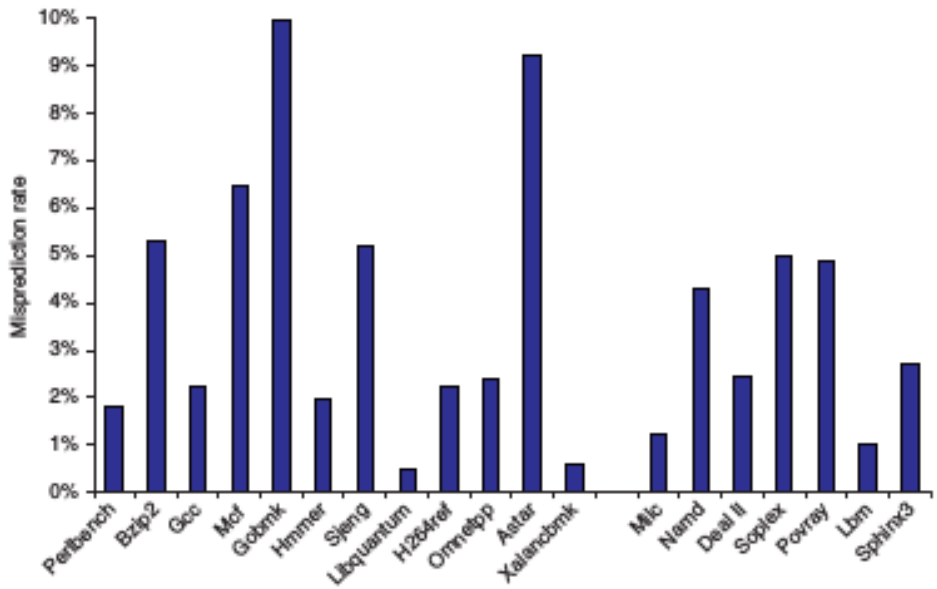
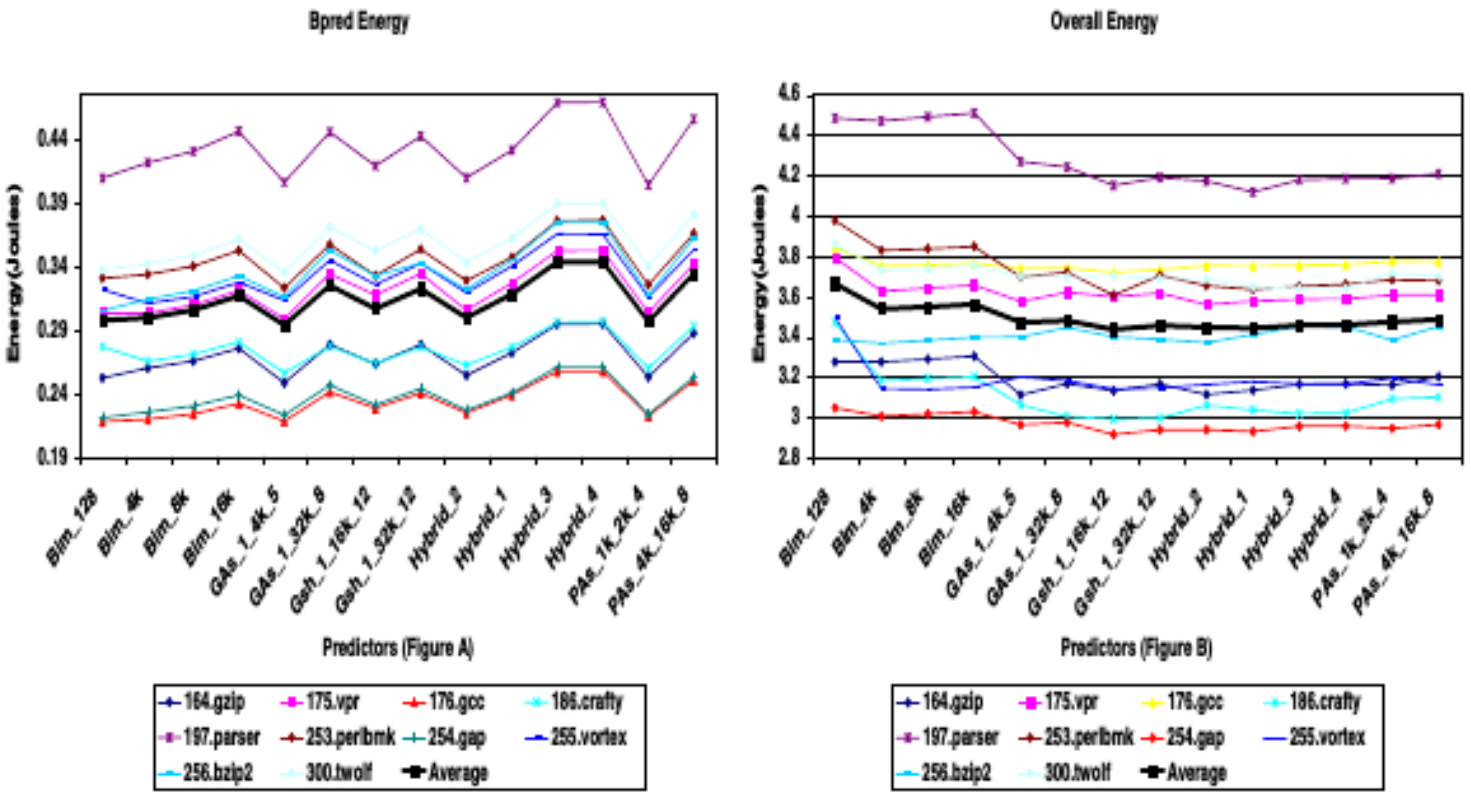


Figure 3.5 The misprediction rate for 19 of the SPECCPU2006 benchmarks versus the number of successfully retired branches is slightly higher on average for the integer benchmarks than for the FP (4% versus 3%). More importantly, it is much higher for a few benchmarks.

"Computer Architecture: A Quantitative Approach", H. and P., 5th Ed, 2012.

Overhead... but overall gains



"Power Issues Related to Branch Prediction", D. Parikh et al., HPCA 2002

Misspeculated instructions

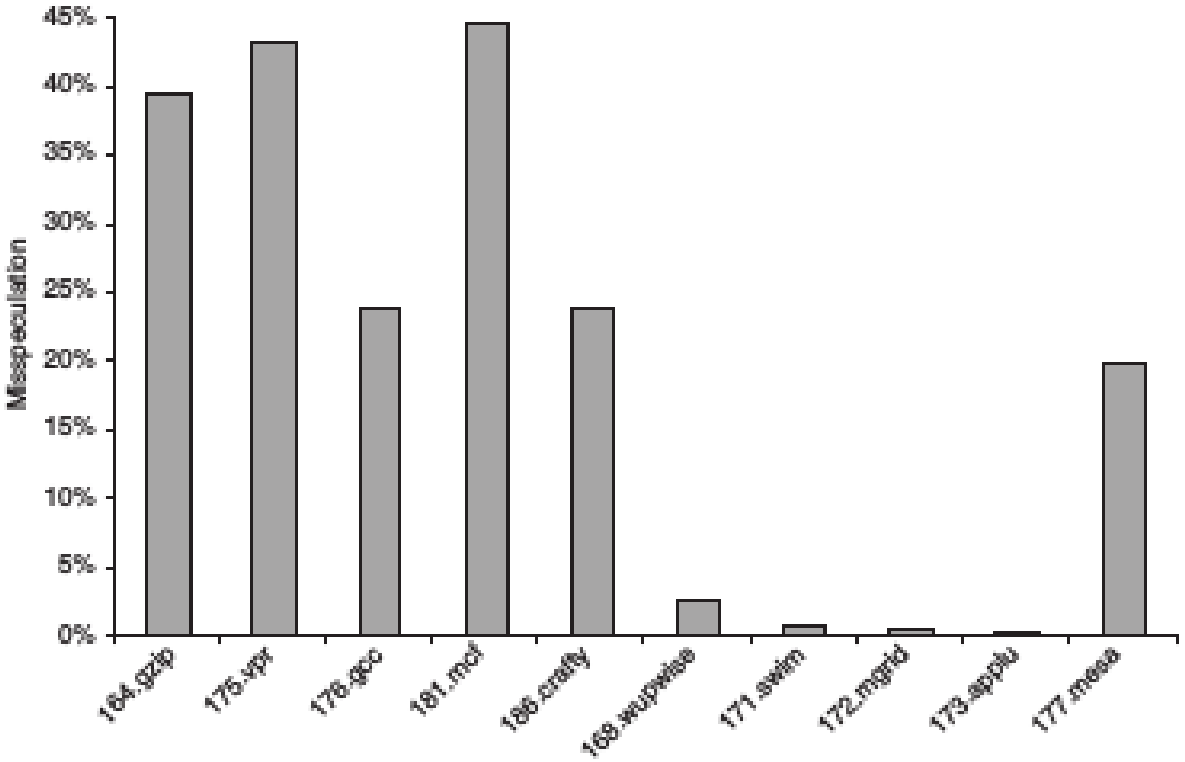


Figure 3.25 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

"Computer Architecture: A Quantitative Approach", H. and P., 5th Ed, 2012.

Out-of-Order processors

Stalled independent instructions

- In this code, SUB depends on LD, but ADD is independent

```
LD x(R1), R2
SUB R2, R3, R4
ADD R5, R6, R7
```

- If LD misses, SUB and ADD stall many cycles
- An independent inst (ADD) can advance its execution? Out-of-order (OoO) vs. In-Order (IO) execution

Classifying data dependences

- Two dependent inst can have a data (real) dependence that create a RAW (Read After Write) hazard

```
MUL R1, R2, R3
ADD R3, R4, R5
```

- An antidependence -> WAR (Write After Read) hazard

```
MUL R3, R4, R5
ADD R1, R2, R3
```

- An out dependence -> WAW (Write After Write) hazard

```
MUL R1, R2, R3
ADD R5, R4, R3
```

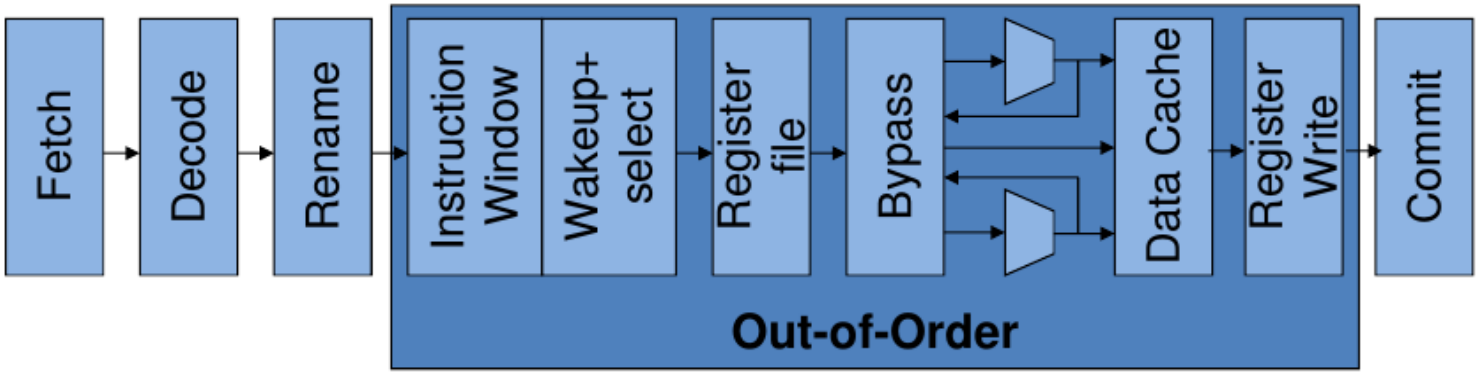
- Anti- and out: false or name dependences

Removing dependences?

- Only real dependences must be honored...
- ...as long as write order is correct (WAW/WAR hazards)
- ...and speculative inst are “undoable”
- Out-of-order execution, in-order completion
- State is not actually updated until all older inst complete
- Beware: the relative order of memory accesses

New elements

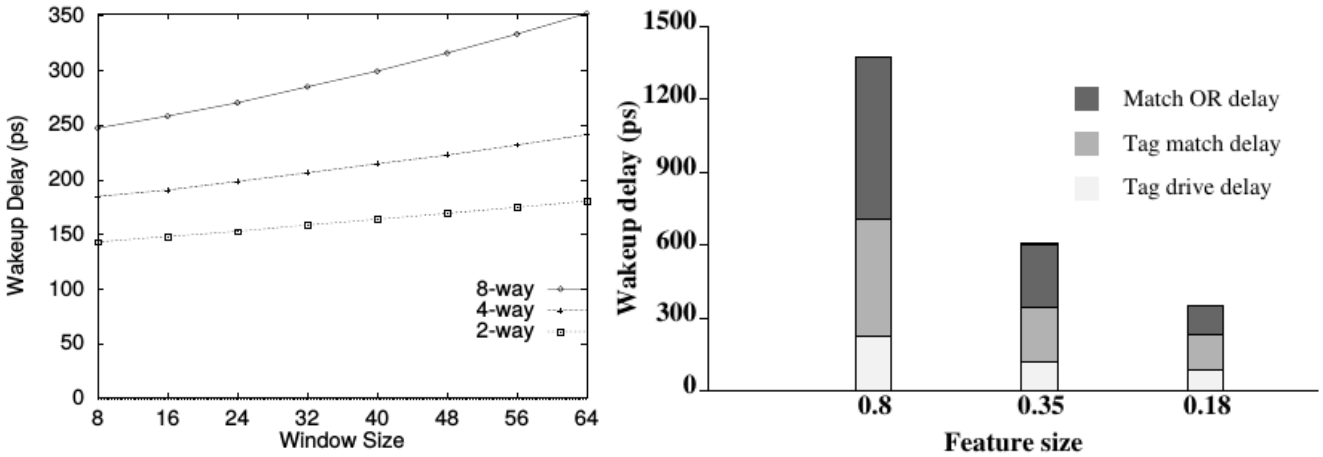
- Issue stage and queue
- Commit stage and ROB
- Register renaming
- Load and store queues



"Runtime Aware Architectures", Mateo Valero, HiPEAC CSW 2014, Barcelona.

Issue stage and queue

- After inst is decoded, place it in issue queue (IQ)
- Select logic picks inst in IQ and sends to FU
  - Inst must have all operands ready
  - And an available FU
  - Up to issue-width inst
- Wake-up logic notify result to dependent inst in IQ
  - IQ needs comparators to check if dependent
  - We must know which operands are ready at insert time



"Complexity-Effective Superscalar Processors", S. Palacharla et al., ISCA 1997.

Energy efficiency?

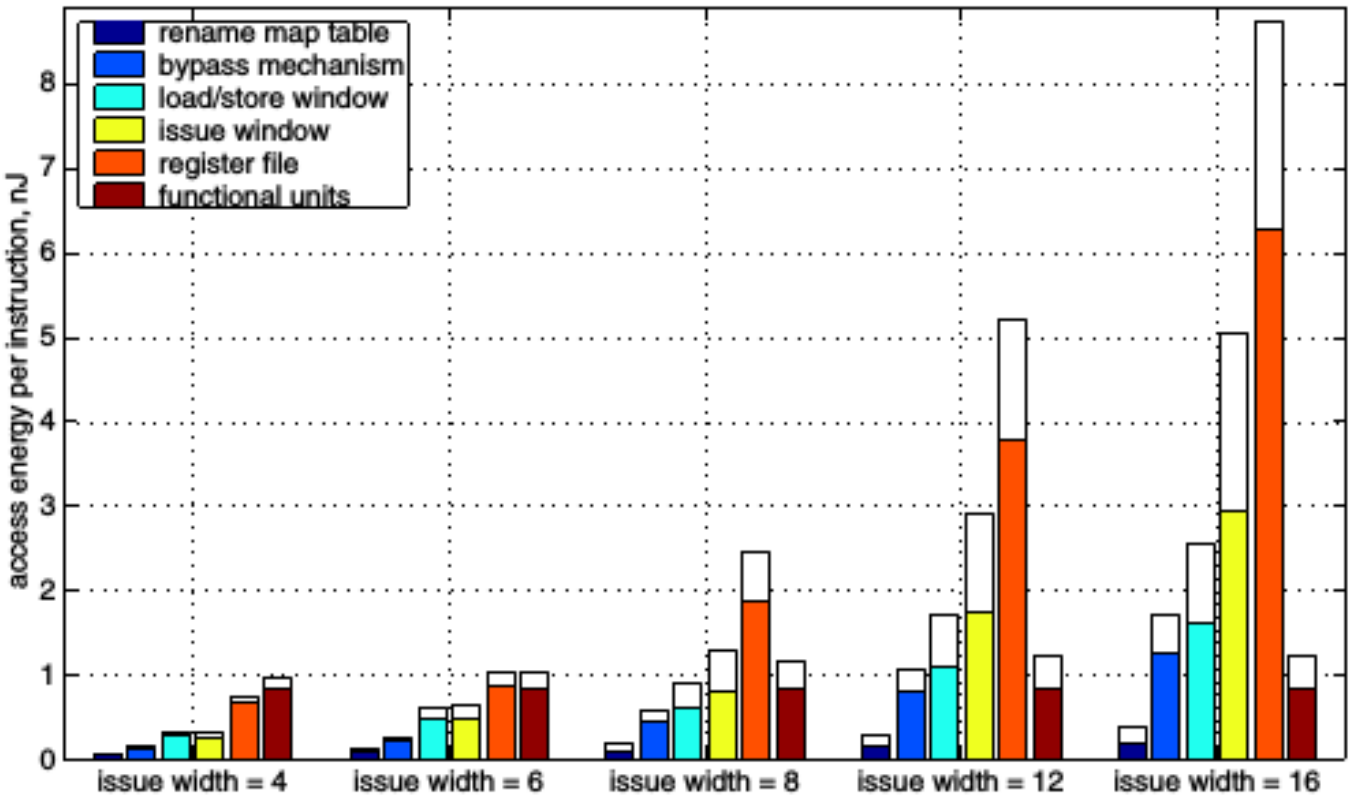
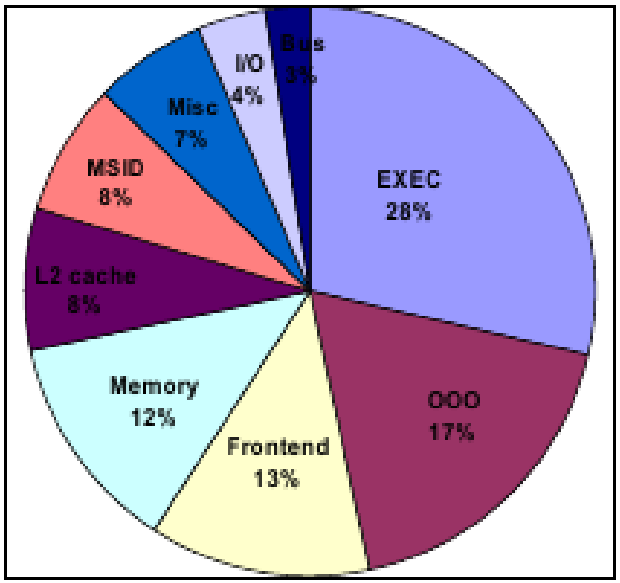


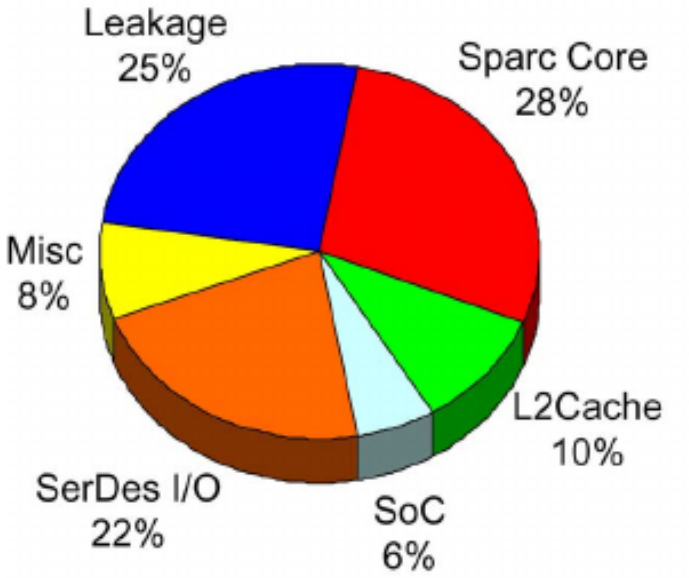
Figure 2: Average energies dissipated per committed instruction (measured on SPEC95); 0.35μ feature size, $V_{dd} = 3.3V$.

"Optimization of High-Performance Superscalar Architectures for Energy Efficiency", V. Zyuban and P. Kogge, ISLPED, 2000.

Power breakdown



Intel Penryn



Sun UltraSPARC T3

"Penryn: 45-nm Next Generation Intel®CoreTM 2 Processor", V. George et al., IEEE ASSCC, 2007.
"A 40 nm 16-Core 128-Thread SPARC SoC Processor", J.L. Shin et al., IEEE JSSCC, Jan. 2011.

Current trends and the ParaDIME project

The ParaDIME project

- **Para**llel **D**istributed **I**nfrastructure for **M**inimization of **E**nergy
- Combine techniques, cooperation and sharing of information between layers (application, runtime, programming model, hardware, device) for energy-efficient data-centers
  - e.g. annotations for reduced precision and high performance vs low power sections

The ParaDIME Consortium

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TECHNISCHE UNIVERSITÄT DRESDEN

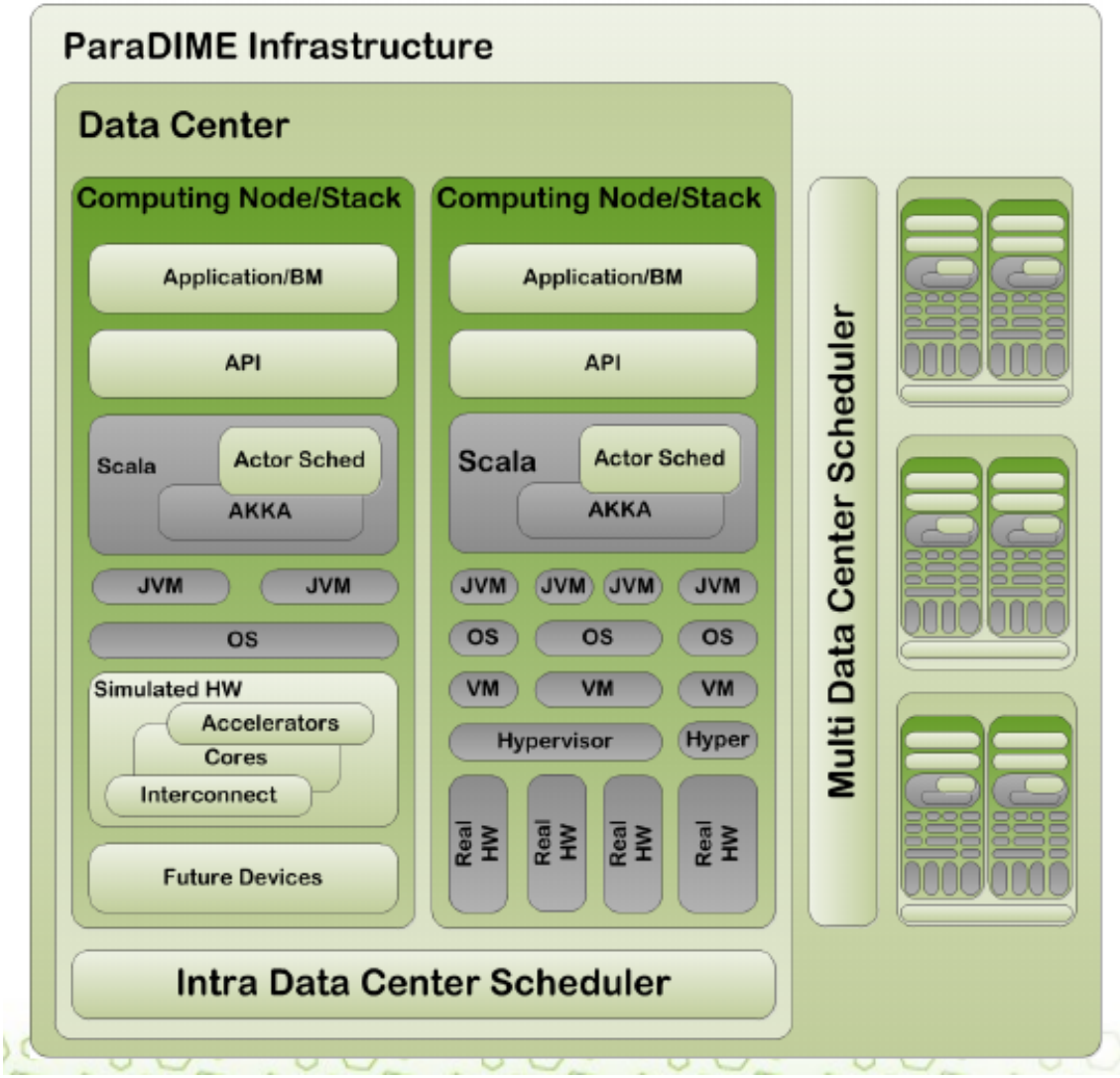
Université de Neuchâtel

AOTERRA
GREENLOCALEFFICIENT

imec

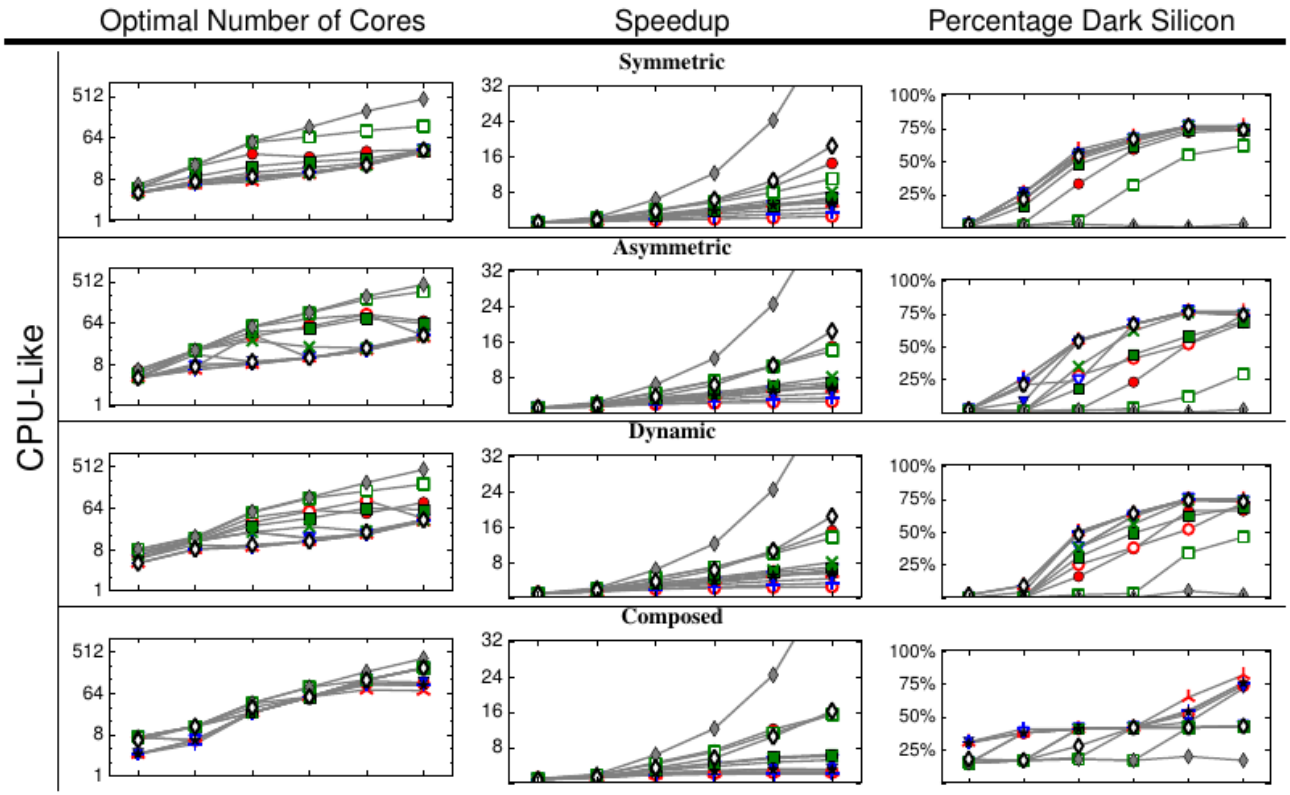
AOTerra is now Cloud&Heat (www.cloudandheat.com)

The ParaDIME stack



Multi/many-cores

- Trend is to increase number of cores (Xeon Phi)
- Programmability is a great concern/open issue
- Dark silicon also consider a problem



"Dark Silicon and the End of Multicore Scaling", H. Esmaeilzadeh et al., ISCA 2011.

Coherence and consistency

- Consistency model: write order
- Cache coherence problem: one proc updates data but another reads old cached value

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

- Coherent system if:
  - If PA reads X after PA writes X, PA reads new value
  - If PA reads X after PB writes X, PA reads new value
  - Writes serialization (seen in same order everywhere)
- Enforce coherence granting exclusive access and invalidating cache lines
- Cache coherence protocols

"Computer Architecture: A Quantitative Approach", H. and P., 5th Ed, 2012.

Cache protocols

- Possible states: Modified, Shared, Invalid (MSI) additionally: Owned, Exclusive (MESI, MOESI)
  - Invalidate (if other proc wants to write)
  - Send cached copy (if it was written)
- Scalability to many cores? Overhead of maintaining coherence?

Message passing

- In ParaDIME we are studying how to build a many-core without cache coherence
  - Precedent: Intel SCCC: Single-Chip Cloud Computer
- Leverage actor-based programming model
- We will include support for cache-to-cache delivery, avoiding unnecessary copies of data

Heterogeneous computing

- Heterogeneous systems can be much more power-efficient (dark silicon, Green500)
- GPGPUs, Xeon Phi, FPGAs, big.LITTLE...
- In ParaDIME, message passing co-processor and vector accelerators

Reduced precision

- Switch off as many elements of the processor as possible, whenever they are not used/necessary (dark silicon again)
- Unused cores, ALUs, etc.
- Shrink cache, ROB, branch predictor at run-time, trading performance and power
- Reduced precision: operate with much smaller data types. Save energy in FU, data movement, etc. at the cost of accuracy
- The programmer indicates in which parts of the code this can be applied

Further reading

- "Computer Architecture: A Quantitative Approach", Hennessy and Patterson, 5th Ed., Morgan Kaufmann, 2012.
- "Computer Organization and Design: The Hardware/Software Interface", Patterson and Hennessy, 5th Ed., Morgan Kaufmann, 2013.
- "Computer Architecture Techniques for Power-Efficiency", Kaxiras and Martonosi, Morgan Claypool, 2008.
- "Dark Silicon and the End of Multicore Scaling", H. Esmaeilzadeh et al., ISCA 2011.
- "Complexity-Effective Superscalar Processors", S. Palacharla et al., ISCA 1997.
- Selected ParaDIME papers:
  - "System-Level Power and Energy Estimation Methodology for Open Multimedia Applications Platform", S. K. Rethinagiri et al., ISVLSI, 2014.
  - "Memory Controller for Vector Processor", T. Hussain, ASAP, 2014.
  - "Leveraging Transactional Memory for Energy-efficient Computing below Safe Operation Margins", A. Cristal et al., TRASACT, 2013.

Conclusions

- The power wall has led to reevaluate ALL C.A.
  - New trade-offs in issue width, out-of-order vs. in-order, etc.
  - Performance vs power consumption, latency vs. throughput, locality
  - New design choices, e.g. many simple cores vs. few complex cores
  - Old ideas can be useful now (vector architectures, dataflow execution, ...)
  - A lot of other ideas
- Multi- (many) core are here to stay, but there is a problem with their programmability
  - Programmability wall
  - A lot of effort: transactional memory, actors, tasks...
- Dark silicon threat and exascale goal puts even more pressure to improve power-efficiency
- While importance of mobile market makes energy minimization the first concern

THANKS!