Trends in System Cost and Performance Balances and Implications for the Future of HPC

John D. McCalpin, PhD

mccalpin@tacc.utexas.edu

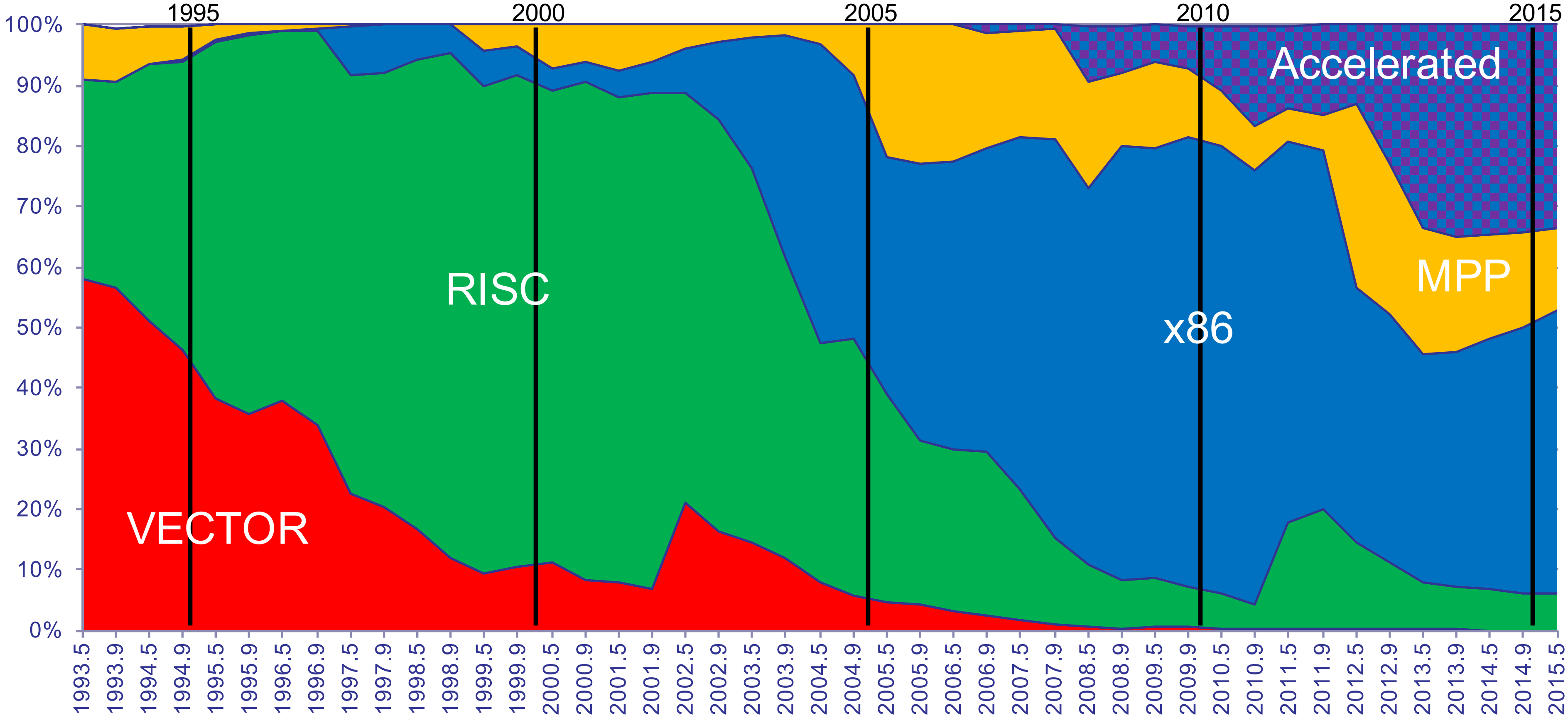
Outline

1. Review: Changes in HPC Systems
2. Technology Trends & System Balances
3. Fundamental Flaws in Modern Architectures
4. An Alternative Architectural Approach

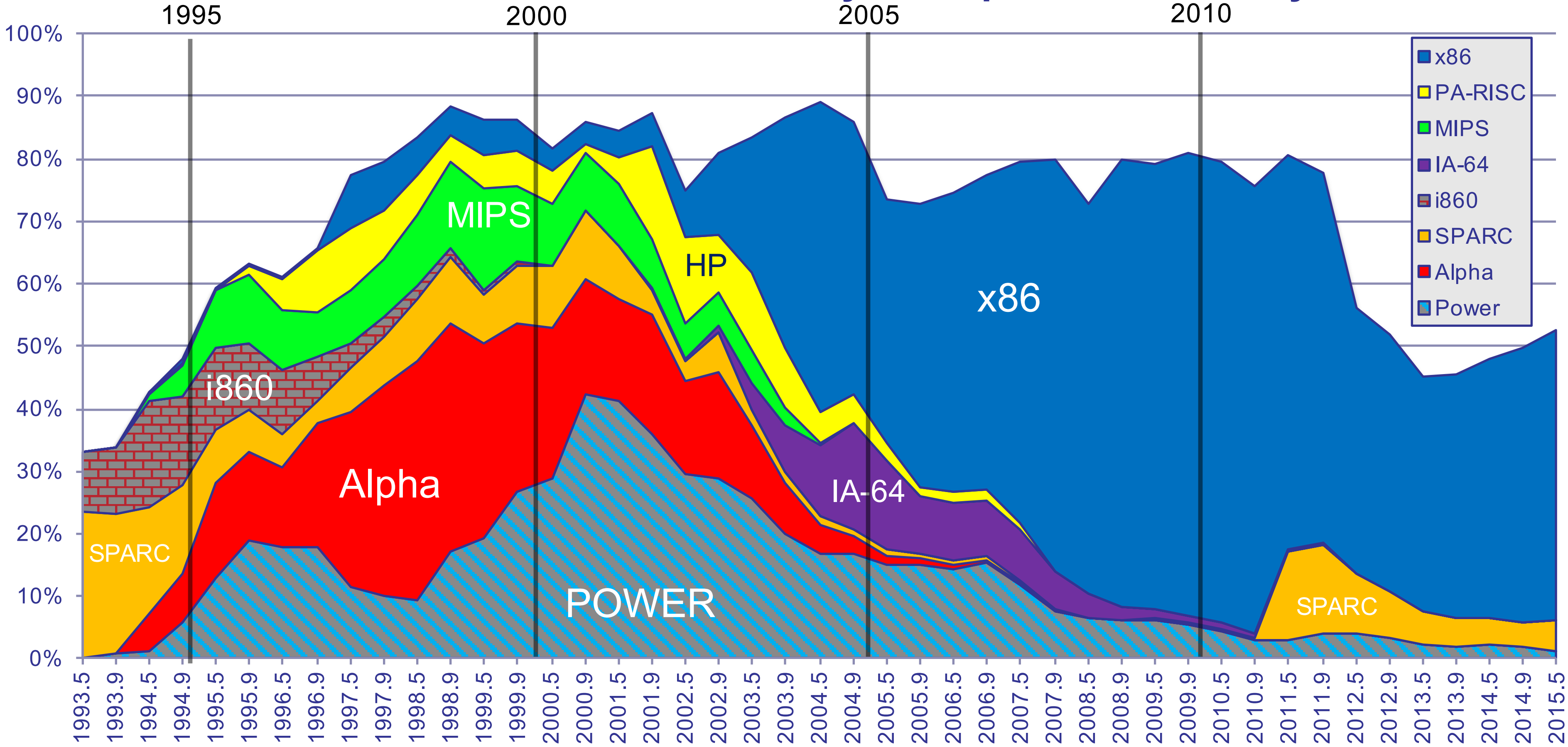
Part 1

CHANGES IN HPC SYSTEMS

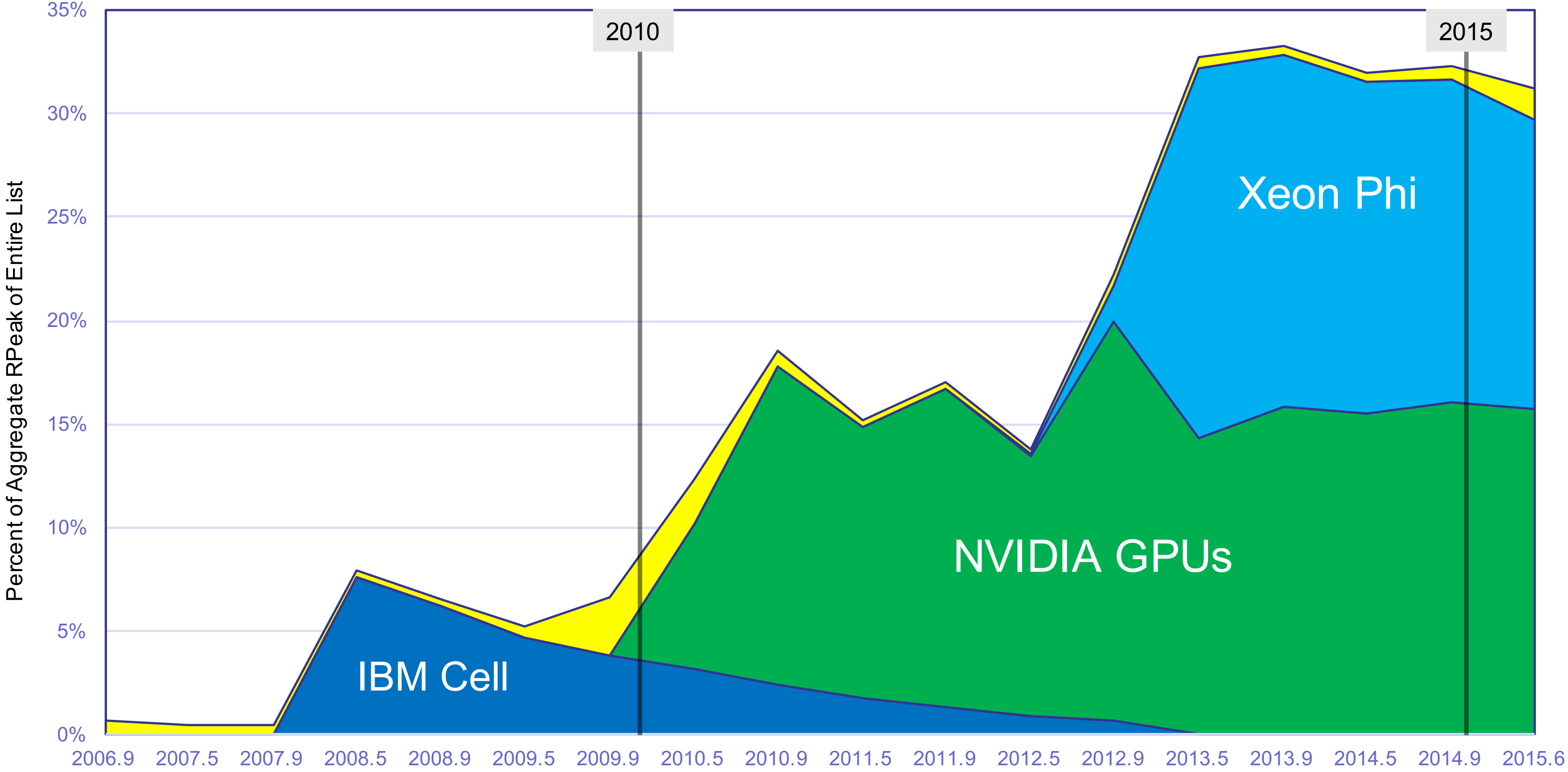
TOP500 Rmax Contributions by System Architecture



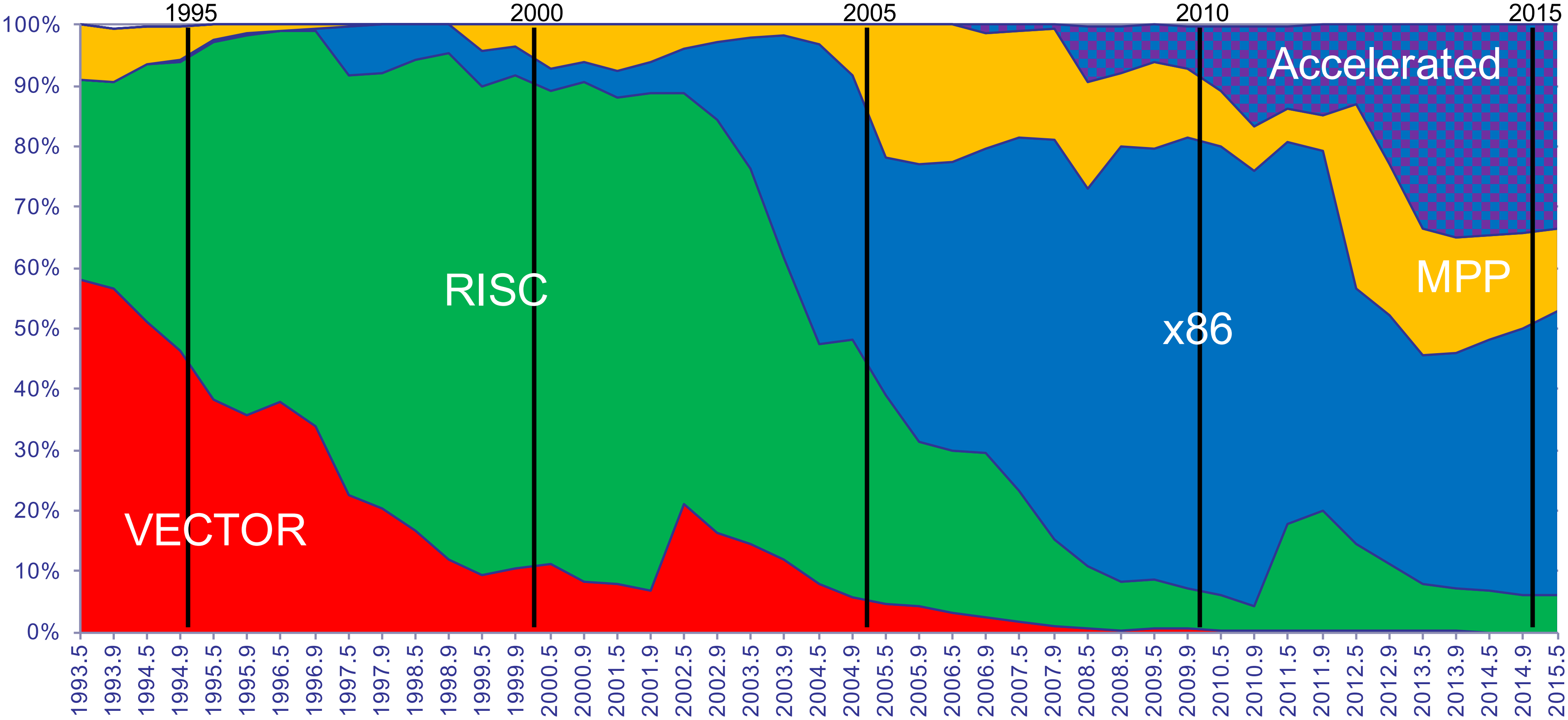
TOP500 Rmax Contributions by Microprocessor Family



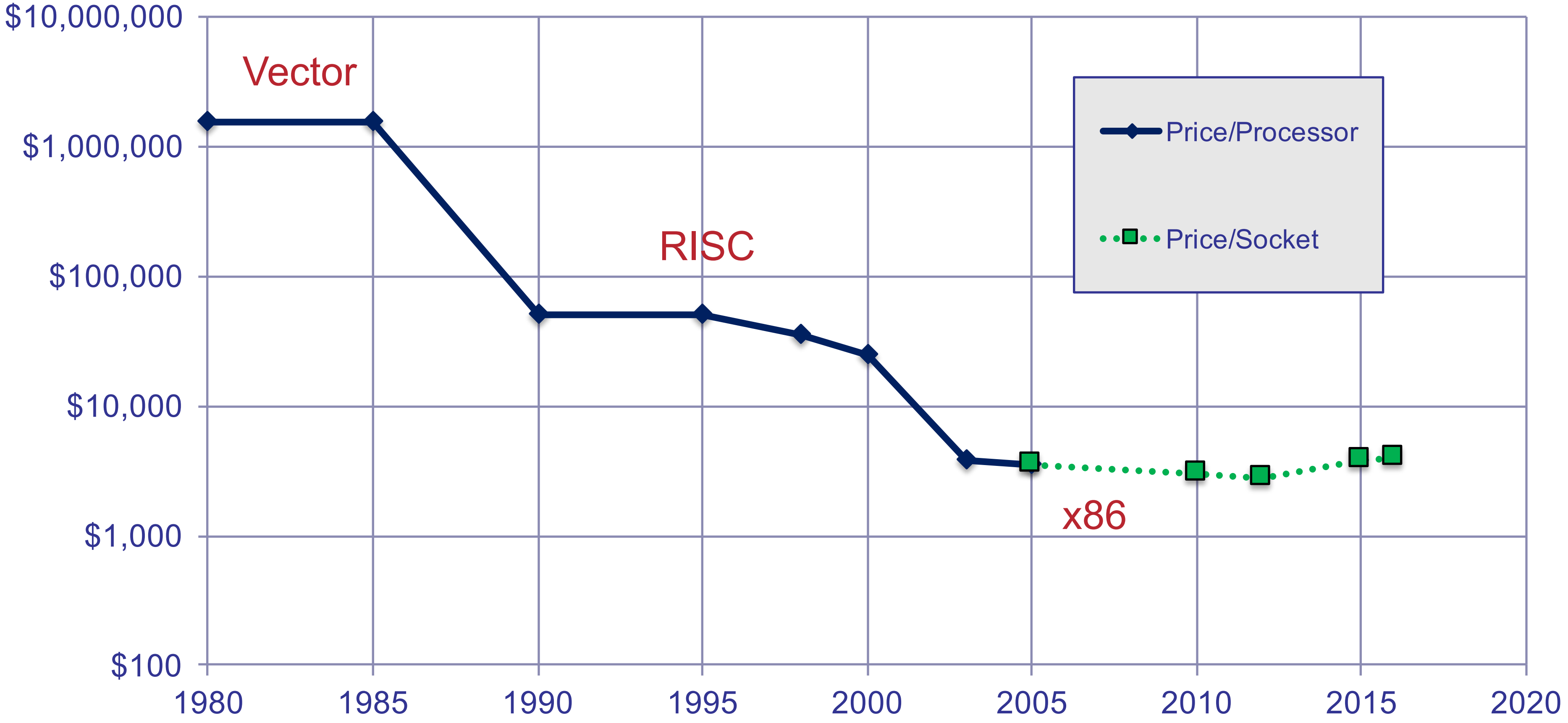
Contributions of Various Accelerator Families to TOP500 RPeak



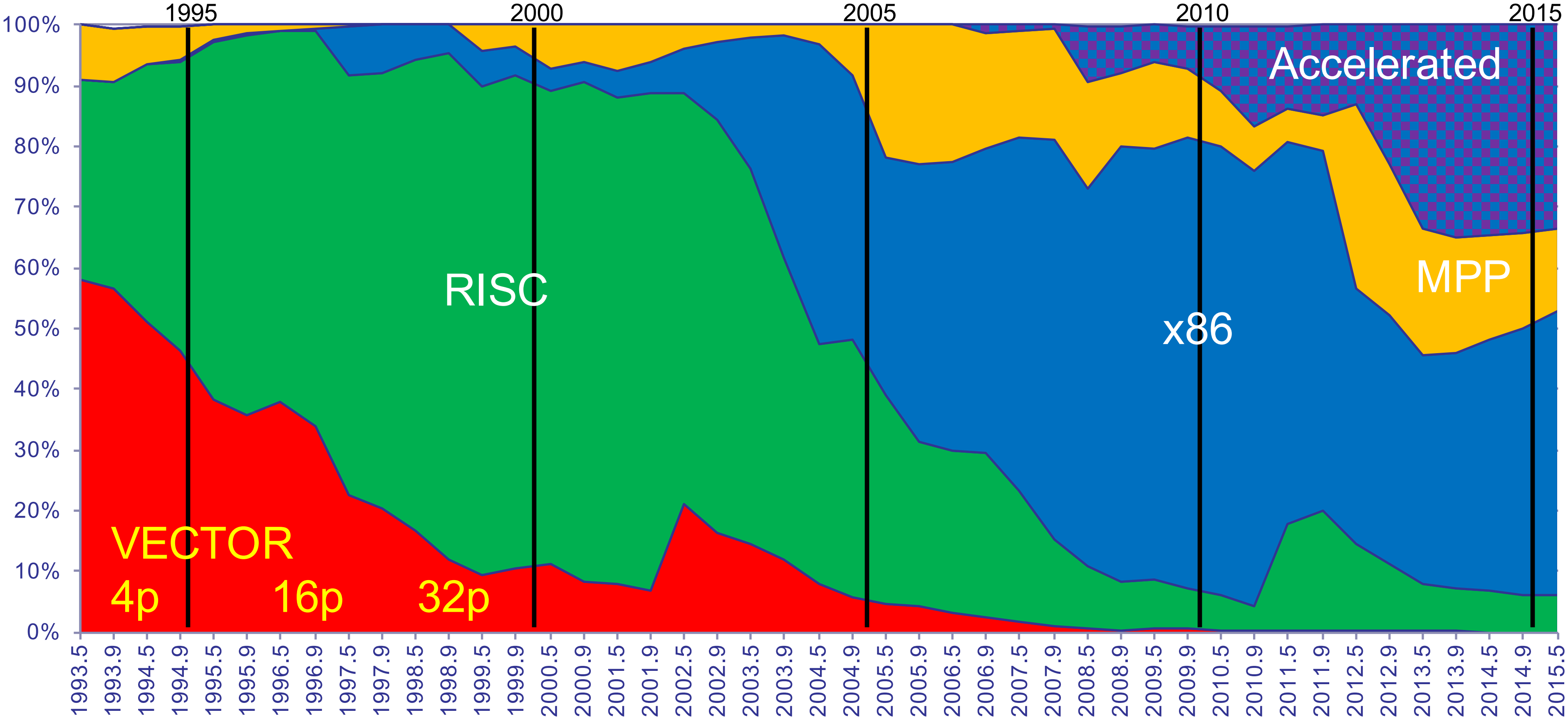
TOP500 Rmax Contributions by System Architecture



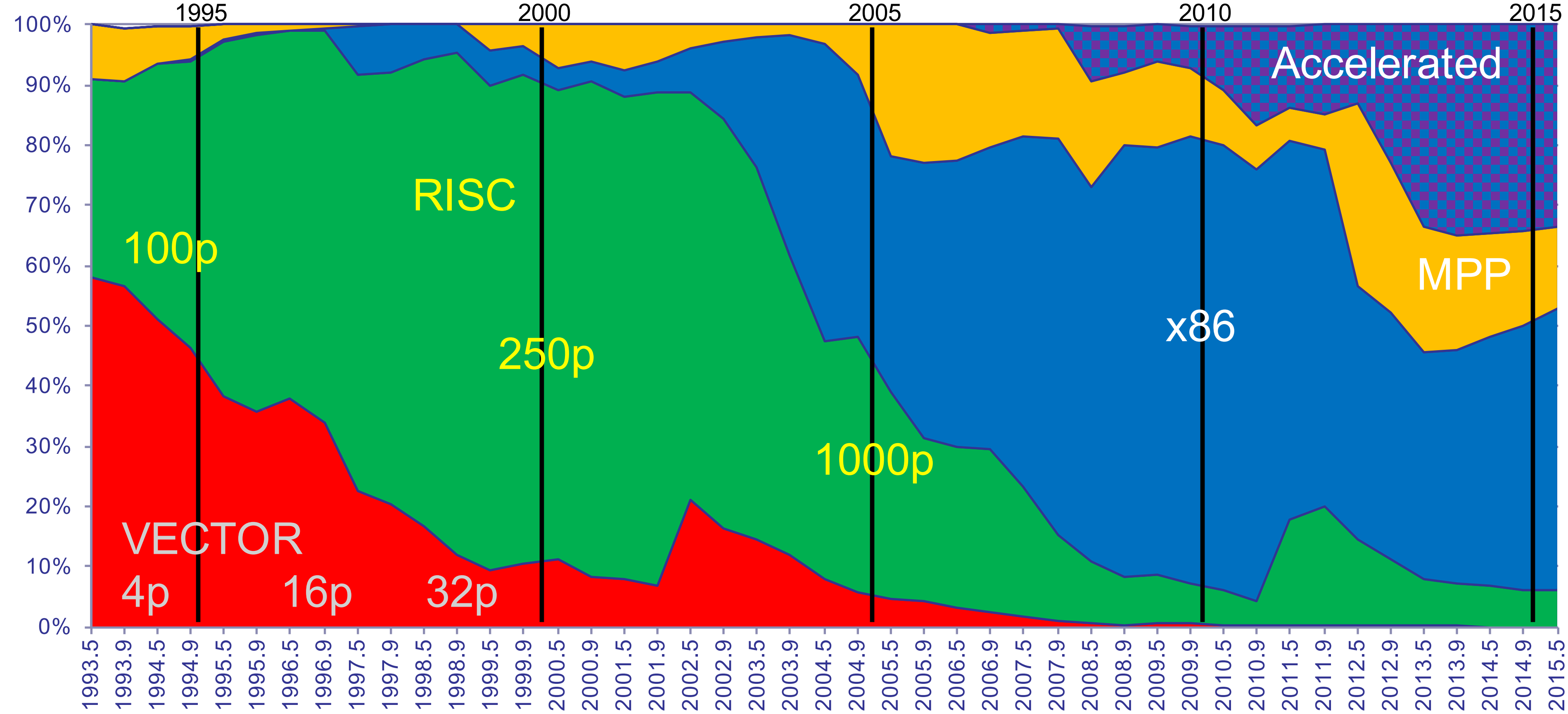
System Price per "Processor"



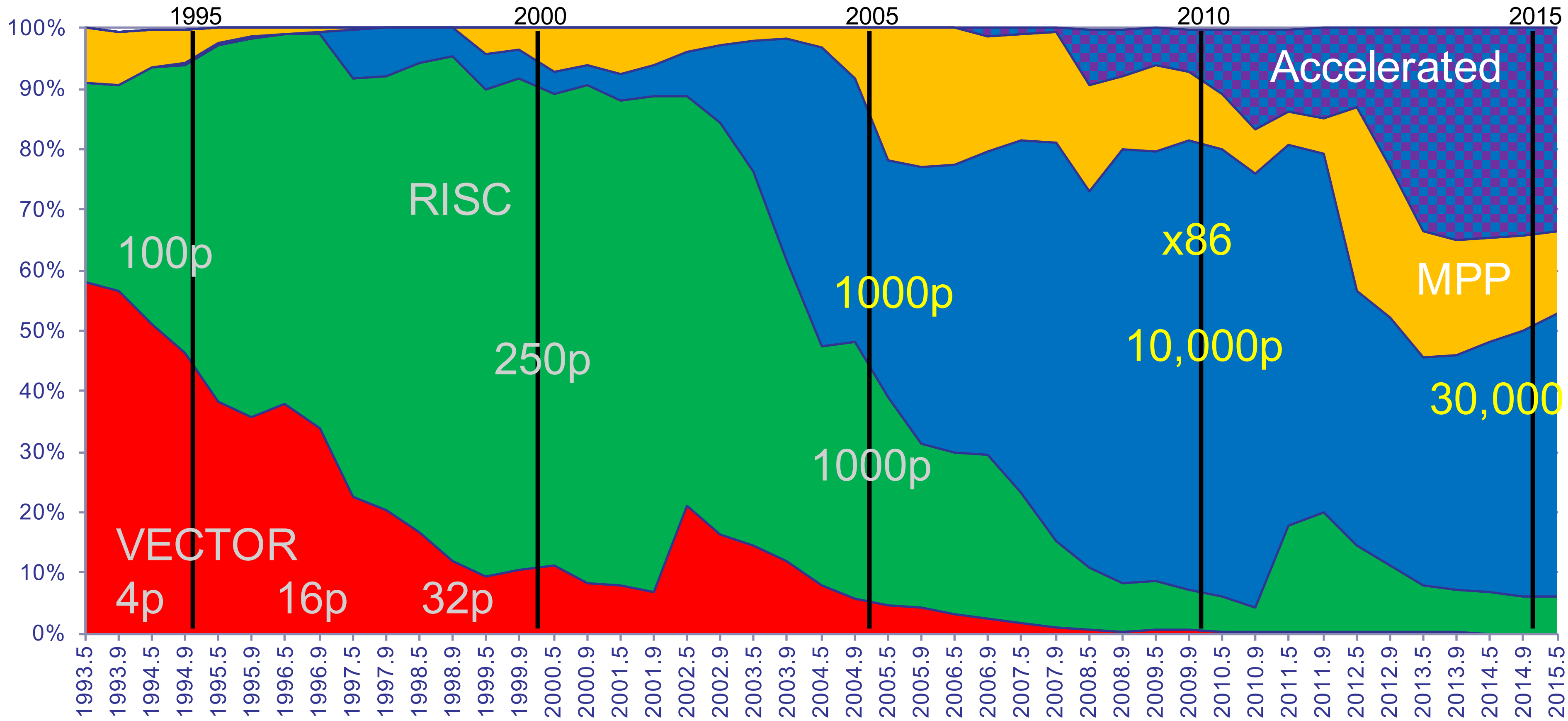
TOP500 Rmax Contributions by System Architecture



TOP500 Rmax Contributions by System Architecture

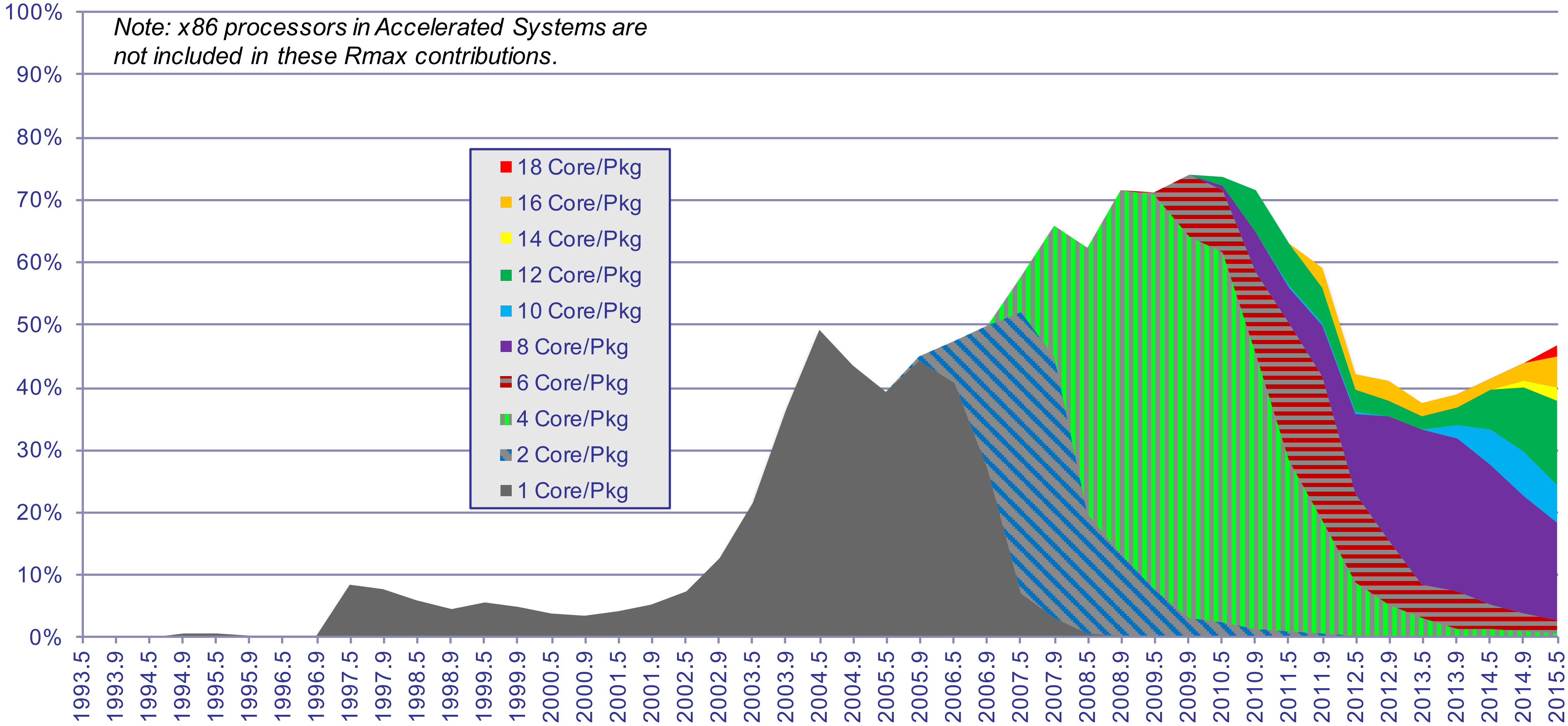


TOP500 Rmax Contributions by System Architecture



TACC

TOP500 Rmax Contributions by Cores/Package



Note: x86 processors in Accelerated Systems are not included in these Rmax contributions.

What about accelerators?

- Accelerators can provide better performance per price and performance per watt, but they do this by increasing the required parallelism – more functional units at lower frequency.
- What about a different approach?

A homogeneous system cannot be “optimal” for a heterogeneous workload!

- “Optimal” here can refer to performance, power, overall cost.

Heterogeneous Systems

- More sites are building “clusters of clusters”, e.g.:
  - Sub-cluster 1: standard 2-socket nodes with small memory
  - Sub-cluster 2: standard 2-socket nodes with large memory
  - Sub-cluster 3: standard 4-socket nodes with very large memory
  - Sub-cluster 4: 1-socket or 2-socket nodes with accelerators, etc…
- This is consistent with the observation that the shift to accelerators has stalled at ~30% of Rpeak, split between many-core and GPU.
  - TACC runs at least 10 clusters, about 1/3 of these have some accelerators (GPU or Xeon Phi or both)

Part 2

TECHNOLOGY TRENDS & SYSTEM BALANCES

Technology Trends

- Performance is many-dimensional, and all the dimensions seem to be changing at different rates!
- CPU:
  - Frequency: -7%/year
  - FP/Hz: +30%/year 2x/2.7 years
  - Cores/package: +24%/year 2x/3.3 years
- DRAM
  - Transfer rate: +15%/year 2x/5 years
  - Width: +7%/year 2x/10 years
  - Latency: Flat to slightly increasing
- Interconnect:
  - Transfer rate: +20%/year 2x/4 years
  - Width: Flat

Technology Trends

- Performance is many-dimensional, and all the dimensions seem to be changing at different rates!
- CPU:
  - Frequency: -7%/year
  - FP/Hz: +30%/year 2x/2.7 years
  - Cores/package: +24%/year 2x/3.3 years
- DRAM
  - Transfer rate: +15%/year 2x/5 years
  - Width: +7%/year 2x/10 years
  - Latency: Flat to slightly increasing
- Interconnect:
  - Transfer rate: +20%/year 2x/4 years
  - Width: Flat

Technology Trends

- Performance is many-dimensional, and all the dimensions seem to be changing at different rates!
- CPU:
  - Frequency: -7%/year
  - FP/Hz: +30%/year 2x/2.7 years
  - Cores/package: +24%/year 2x/3.3 years
- DRAM
  - Transfer rate: +15%/year 2x/5 years
  - Width: +7%/year 2x/10 years
  - Latency: Flat to slightly increasing
- Interconnect:
  - Transfer rate: +20%/year 2x/4 years
  - Width: Flat

Technology Trends

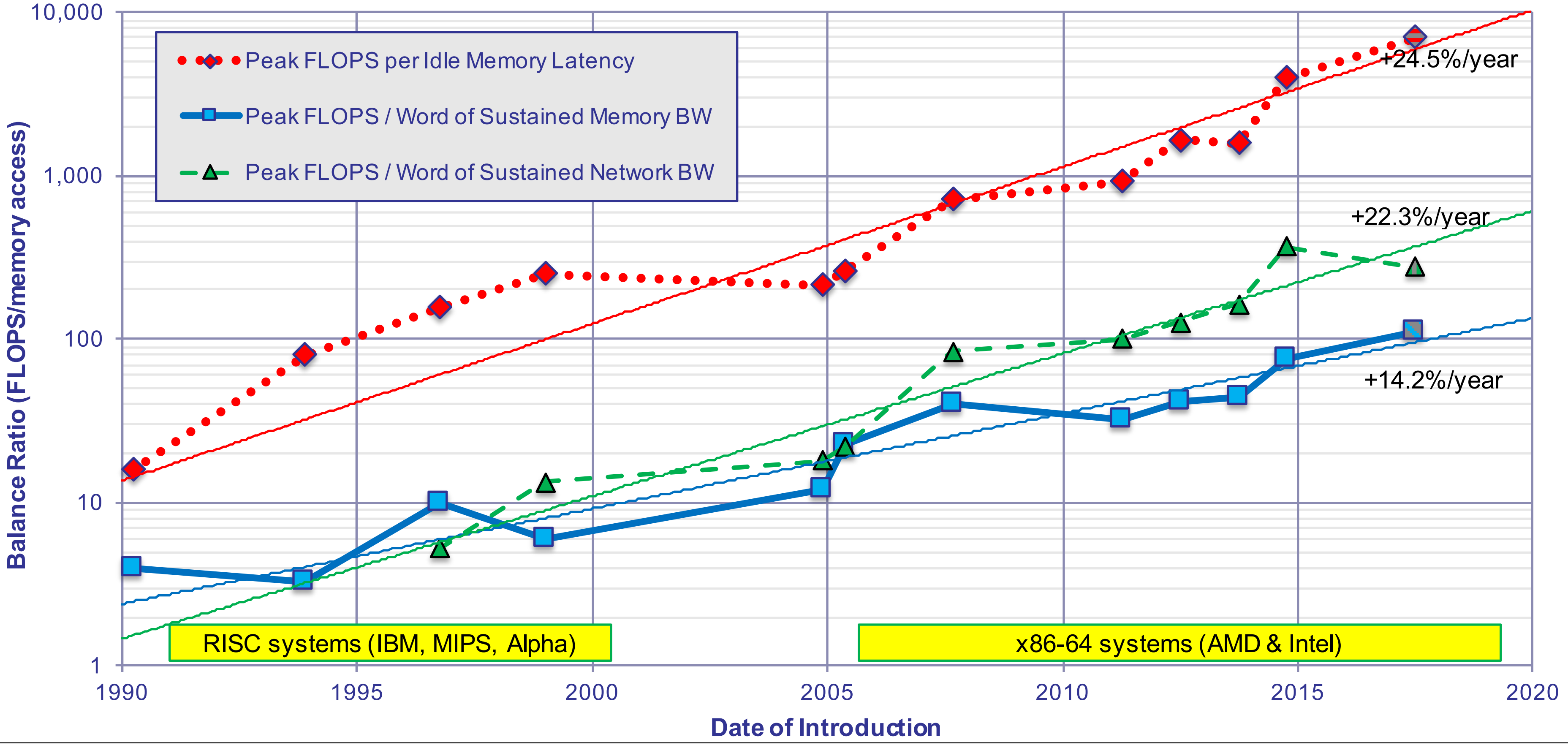
- Performance is many-dimensional, and all the dimensions seem to be changing at different rates!
- CPU:
  - Frequency: -7%/year
  - FP/Hz: +30%/year 2x/2.7 years
  - Cores/package: +24%/year 2x/3.3 years
- DRAM
  - Transfer rate: +15%/year 2x/5 years
  - Width: +7%/year 2x/10 years
  - Latency: Flat to slightly increasing
- Interconnect:
  - Transfer rate: +20%/year 2x/4 years
  - Width: Flat

FLOPS vs BW “Balance” Ratios

- Net CPU trends: **1.5x/year** to **1.6x/year** in Peak FLOPs
- Net DRAM trends: **1.23x/year** in sustained BW
- Net Interconnect trends: **1.2x/year** in sustained BW

- This suggests that processors should be increasingly imbalanced with respect to data motion….

Bandwidth is getting more costly, Latency is much worse



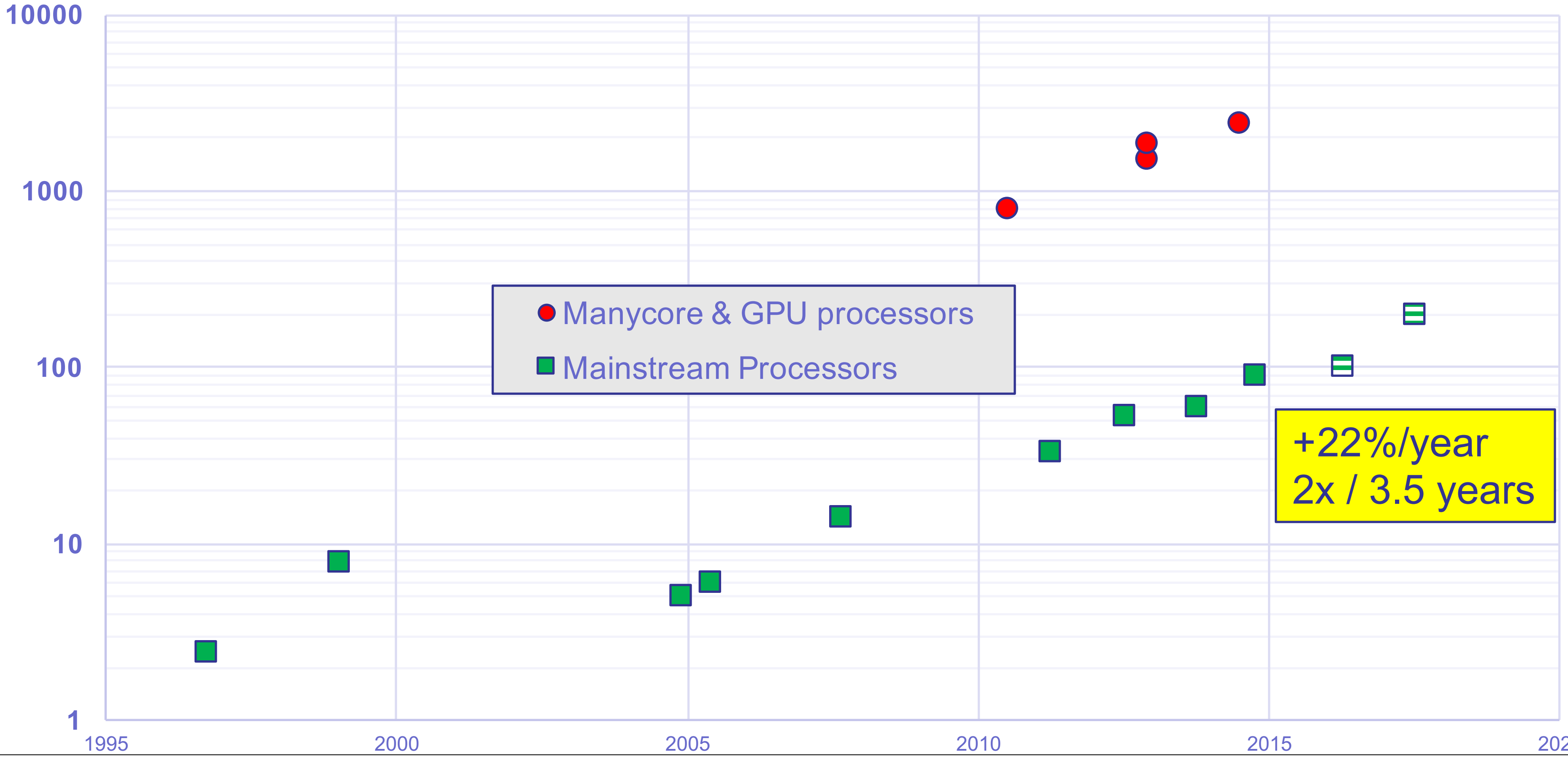
Latency, Bandwidth, and Concurrency

- “Little’s Law” from queuing theory describes the relationship between latency (or occupancy), bandwidth, and concurrency.

Latency * Bandwidth = Concurrency

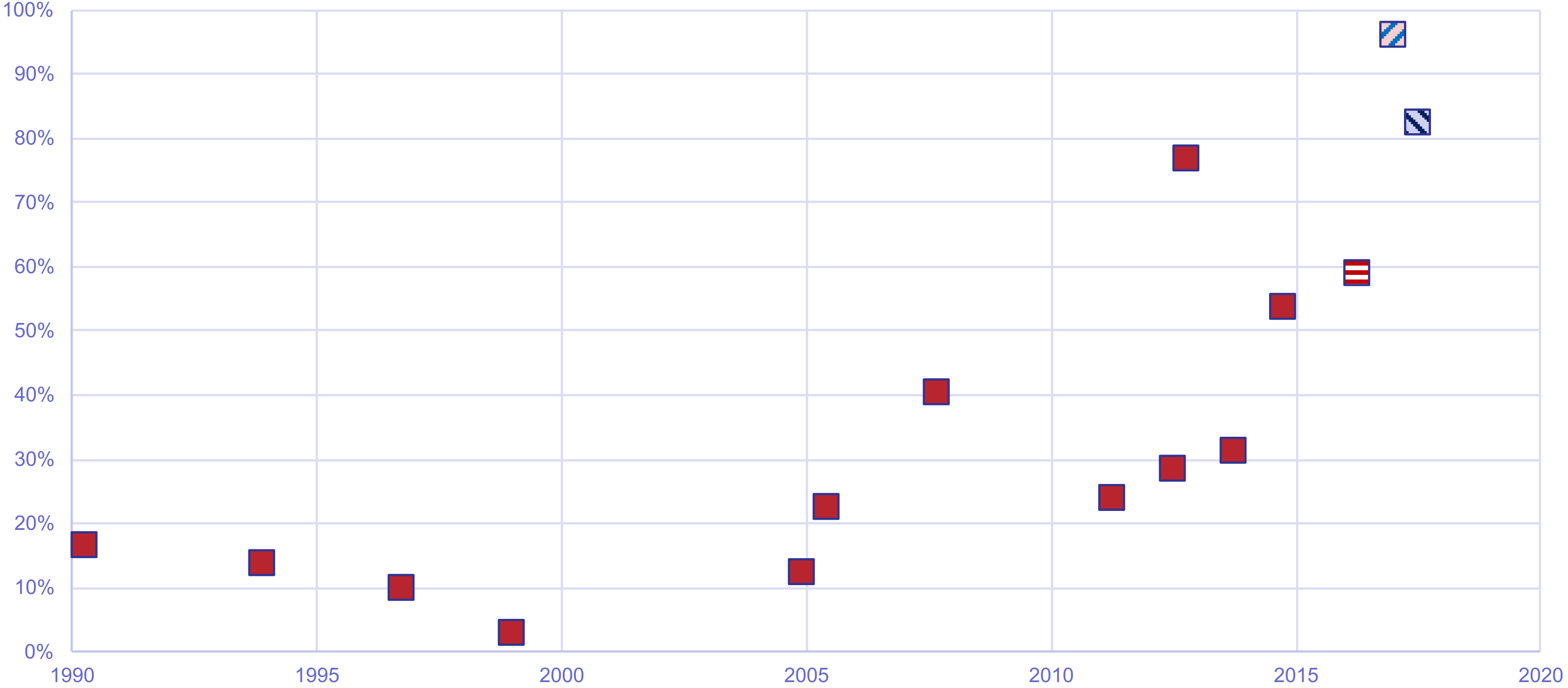
- Flat Latency * increasing Bandwidth → increasing Concurrency
  - Increasing concurrency → decreasing locality
  - Decreasing locality → decreased DRAM efficiency
    - Unless compensated by massive reordering
  - Decreasing locality → decreasing energy efficiency

Latency-Bandwidth Products per Package (64B transfers)



+22%/year
2x / 3.5 years

Required Memory Bandwidth for Blocked DGEMM vs Maximum Sustainable Memory Bandwidth



What about Power/Energy?

- Power density is important in processor implementations
  - Frequencies can be limited by small-scale (core-sized) hot spots
  - Multi-core frequencies are now limited by package cooling
  - E.g., Xeon E5 v3 (Haswell) can only run DGEMM or LINPACK on ½ of the cores before running out of power & needing to throttle frequency
- Power is not a first-order concern in cost!!!
  - Purchase price is $2500-$4000/socket
  - Socket draws 100-150 Watts & needs 40-50 Watts for cooling
  - At $0.10/kWh, this is 5%-7% of purchase price per year
  - This ratio is very hard to change!!!

What about Power/Energy?

- Power density is important in processor implementations
  - Frequencies can be limited by small-scale (core-sized) hot spots
  - Multi-core frequencies are now limited by package cooling
  - E.g., Xeon E5 v3 (Haswell) can only run DGEMM or LINPACK on ½ of the cores before running out of power & needing to throttle frequency
- Power is not a first-order concern in cost!!!
  - Purchase price is $2500-$4000/socket
  - Socket draws 100-150 Watts & needs 40-50 Watts for cooling
  - At $0.10/kWh, this is 5%-7% of purchase price per year
  - This ratio is very hard to change!!!

Part 3

FUNDAMENTAL FLAWS IN CURRENT COMPUTER ARCHITECTURES

What is an “Architecture”?

- An “architecture” describes the ***explicit*** functionality of a computer system
  - This is typically expressed as a defined instruction set with required behaviors
- The architecture does not *limit* the functionality of the system, but only the *explicit functionality* is directly visible and only these explicit functions are *directly optimizable* functions.
- What if this explicit functionality does not represent the most important functions that need to be optimized?
  - Optimization would necessarily be indirect and almost certainly inefficient.

What is an “Architecture”?

- An “architecture” describes the ***explicit*** functionality of a computer system
  - This is typically expressed as a defined instruction set with required behaviors
- The architecture does not *limit* the functionality of the system, but only the *explicit functionality* is directly visible and only these explicit functions are *directly optimizable* functions.
- What if this explicit functionality does not represent the most important functions that need to be optimized?
  - Optimization would necessarily be indirect and almost certainly inefficient.

What is an “Architecture”?

- An “architecture” describes the ***explicit*** functionality of a computer system
  - This is typically expressed as a defined instruction set with required behaviors
- The architecture does not *limit* the functionality of the system, but only the *explicit functionality* is directly visible and only these explicit functions are *directly optimizable* functions.
- What if this explicit functionality does not represent the most important functions that need to be optimized?
  - Optimization would necessarily be indirect and almost certainly inefficient.

The Missing Elephant(s) in the Room

- So what are the most important functions to optimize?

- FIRST! Applications differ by orders of magnitude in their requirements for different performance components!

- Here I focus on the data motion issues suggested by the technology scaling:
  - Memory Access ("vertical" data motion)
  - Interprocessor Communication ("horizontal" data motion)

The Missing Elephant(s) in the Room

- So what are the most important functions to optimize?

- FIRST! Applications differ by many orders of magnitude in their requirements for different performance components!

- Here I focus on the data motion issues suggested by the technology scaling:
  - Memory Access (“vertical” data motion)
  - Interprocessor Communication (“horizontal” data motion)

Elephant #1: Vertical Data Motion

- Memory Access ("vertical" data motion)
  - A load from memory has an effective performance cost that is a random value between 0 and 1000 processor cycles.
  - The power cost is also a random value with a similarly large range.

- In current architectures Data Motion is invisible & uncontrollable
  - Easy to use, effectively impossible to optimize.
  - Cache hints are seldom useful
    - Limited by implementation details (e.g., inclusivity)
    - Limited by random page coloring, associativity, undocumented pseudo-LRU, etc.

Invisible, Uncontrollable Vertical Data Motion

- This was the right answer in the late 1980s and early 1990's….
  - Memory access was less expensive than arithmetic until ~1990.
  - Single-layer caches were only trying to hide a modest ratio of CPU cycle time to memory latency – e.g., 50 ns vs 400 ns (8:1) on my 1990-era IBM RS/6000-320.
  - Single processor systems did not need communication/synchronization.
- This was a reasonably practical hack for most of the 1990's
  - Performance gains from reduced system sharing and faster CPUs allowed for a very rapid rate of performance growth.
  - This allowed parallelism to remain modest for most users.

Invisible, Uncontrollable […] (cont’d)

- Invisible & uncontrollable is not an option if we need efficiency to increase and price to drop in the many-core era.
- Example: STREAM on Xeon E5-2660 v3 (Haswell EP)
  - Bandwidth (up to) 55.9 GB/s per socket (82% of peak)
    - 5 cores required to reach asymptotic BW
    - 1 core at <1 GHz could deliver corresponding FLOPS
  - Energy Use can be as low as 206 pJ/bit
    - 28 pJ/bit in DRAM, 178 pJ/bit in cores (using 5 cores)
  - Single core: 19.4 GB/s @ 1008 pJ/bit

Elephant #2: Horizontal Data Motion

- Interprocessor Communication (“horizontal” data motion)
  - Completely implicit in architectural specifications

- Example: Producer/consumer latency on Xeon E5-2680
  - Same chip: >200 cycles (~67 ns) (>1600 Peak FLOPS)
  - Different chips: >750 cycles (~245 ns) (>6000 Peak FLOPS)
  - Non-optimized implementations can be dramatically slower (10x or more)

- Example: OpenMP Barrier Synchronization on Xeon E5-2680:
  - 8 threads, same chip: ~1580 cycles (~510 ns) (>100k Peak FLOPS)
  - 16 threads, two chips: ~3875 cycles (~1250 ns) (~500k Peak FLOPS)

Elephant #2: Horizontal Data Motion

- Interprocessor Communication ("horizontal" data motion)
  - Completely implicit in architectural specifications

- Example: Producer/consumer latency on Xeon E5-2680
  - Same chip: >200 cycles (~67 ns) (>1600 Peak FLOPS)
  - Different chips: >750 cycles (~245 ns) (>6000 Peak FLOPS)
  - Non-optimized implementations can be dramatically slower (10x or more)

- Example: OpenMP Barrier Synchronization on Xeon E5-2680:
  - 8 threads, same chip: ~1580 cycles (~510 ns) (>100k Peak FLOPS)
  - 16 threads, two chips: ~3875 cycles (~1250 ns) (~500k Peak FLOPS)

Elephant #2: Horizontal Data Motion

- Interprocessor Communication ("horizontal" data motion)
  - Completely implicit in architectural specifications

- Example: Producer/consumer latency on Xeon E5-2680
  - Same chip: >200 cycles (~67 ns) (>1600 Peak FLOPS)
  - Different chips: >750 cycles (~245 ns) (>6000 Peak FLOPS)
  - Non-optimized implementations can be dramatically slower (10x or more)

- Example: OpenMP Barrier Synchronization on Xeon E5-2680:
  - 8 threads, same chip: ~1580 cycles (~510 ns) (>100k Peak FLOPS)
  - 16 threads, two chips: ~3875 cycles (~1250 ns) (~500k Peak FLOPS)

Part 4: “A New Hope”

AN ALTERNATIVE ARCHITECTURAL APPROACH

Caveat: What this is Not

- Existing architectures are remarkably well-suited to handling complex, branchy code with low rates of main memory access (excellent cache re-use) and low off-node bandwidth requirements.

- If your HPC application looks like this, then keep on using what is available!
  - NAMD uses ~30% of cycles on TACC's Stampede system
  - Extremely high cache re-use, almost no stalls, excellent parallel scaling
  - SIMD vectorization does not help performance, but otherwise excellent

Target of this set of proposals

- Enable dramatically reduced cost and energy consumption in algorithms that are limited by vertical or horizontal data motion on current systems.
  - Note that significantly reduced acquisition cost will make power a first-order expense unless power is also significantly reduced
  - I.e., a $5 processor must draw 0.5W or less to keep power cost at 5%
- Big gains come from exposing functions to HW that are currently inefficient due to architectural choices.

A New Paradigm: Quit Fighting Physics

1. Don’t throw away information!
   - Information about spatial locality & about temporal locality
   - Semantic information: e.g., private memory references vs communication
2. Don’t move data if you don’t need to move it!
   - If you do need to move it, control the motion precisely (where/when/how)
3. Don’t use expensive processors to do simple computations!
   - Distribute highly efficient processors everywhere in the system
4. Don’t use a serial programming model + hacks!
   - Exploit the human brain’s ability to understand causality

1. Don’t throw away information

- Data Motion is a first-order feature of the architecture
  - Information about expected memory reference patterns must be conveyed from the source code through the compiler to the processor core and then to the memory hierarchy.
- Communication is a first-order feature of the architecture.
  - Communication is distinguishable from private memory references and will be treated differently by the hardware.

2. Don’t move data unless necessary

- Move computation to memory, not data to processor
  - “Processor At Memory”, not necessarily “In Memory”
- Vertical Data Motion:
  - Accesses that can be analyzed should go through explicitly controlled non-coherent scratchpad memories.
  - Give memory controllers visibility into future access patterns – no need for huge, expensive reorder buffers to get excellent performance at minimum energy cost.
- Horizontal Data Motion
  - Communication & Synchronization in a single message

2a. Vertical Data Motion

- Data motion through an invisible, uncontrollable hierarchy can be done (of course), and done well in some cases, but it can't be done well at low complexity and low power consumption.
  - Development cost & power consumption must both be very low for processors to be cheap enough to distribute to the DRAMs.
- Scratchpad memories have lower power, smaller size, no impact on latency to more distant memory, and no controllability problems due to limited associativity and LRU mechanisms.
  - Vector load/store architecture to explicit on-chip scratchpad memory

2b. Horizontal Data Motion

- Communication & Synchronization in a single message
  - Requires metadata – full/empty (valid/invalid) bit is a minimum
- Memory references with side effects can be used to couple processors to hardware FIFOs and other very efficient mechanisms for horizontal data motion via dataflow.
  - Current processors are full of efficient HW FIFOs that SW cannot use
- Mailboxes, doorbells, & work queues can allow much smaller overhead for fine-grained parallelism using control flow.
  - MDP(*,1987) message processing in <1 usec with 10 MHz CPU.

(*) Dally, et al. Message-Driven-Processor (1987)

3. Use efficient processors

- Distributed processors don’t need expensive features
  - No caches, minimum translation/protection
  - Accesses that can be analyzed should go through explicitly controlled non-coherent scratchpad memories.
- Don’t provide more resources than are practically useful, but make sure they are optimized for the task
  - Vector processors with scratchpad memories
  - Not SIMD – this is neither needed, nor desirable
  - Linear Algebra/FFT PE (*) – 2 GFLOPS (scalar 64-bit FP), 0.05 Watts (max), 0.12 mm^2 in 45nm

(*) Pedram, McCalpin, Gerstlauer 2013 & 2014

4. Don’t use serial programming models

- *“[…] non-trivial multi-threaded programs are incomprehensible to humans.”* (Edward Lee, Berkeley, 2006)

- BUT, humans intuitively understand causality, so data dependence is not just comprehensible, it is natural

- The challenge is to build an intrinsically parallel programming model that can be efficiently mapped to hardware that has an intrinsic hierarchical structure that is not derived from the problem that we want to solve.

Programming Models (cont'd)

- Sequoia had some good ideas on how to manage the memory hierarchy
  - Not particularly successful on cached systems because they are not actually controllable (associativity, undocumented LRU, undocumented prefetchers, etc).
- Victor Eijkhout: Integrated Model for Parallelism (IMP)
  - Specifies data dependence for computational kernels, compiler and run-time derive the required communication and scheduling.
  - Analyzable HW + Analyzable SW → productivity and efficiency

Summary & Closing Thoughts

- Market + Technology are keeping us stuck with outdated architectures that cannot deal with increasing parallelism
- Physics dictates that reduced cost & increased energy efficiency must come from even more parallelism
- Good News: Some things are slow because the architecture does not make them explicit – we can fix these by exposing this functionality & allowing HW to be optimized for them
- Programming models based on data dependence can allow high-level expression and much more effective automatic program transformations.

Summary & Closing Thoughts

- Market + Technology are keeping us stuck with outdated architectures that cannot deal with increasing parallelism
- Physics dictates that reduced cost & increased energy efficiency must come from even more parallelism
- Good News: Some things are slow because the architecture does not make them explicit – we can fix these by exposing this functionality & allowing HW to be optimized for them
- Programming models based on data dependence can allow high-level expression and much more effective automatic program transformations.

Summary & Closing Thoughts

- Market + Technology are keeping us stuck with outdated architectures that cannot deal with increasing parallelism
- Physics dictates that reduced cost & increased energy efficiency must come from even more parallelism
- Good News: Some things are slow because the architecture does not make them explicit – we can fix these by exposing this functionality & allowing HW to be optimized for them
- Programming models based on data dependence can allow high-level expression and much more effective automatic program transformations.

Summary & Closing Thoughts

- Market + Technology are keeping us stuck with outdated architectures that cannot deal with increasing parallelism
- Physics dictates that reduced cost & increased energy efficiency must come from even more parallelism
- Good News: Some things are slow because the architecture does not make them explicit – we can fix these by exposing this functionality & allowing HW to be optimized for them
- Programming models based on data dependence can allow high-level expression and much more effective automatic program transformations.

John D. McCalpin, PhD

mccalpin@tacc.utexas.edu

512-232-3754

For more information:

www.tacc.utexas.edu

TACC

Appendix

BACKUP SLIDES

Change of System Sizes

- Since ~2003, the list has been dominated by clusters of “small-node” systems.
- Since ~2005 these clusters have been dominated by x86.
- Sizes have changed qualitatively:
  - 1995: 60% of systems had <= 16 cores – easy!
  - 2000: 60% of systems had <= 128 cores – a bit of work
  - 2005: >50% of systems had > 512 cores – a lot of work
  - 2010: >80% of systems had > 4096 cores – beyond almost all users
  - 2015: >70% of systems have > 16,384 cores – only a few users?