

Advanced Computing Architectures

Alessandro Pellegrini

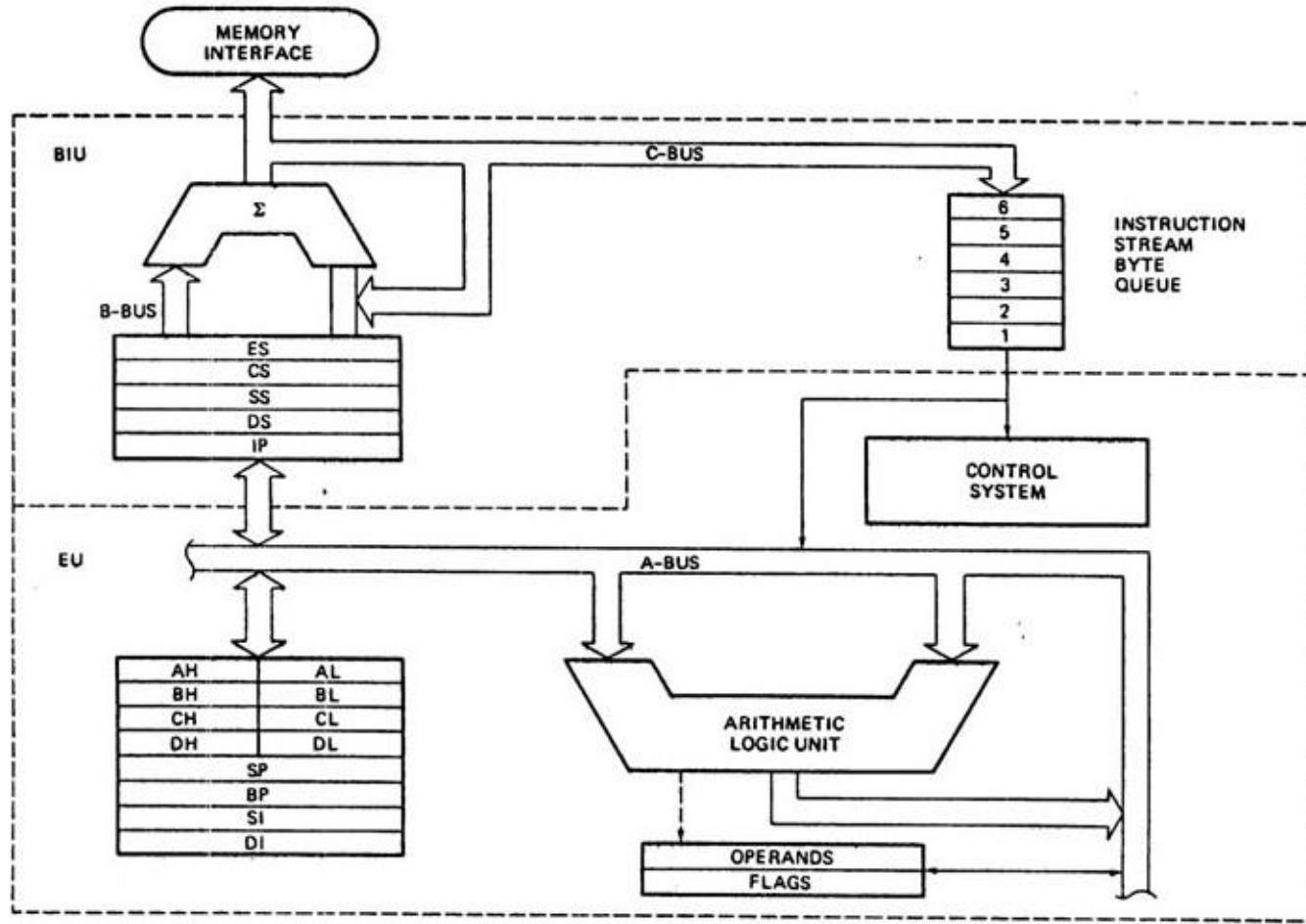
A.Y. 2019/2020



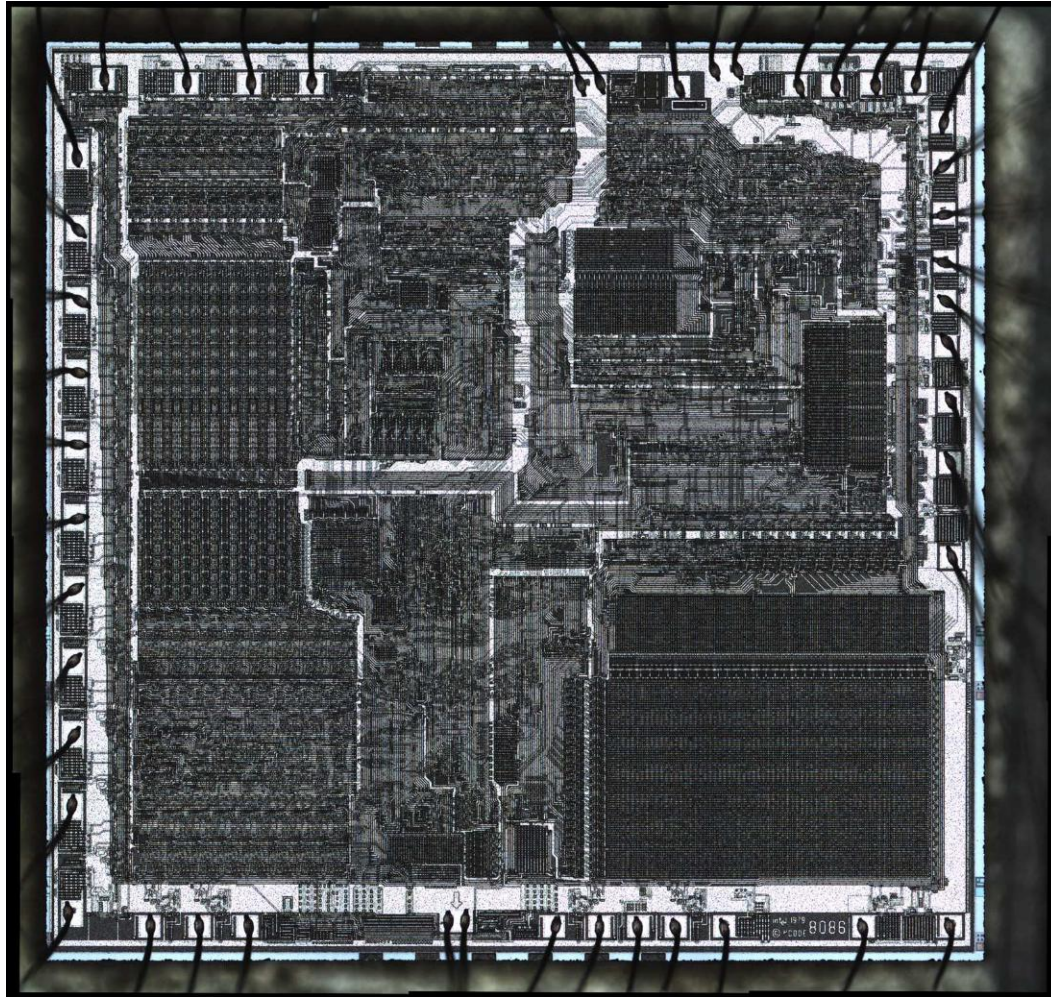
SAPIENZA

UNIVERSITÀ DI ROMA

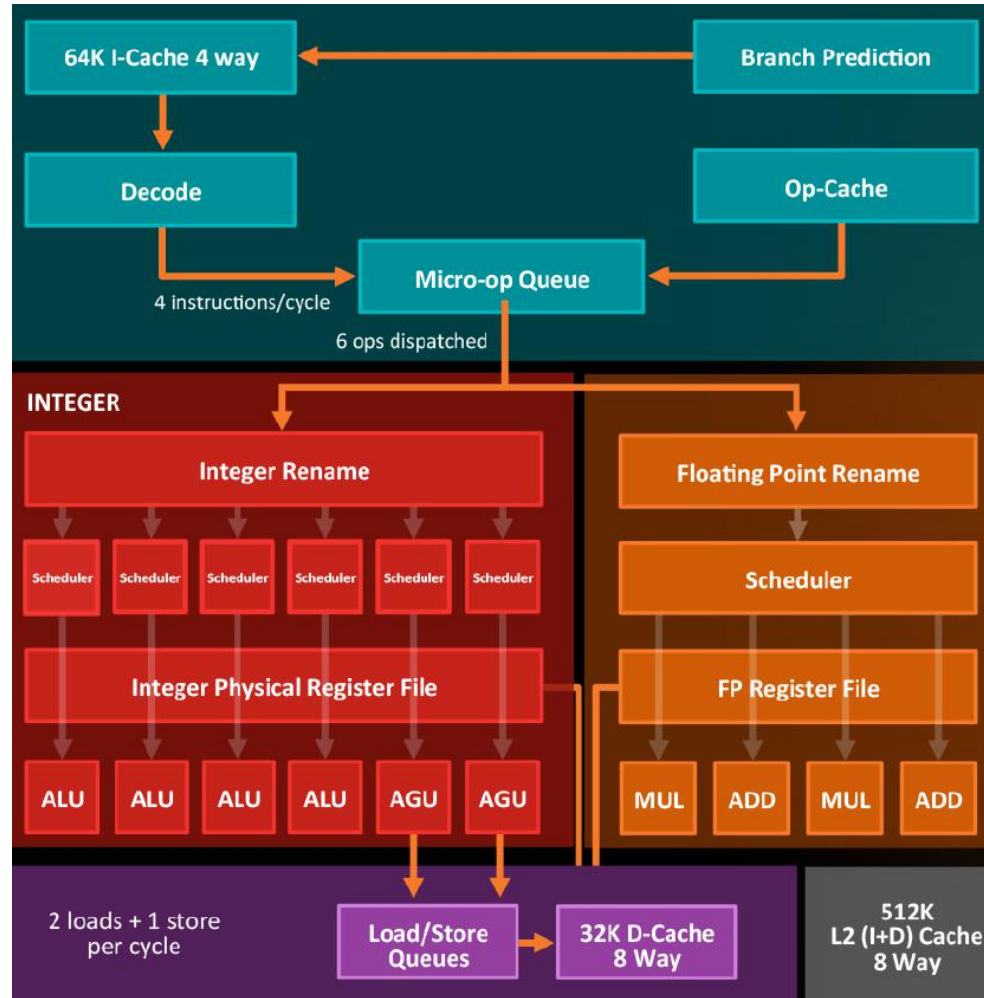
Intel 8086 (1978)



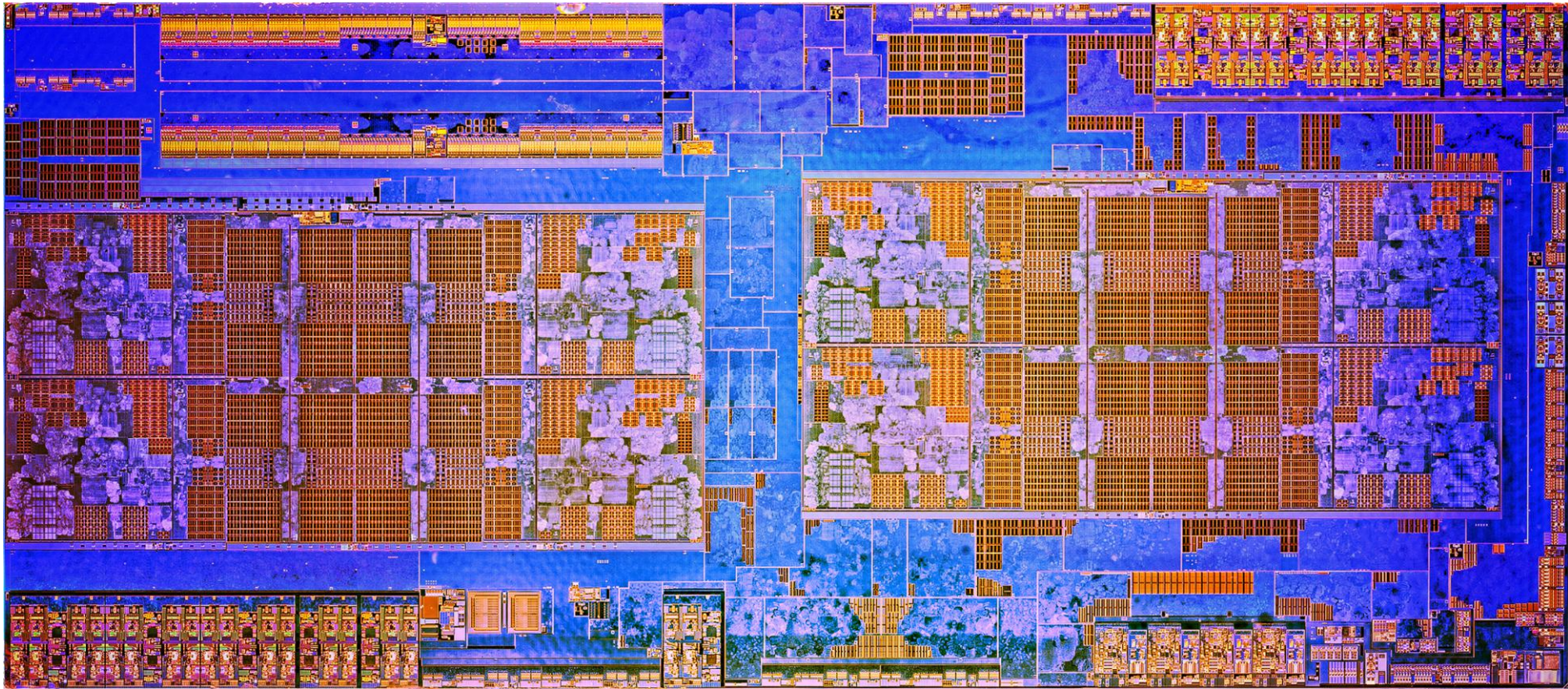
Intel 8086 (1978)



AMD Ryzen (2017)



AMD Ryzen (2017)



Moore's Law (1965)

The number of transistors in a dense integrated circuit doubles approximately every two years

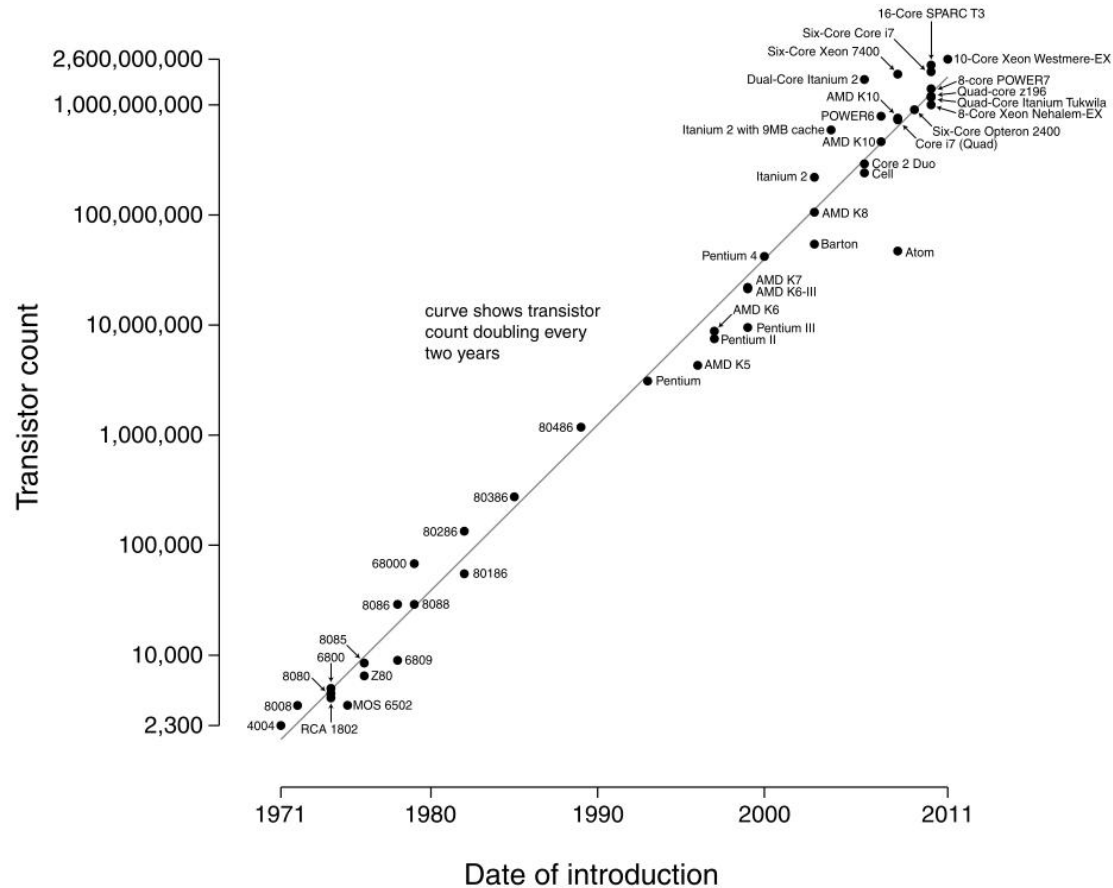
— *Gordon Moore, Co-founder of Intel*

Moore's law is dead, long live Moore's law

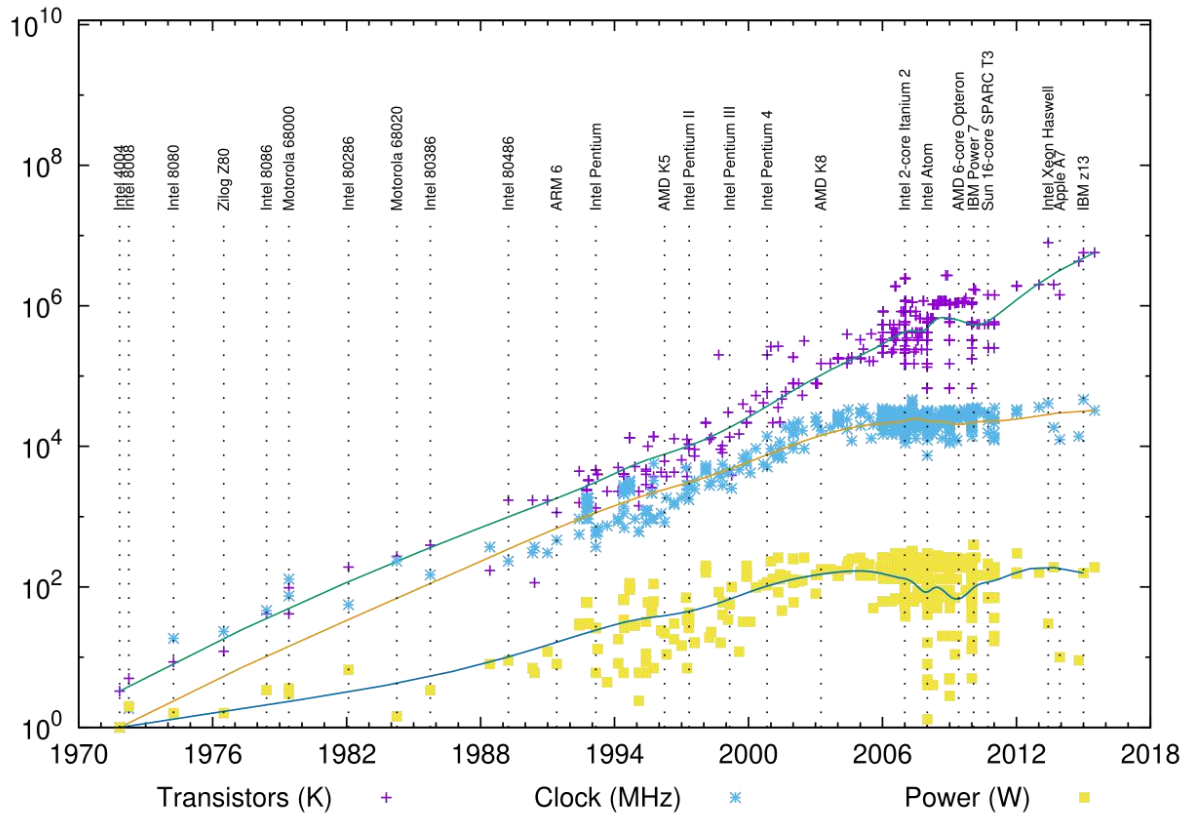


Moore's Law (1965)

Microprocessor Transistor Counts 1971-2011 & Moore's Law



The free lunch

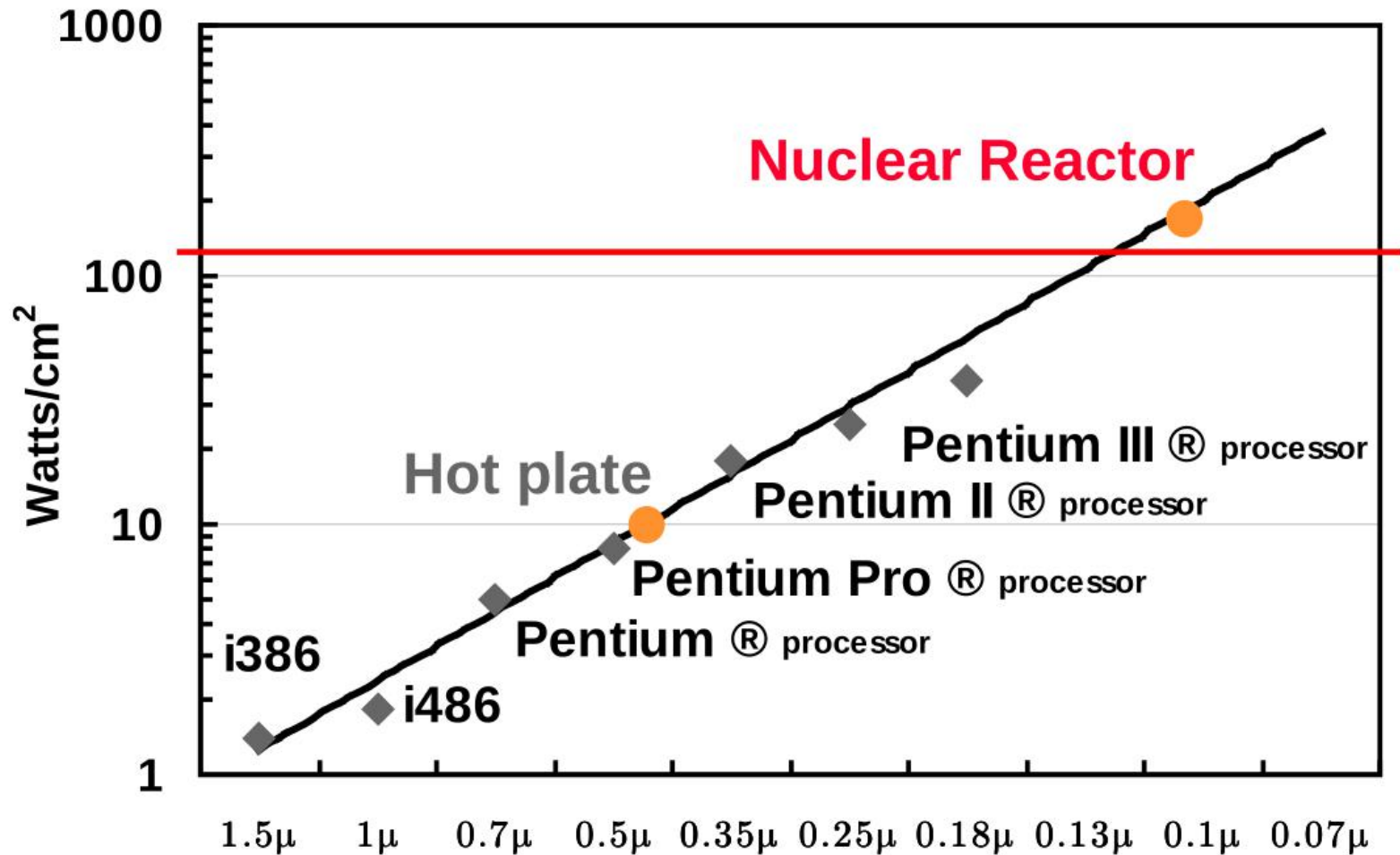


- Implications of Moore's Law have changed since 2003
- 130W is considered an upper bound (the *power wall*)

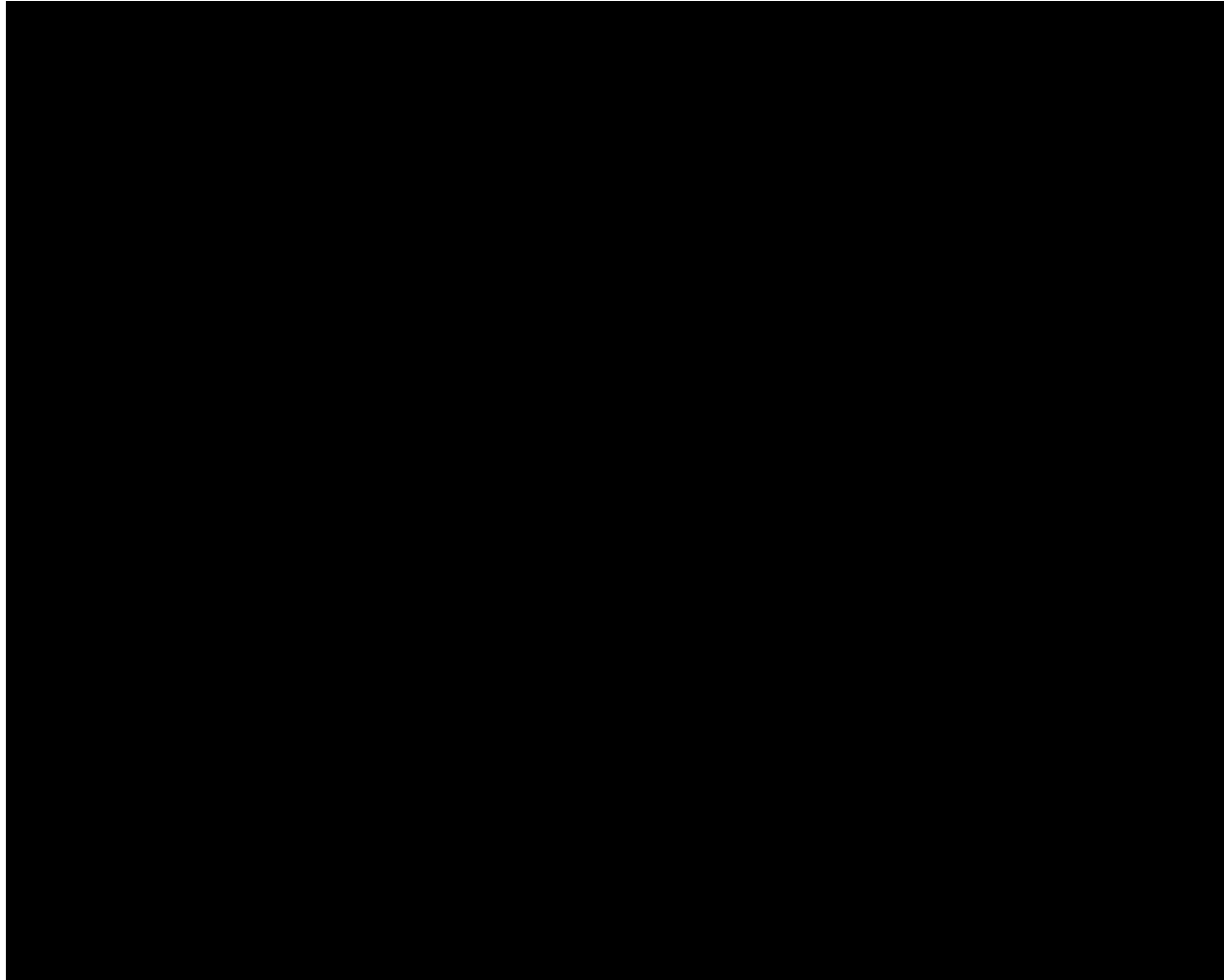
$$P = ACV^2f$$



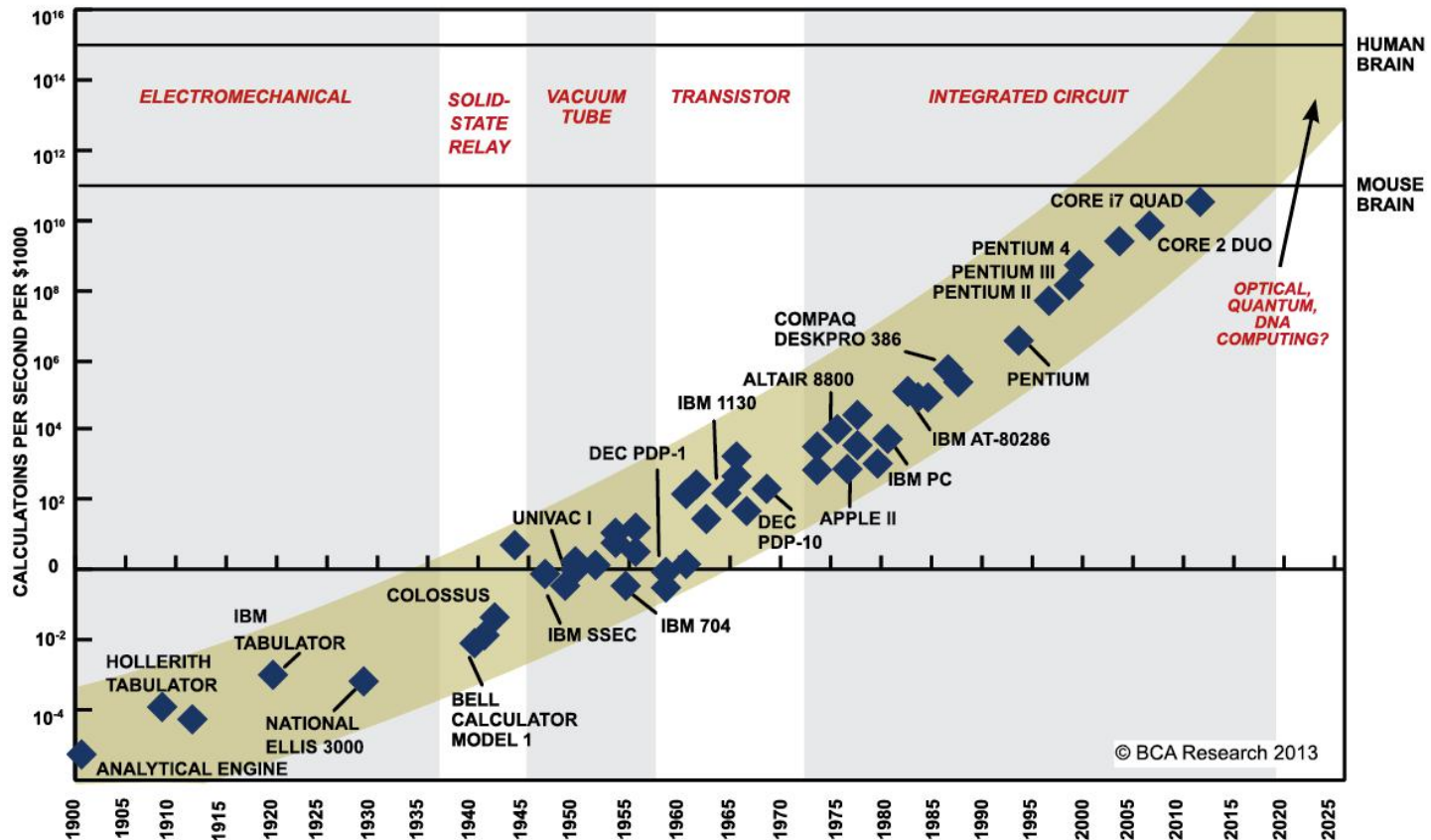
The Power Wall



The *Power Wall*



Performance over Time

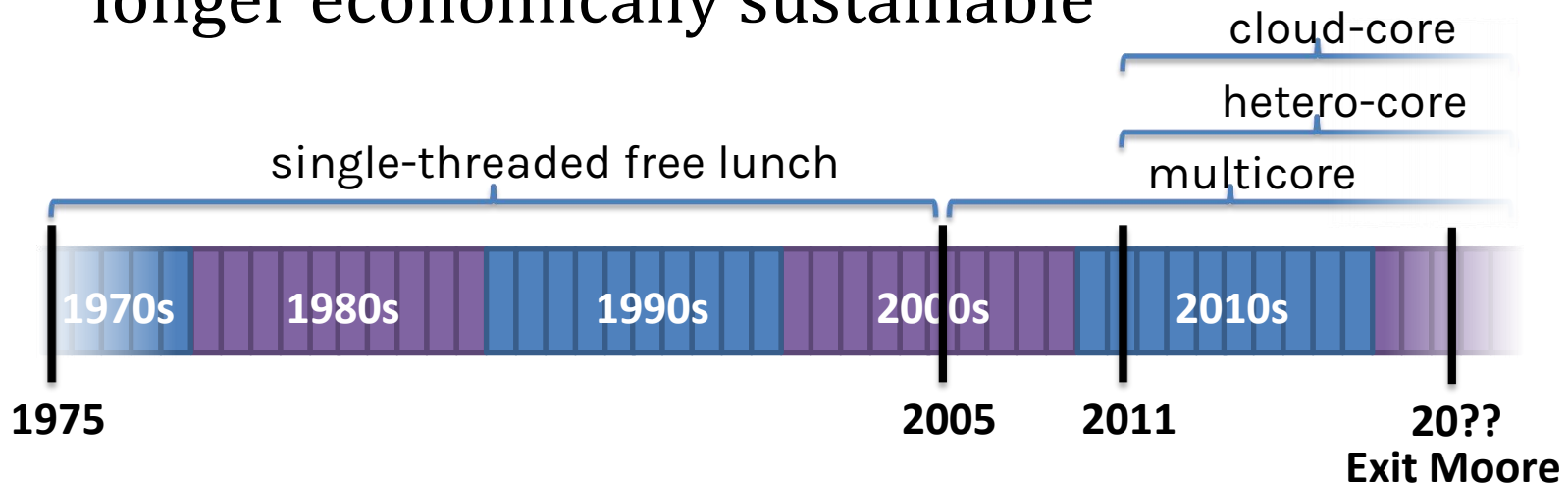


SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, THE VIKING PRESS, 2006. DATAPPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.



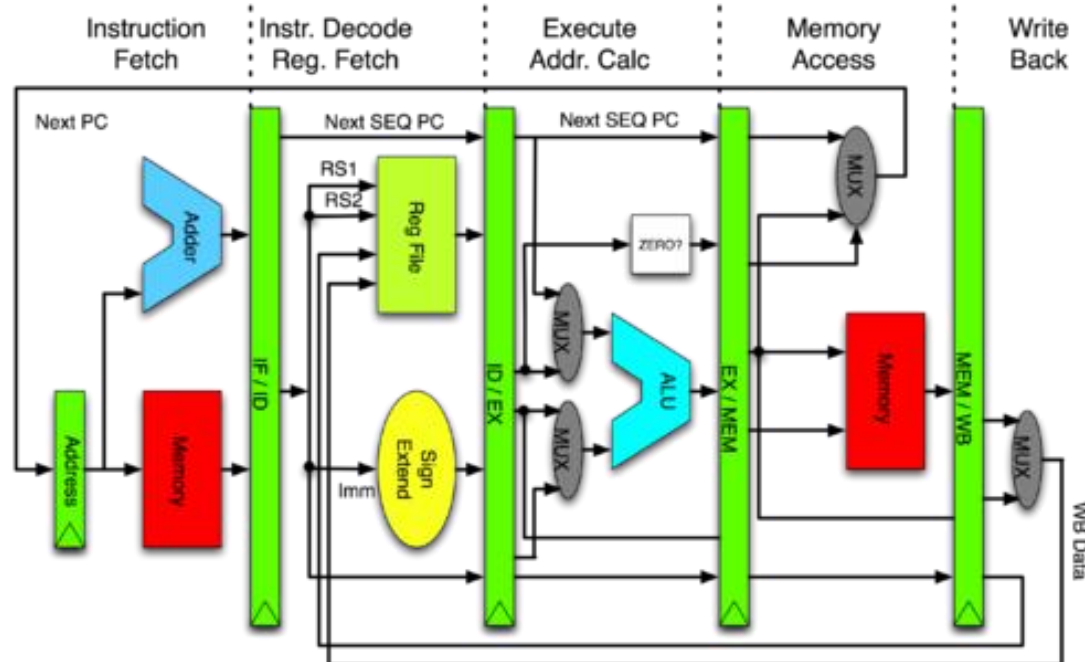
Mining Moore's Law

- We can look at Moore's law as a gold mine:
 - You start from the motherlode
 - When it's exhausted, you start with secondary veins
 - You continue mining secondary veins, until it's no longer economically sustainable



Trying to speedup: the pipeline (1980s)

- Temporal parallelism
- Number of stages increases with each generation
- Maximum Cycles Per Instructions (CPI)=1



Superscalar Architecture (1990s)

- More instructions are simultaneously executed on the same CPU
- There are redundant functional units that can operate in parallel
- Run-time scheduling (in contrast to compile-time)

IF	ID	EX	MEM	WB					
IF	ID	EX	MEM	WB					
	IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB				
		IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB	



Speculation

- In what stage does the CPU fetch the next instruction?
- If the instruction is a conditional branch, when does the CPU know whether the branch is taken or not?
- *Stalling* has a cost:
 $nCycles \cdot branchFrequency$
- A guess on the outcome of a compare is made
 - if wrong the result is discarded
 - if right the result is flushed

```
a ← b + c
if a ≥ 0 then
    d ← b
else
    d ← c
end if
```



Branch Prediction

- Performance improvement depends on:
 - whether the prediction is correct
 - how soon you can check the prediction
- Dynamic branch prediction
 - the prediction changes as the program behaviour changes
 - implemented in hardware
 - commonly based on branch history
 - predict the branch as taken if it was taken previously
- Static branch prediction
 - compiler-determined
 - user-assisted (e.g., `likely` in kernel's source code; `0x2e`, `0x3e` prefixes for Pentium 4)

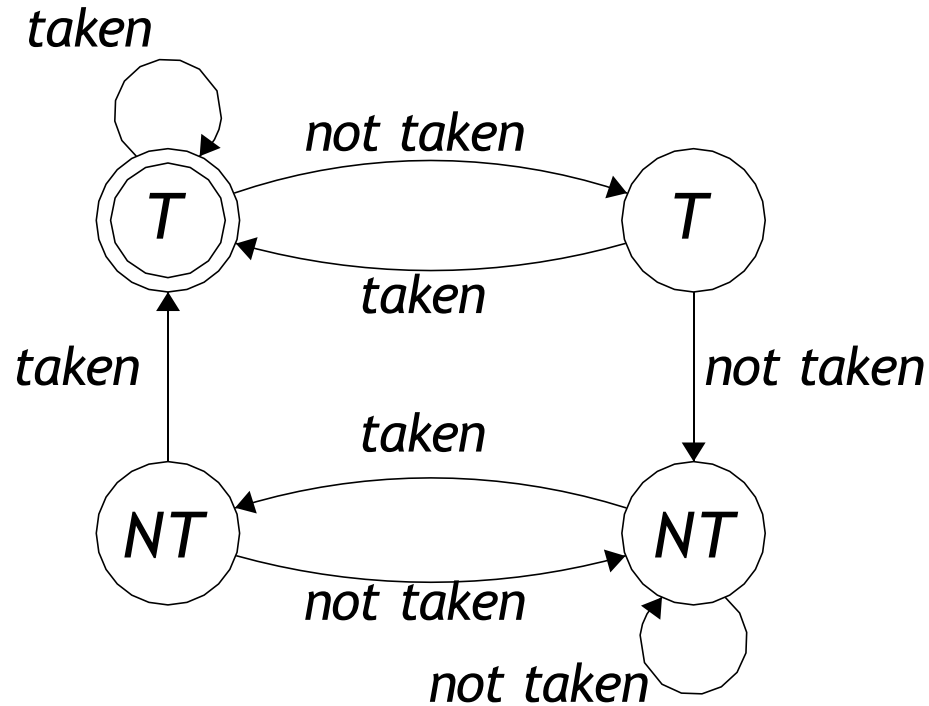


Branch Prediction Table

- Small memory indexed by the lower bits of the address of conditional branch instruction
- Each instruction is associated with a prediction
 - *Take* or *not take* the branch
- If the prediction is *take* and it is correct:
 - Only one cycle penalty
- If the prediction is *not take* and it is correct:
 - No penalty
- If the prediction is *incorrect*:
 - Change the prediction
 - Flush the pipeline
 - Penalty is the same as if there were no branch prediction



Two-bit Saturating Counter



Branch Prediction is Important

- Conditional branches are around 20% of the instructions in the code
- Pipelines are deeper
 - A greater misprediction penalty
- Superscalar architectures execute more instructions at once
 - The probability of finding a branch in the pipeline is higher
- Object-oriented programming
 - Inheritance adds more branches which are harder to predict
- Two-bits prediction is not enough
 - Chips are denser: more sophisticated hardware solutions could be put in place



How to Improve Branch Prediction?

- Improve the prediction
 - Correlated (two-levels) predictors [Pentium]
 - Hybrid local/global predictor [Alpha]
- Determine the target earlier
 - Branch target buffer [Pentium, Itanium]
 - Next address in instruction cache [Alpha, UltraSPARC]
 - Return address stack [Consolidated into all architecture]
- Reduce misprediction penalty
 - Fetch both instruction streams [IBM mainframes]



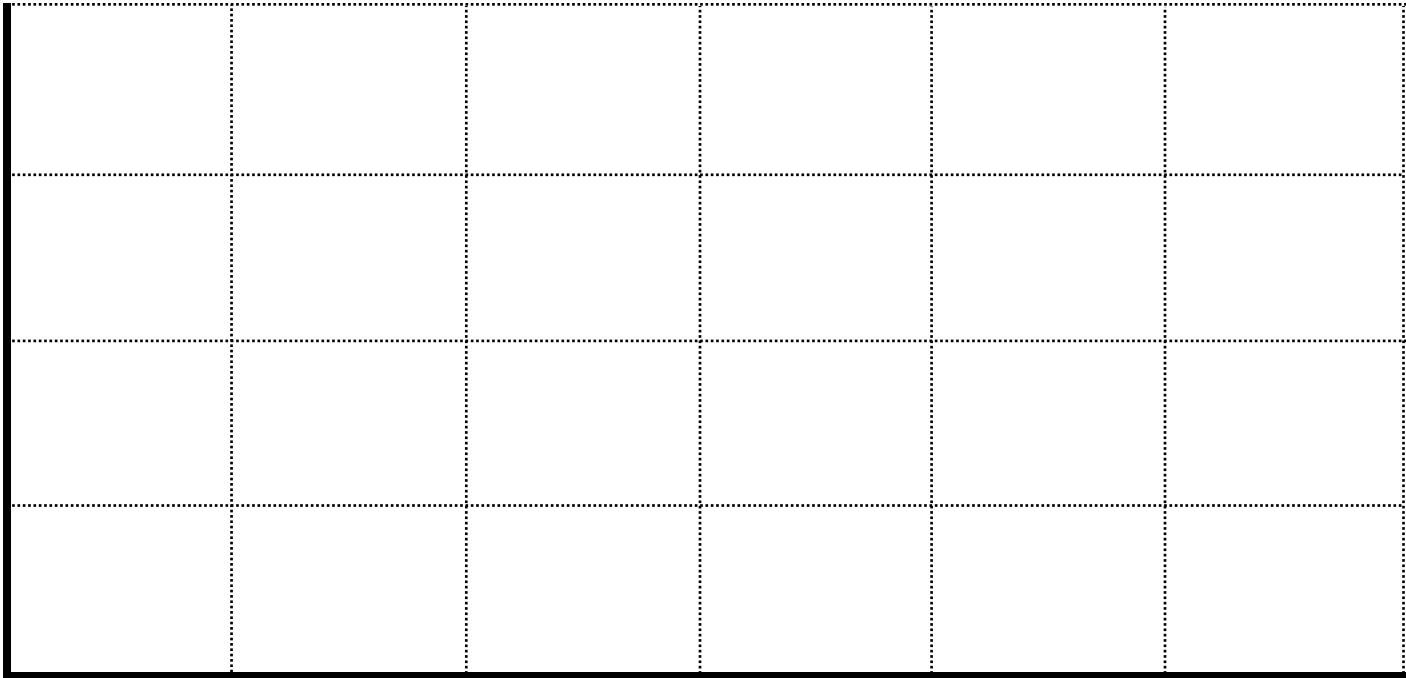
Return Address Stack

- Registers are accessed several stages after instruction's fetch
- Most of indirect jumps (85%) are function-call returns
- Return address stack:
 - it provides the return address early
 - this is pushed on call, popped on return
 - works great for procedures that are called from multiple sites
 - BTB would predict the address of the return from the last call



Charting the landscape

Processors



Memory



Charting the landscape: Processors



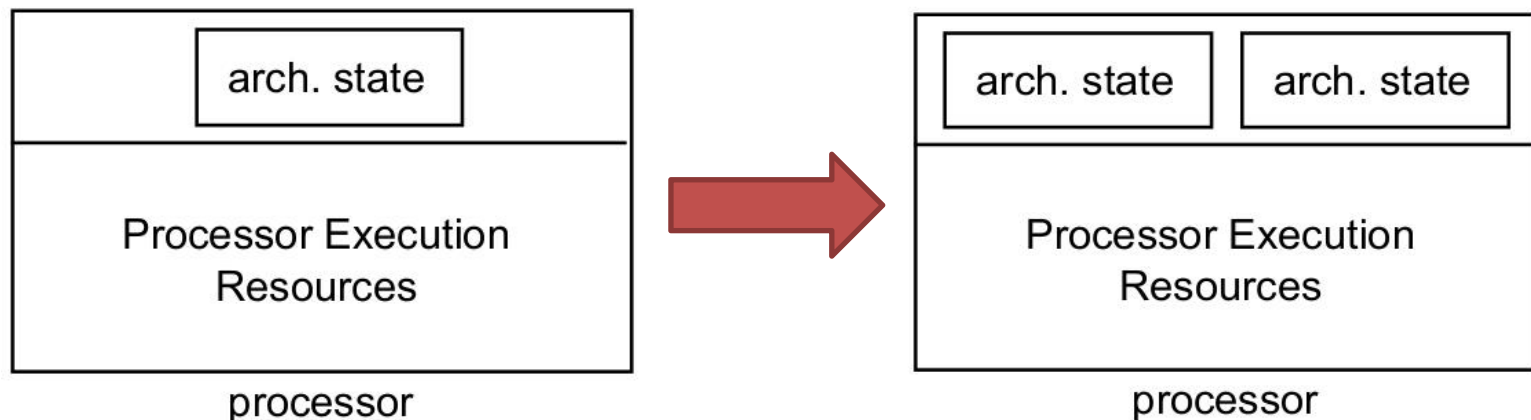
- The “big” cores: they directly come from the motherlode
- Best at running sequential code
- Any inexperienced programmer should be able to use them effectively

Memory



Simultaneous Multi-Threading—SMT (2000s)

- A physical processor appears as multiple logical processors
- There is a copy of the architecture state (e.g., control registers) for each logical processor
- A single set of physical execution resources is shared among logical processors
- Requires less hardware, but some sort of arbitration is mandatory



The Intel case: Hyper-Threading on Xeon CPUs

- **Goal 1:** minimize the die area cost (share of the majority of architecture resources)
- **Goal 2:** when one logical processor stalls, the other logical process can progress
- **Goal 3:** in case the feature is not needed, incur in no cost penalty
- The architecture is divided into two main parts:
 - Front end
 - Out-of-order execution engine

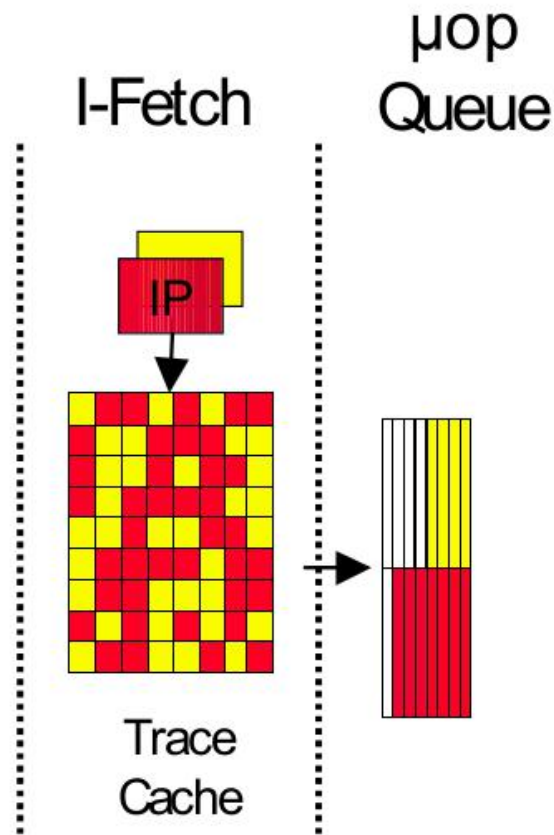


Xeon Front End

- The goal of this stage is to deliver instruction to later pipeline
- stages
- Actual Intel's cores do not execute CISC instructions
 - Intel instructions are cumbersome to decode: variable length, many different options
 - A Microcode ROM decodes instructions and converts them into a set of semantically-equivalent RISC μ -ops
- μ -ops are cached into the Execution Trace Cache (TC)
- Most of the executed instructions come from the TC



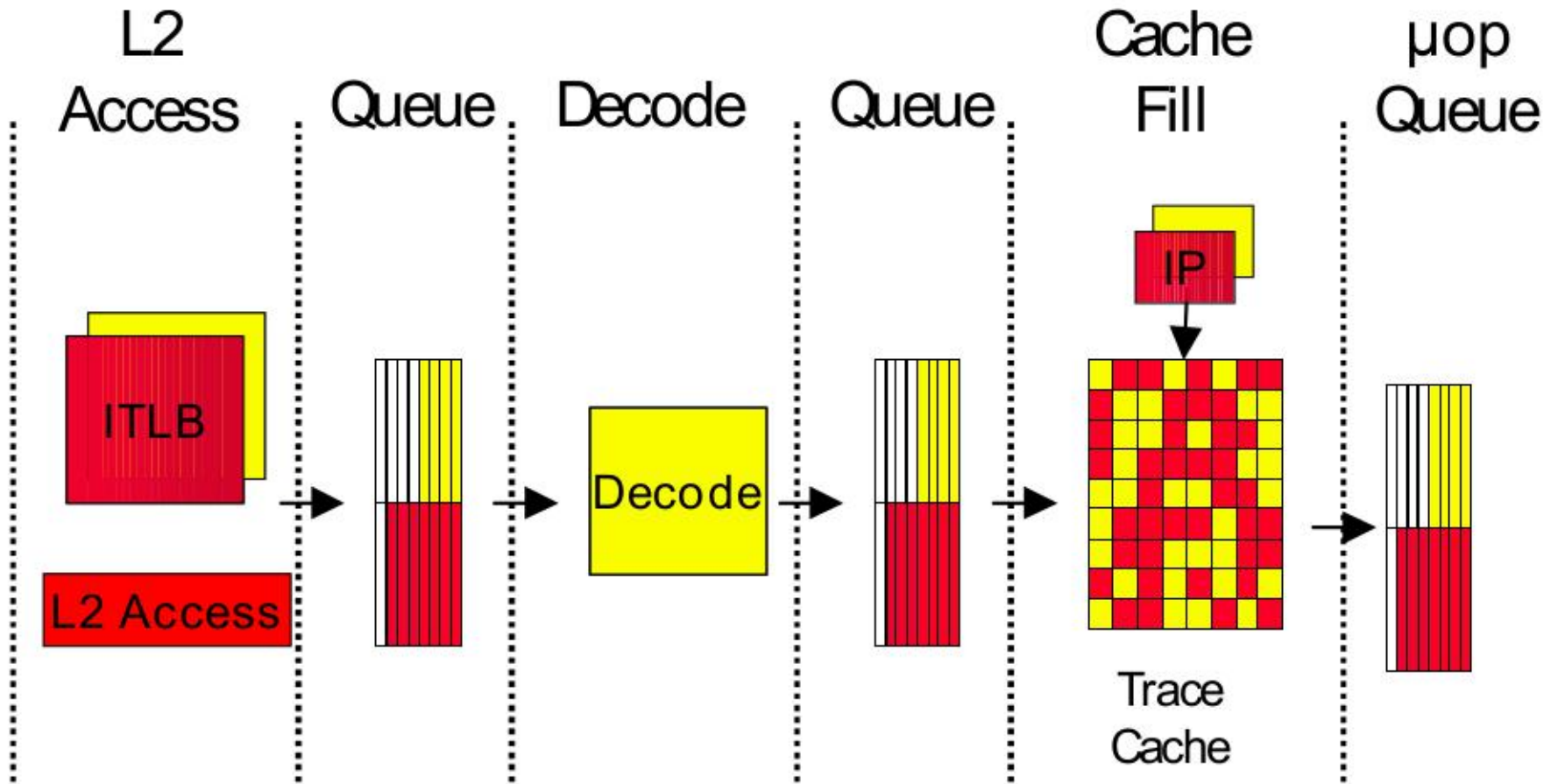
Trace Cache Hit



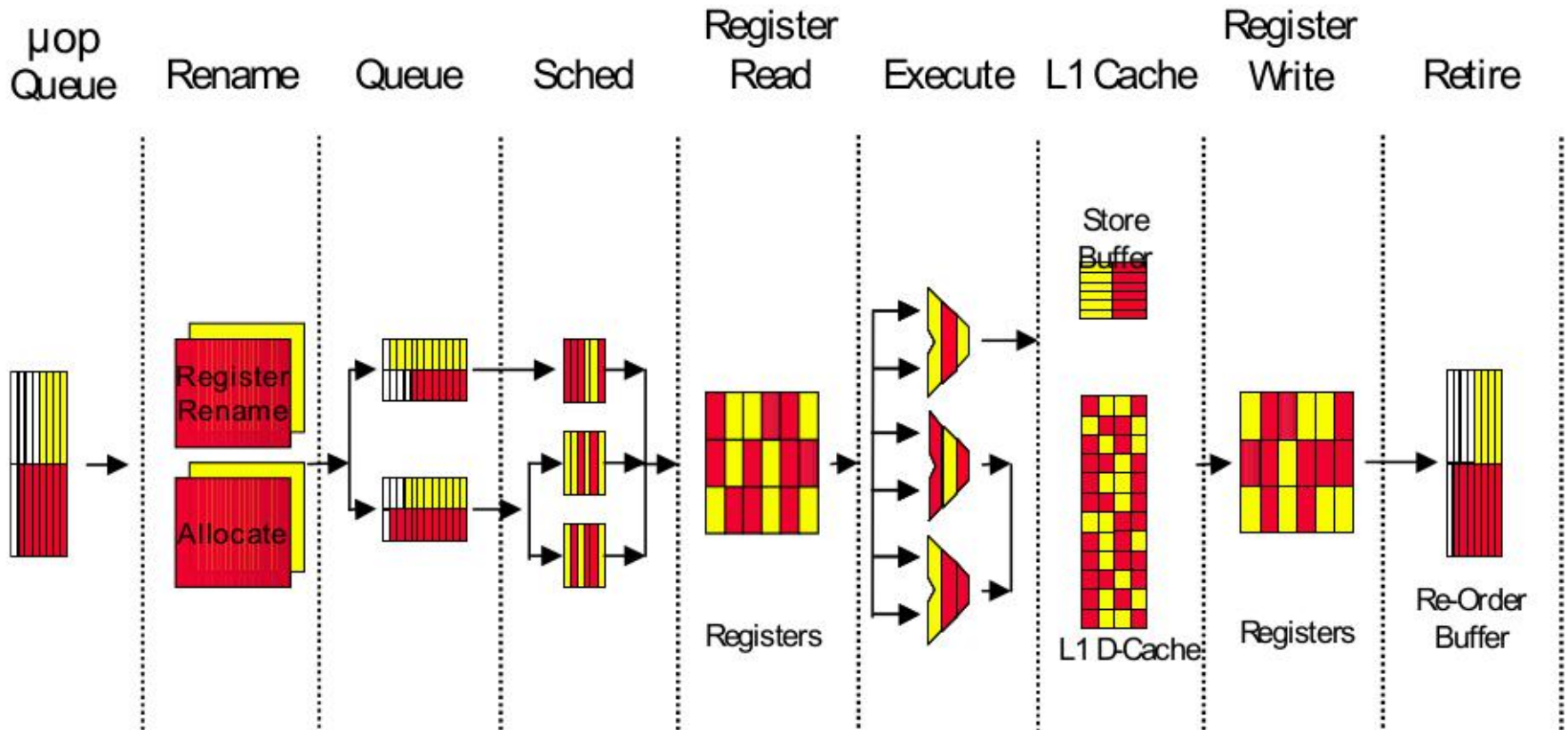
- Two sets of next-instruction pointers
- Access to the TC is arbitrated among logical processors at each clock cycle
- In case of contention, access is alternated
- TC entries are tagged with thread information
- TC is 8-way associative, entries are replaced according to a LRU scheme



Trace Cache Miss

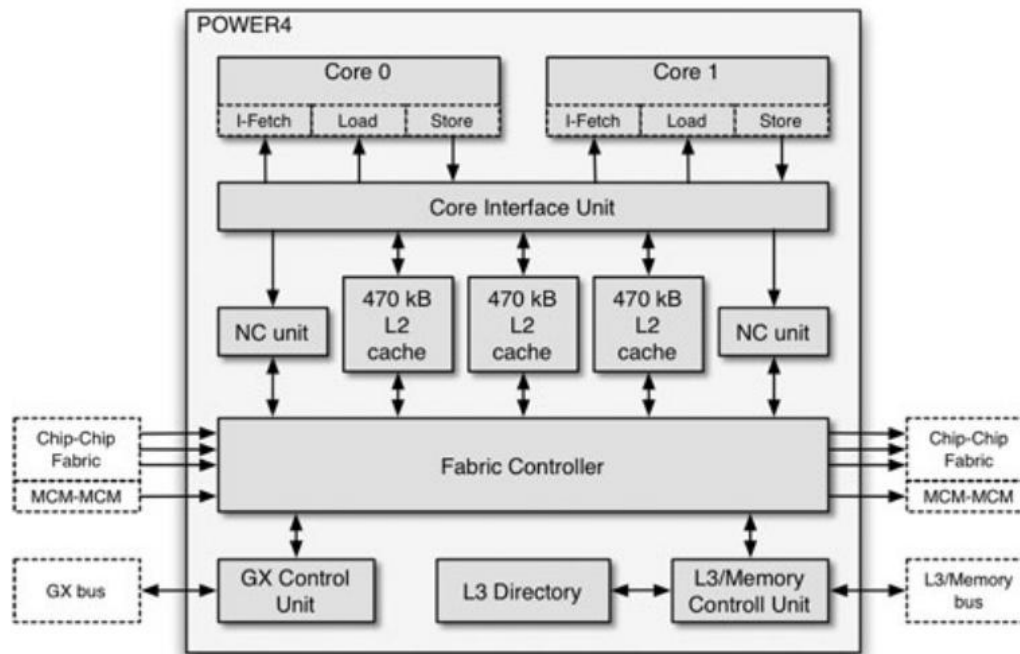


Xeon Out-of-order Pipeline

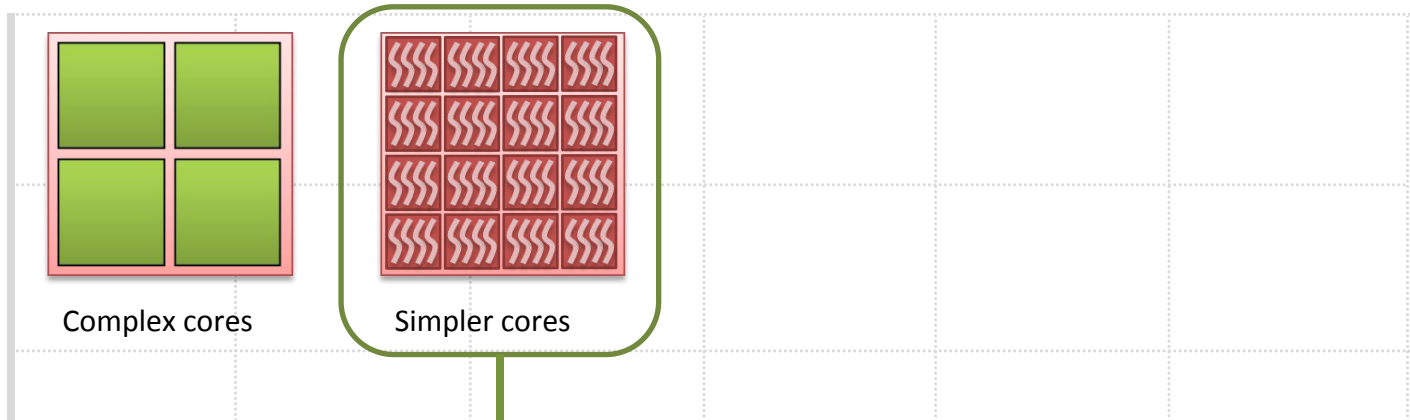


Multicores (2000s)

- First multicore chip: IBM Power4 (1996)
- 1.3 GHz dual-core PowerPC-based CPU



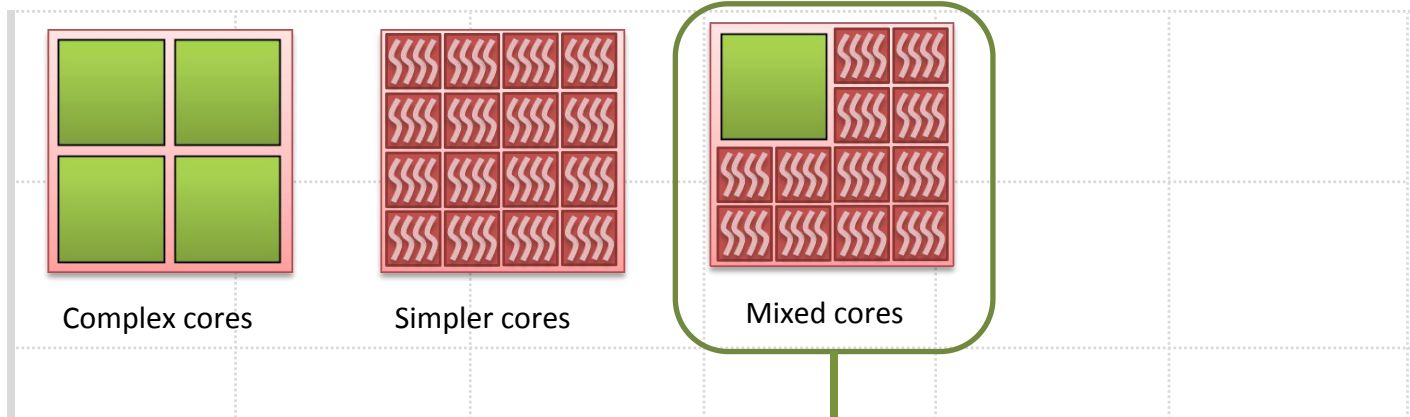
Charting the landscape: Processors



- “small” traditional cores
- Best at running parallelizable code, that still requires the full expressiveness of a mainstream programming language
- They could require “programming hints” (annotations)
- They make parallelism mode explicit



Charting the landscape: Processors



- A simpler core takes less space
- You can cram more on a single die in place of a complex core
- A good tradeoff for energy consumption
- Good to run legacy code and exploit benefits of Amdahl's law
- Typically they have the same instruction set (ARM big.LITTLE/DynamiQ), but some don't (CellBE)

Memory

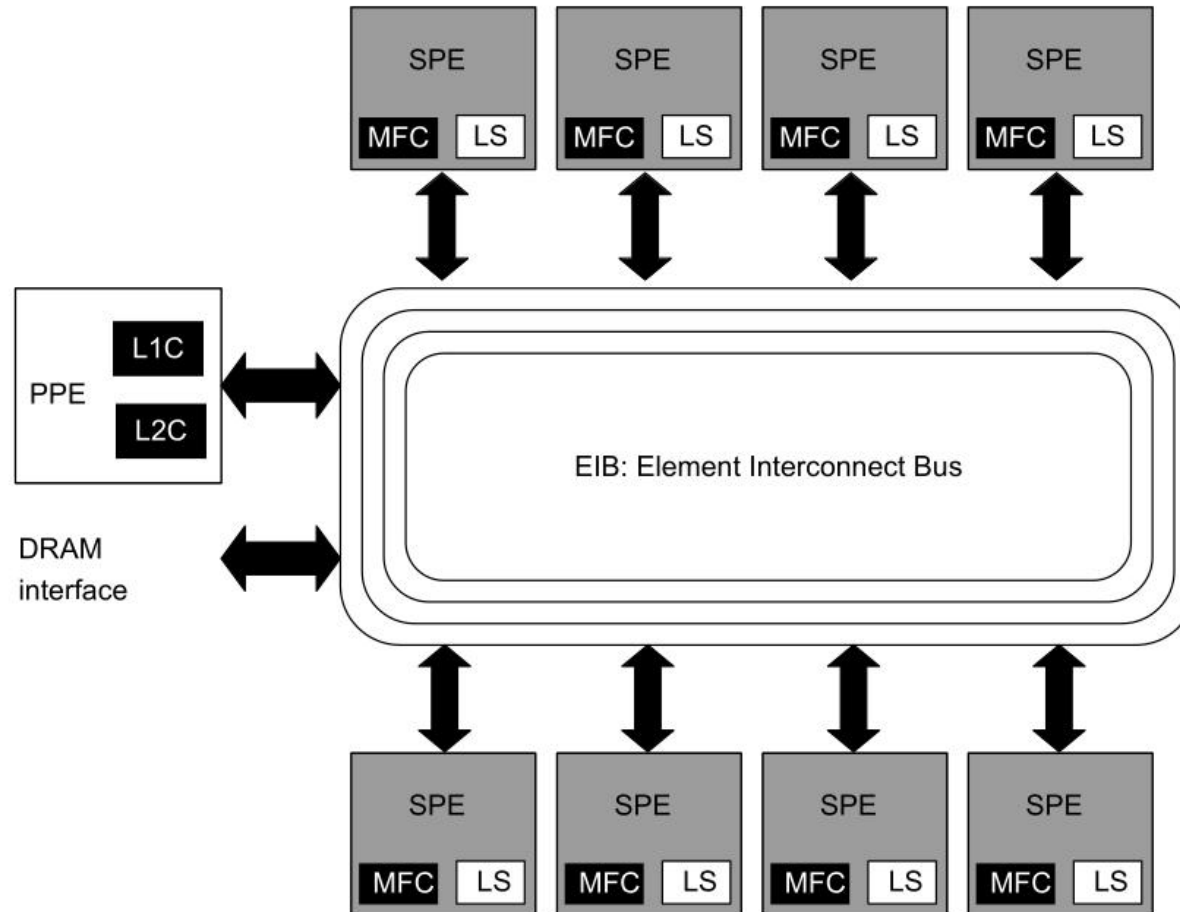


The Cell Broadband Engine (Cell BE)

- Produced in co-operation by Sony, Toshiba, and IBM
- Nine cores interconnected through a circular bus—Element Interconnect Bus (EIB):
 - one Power Processing Element (PPE): 3.2 GHz 2-way SMT processor
 - eight Synergistic Processing Elements (SPE): SIMD processors, 256 Kb of local memory
- The priority is performance over programmability
- SPEs need to be managed (offloading) explicitly by the PPE
- SPEs are number crunching elements
- Many applications: video cards, video games (Playstation 3), home cinema, supercomputing, ...



The Cell Broadband Engine (Cell BE)



Cell BE programming: the PPE

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <errno.h>
4 #include <libspe2.h>
5 #include <pthread.h>
6
7 /* a handle to the program that will be offloaded to the SPEs */
8 extern spe_program_handle_t simple_spu;
9 #define MAX_SPU_THREADS 8
10 void *ppu_thread_function(void *arg);
11
12 int main(void) {
13     int i, spu_threads;
14     spe_context_ptr_t ctxs[MAX_SPU_THREADS];
15     pthread_t threads[MAX_SPU_THREADS];
16
17     /* Determine the number of SPE threads to create */
18     spu_threads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
```



Cell BE programming: the PPE

```
19     if (spu_threads > MAX_SPU_THREADS)
20         spu_threads = MAX_SPU_THREADS;
21
22     /* Create several SPE-threads to execute 'simple_spu' */
23     for(i = 0; i < spu_threads; i++) {
24         /* Create context */
25         if ((ctxs[i] = spe_context_create(0, NULL)) == NULL) {
26             perror ("Failed creating context");
27             exit(1);
28         }
29
30         /* Load program into context */
31         if (spe_program_load (ctxs[i], &simple_spu)) {
32             perror ("Failed loading program");
33             exit(1);
34         }
35
36
```



Cell BE programming: the PPE

```
37     /* Create thread for each SPE context */
38     if (pthread_create (&threads[i], NULL, &ppu_thread_function, &ctxs[
        i])) {
39         perror ("Failed creating thread");
40         exit(1);
41     }
42
43     /* Wait for SPU-thread to complete execution. */
44     for (i = 0; i < spu_threads; i++) {
45         if (pthread_join (threads[i], NULL)) {
46             perror("Failed pthread_join");
47             exit(1);
48         }
49     }
50     return 0;
51 }
52
53
```



Cell BE programming: the PPE

```
54
55 void *ppu_thread_function(void *arg) {
56     spe_context_ptr_t ctx;
57     unsigned int entry = SPE_DEFAULT_ENTRY;
58     ctx = *((spe_context_ptr_t *)arg);
59
60     if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
61         perror ("Failed running context");
62         exit(1);
63     }
64     pthread_exit(NULL);
65 }
```



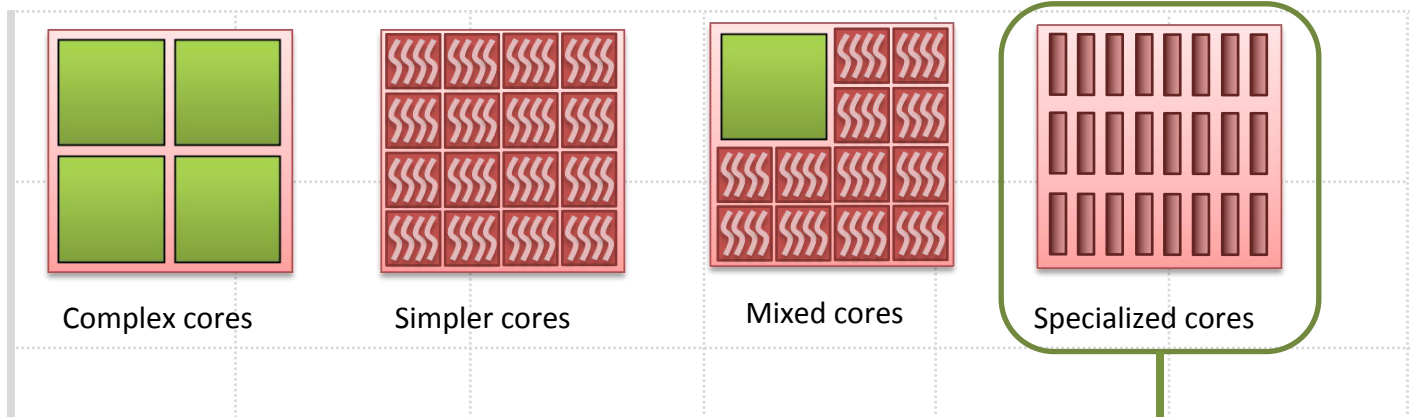
Cell BE programming: the SPE

```
1 #include <stdio.h>
2
3 int main(unsigned long long id) {
4     /* The first parameter of an spu program will always be the spe_id
5        of the spe thread that issued it. */
6     printf("Hello Cell (0x%llx)\n", id);
7     return 0;
8 }
```

Why do we need an additional `main()`?



Charting the landscape: Processors



- Typical of GPUs, DSPs, and SPUs
- They are more limited in power, and often do not support all features of mainstream languages (e.g., exception handling)
- Best at running highly-parallelizable code

Memory

Processors

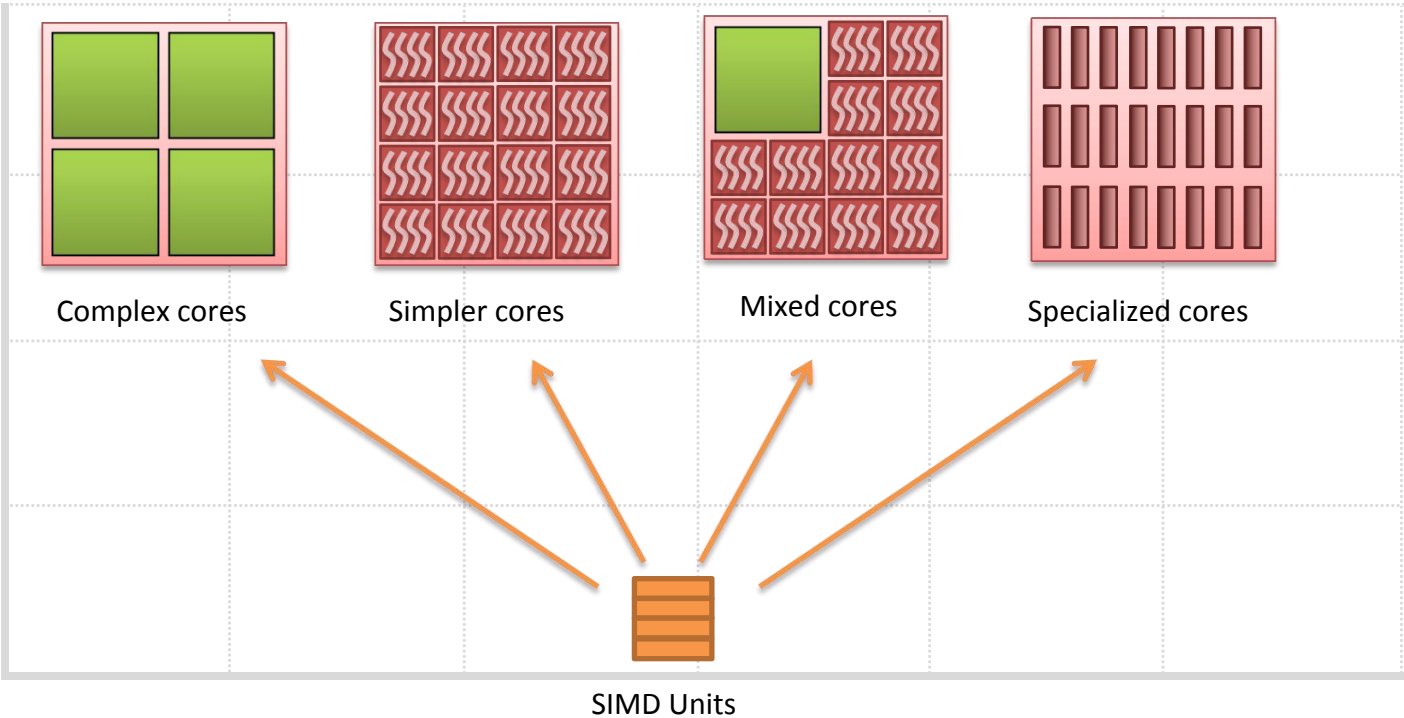


Graphics Processing Unit (GPU)

- Born specifically for graphics, then re-adapted for scientific computation
- The same operation is performed on a large set of objects (points, lines, triangles)
- Large number of ALUs (~ 100)
- Large number of registers ($\sim 32\text{k}$)
- Large bandwidth



Charting the landscape: Processors



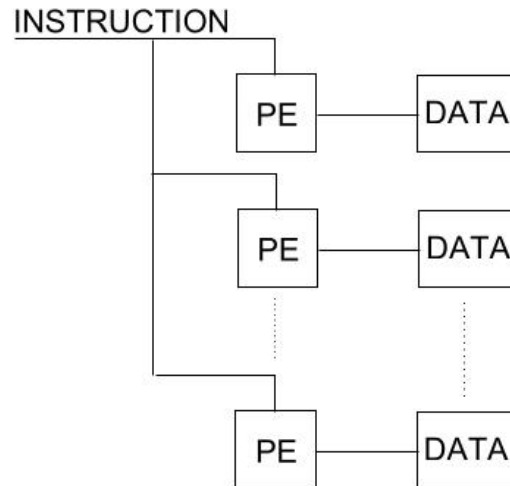
Processors

Memory



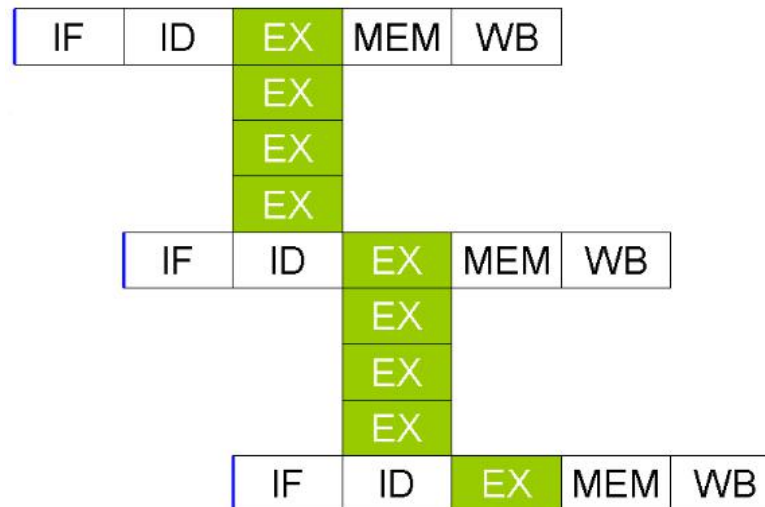
SIMD—Single Instruction Stream, Multiple Data Stream

- One operation executed on a set of data (e.g., matrix operations)
- Data-level Parallelism
- Synchronous operation



Vector Processor (or Array Processor)

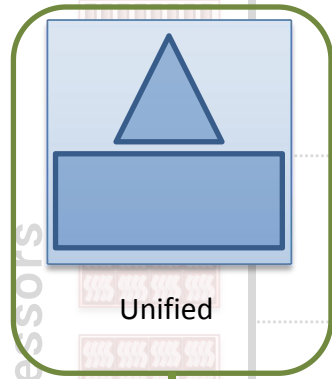
- Vector registers
- Vectorized and pipelined functional units
- Vector instructions
- Interleaved memory
- Strided memory access and hardware scatter/gather



Charting the landscape: Processors



Charting the landscape: Memory



Unified

- The “traditional” memory taught in engineering courses, used by all computers until 2000's
- A single address space
- A single memory hierarchy
- Any inexperienced programmer should be able to use it effectively
- There is the cache anyhow: notion of *locality* and *access order*

Memory



Charting the landscape: Memory



- Still a single chunk or RAM
- Multiple cache hierarchies are introduced
- We still enjoy a single address space, and have increased performance (per-core caches)
- New “performance effects”:
 - *locality* is more important (L2 cache is shared among subgroups of cores)
 - *layout matters*: the *false cache-sharing* performance anomaly

Memory



Cache Coherence (CC)

- CC defines the correct behaviour of caches, *regardless of how they are employed by the rest of the system*
- Typically programmers don't see caches, but caches are usually part of shared-memory subsystems

t	Core C1	Core C2	C1 Cache	C2 Cache
0	load r1, A		A: 42	
1		load r1, A	A: 42	A: 42
2	add r1, r1, \$1 store r1, A		A: 43	A: ??

- What is the value of A in C2?



Strong Coherence

- Most intuitive notion of CC is that cores are *cache-oblivious*:
 - All cores see at any time the same data for a particular memory address, *as they should have if there were no caches in the system*
- Alternative definition:
 - All memory read/write operations to a single location A (coming from all cores) must be executed in a *total order* that respects the order in which each core commits its own memory operations



Strong Coherence

- A sufficient condition for strong coherence is jointly given by implementing two *invariants*:
 1. Single-Writer/Multiple-Readers (SWMR)
 - For any memory location A, at any given epoch, either a single core may read and write to A or some number of cores may only read A
 2. Data-Value (DV)
 - The value of a memory location at the start of an epoch is the same as its value at the end of its latest read-write epoch



CC Protocols

- A CC protocol is a distributed algorithm in a message-passing distributed system model
- It serves two main kinds of memory requests
 - $Load(A)$ to read the value of memory location A
 - $Store(A, v)$ to write the value v into memory location A
- It involves two main kinds of actors
 - Cache controllers (i.e., L1, L2, ..., LLC)
 - Memory controllers
- It enforces a given notion of coherence
 - Strong, weak, no coherence

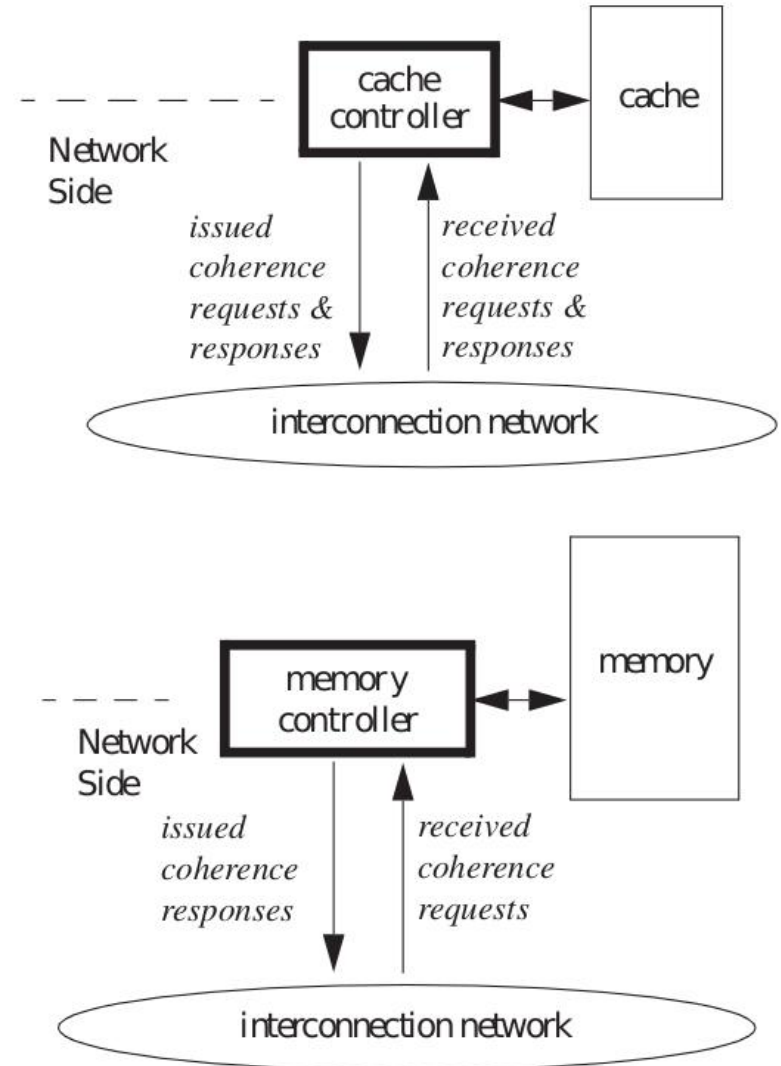
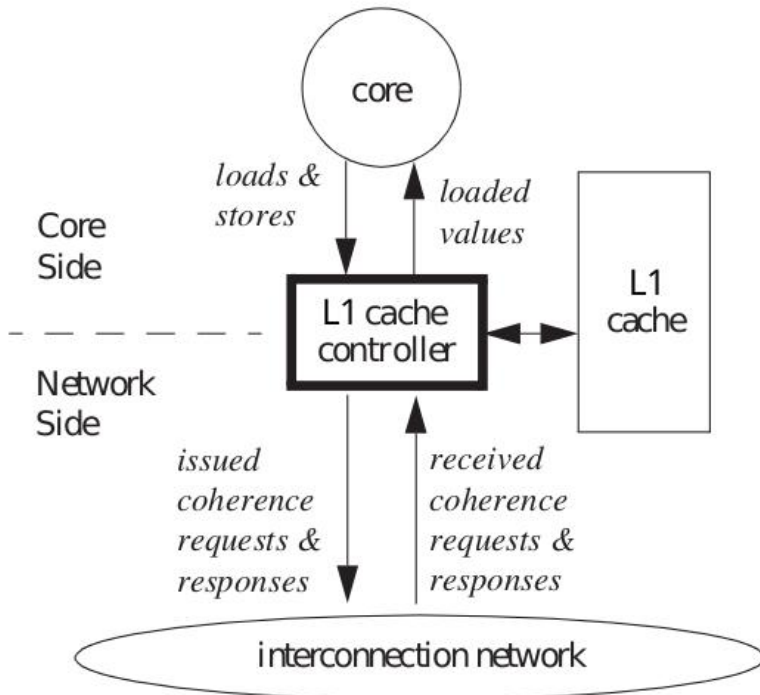


Coherency Transactions

- A memory request may translate into some *coherency transactions* and produce the exchange of multiple *coherence messages*
- There are two main kinds of coherency transactions:
 - *Get*: Load a cache block b into cache line l
 - *Put*: Evict a cache block b out of cache line l



Cache and Memory Controllers



Finite-State Machines

- Cache controllers manipulate local finite-state machines (FSMs)
- A single FSM describes the state of a copy of a block (not the block itself)
- States:
 - Stable states, observed at the beginning/end of a transaction
 - Transient states, observed in the midst of a transaction
- Events:
 - Remote events, representing the reception of a coherency message
 - Local events, issued by the parent cache controller
- Actions:
 - Remote action, producing the sending of a coherency message
 - Local actions, only visible to the parent cache controller



Families of Coherence Protocols

- **Invalidate protocols:**

- When a core writes to a block, all other copies are invalidated
- Only the writer has an up-to-date copy of the block
- Trades latency for bandwidth

- **Update protocols:**

- When a core writes to a block, it updates all other copies
- All cores have an up-to-date copy of the block
- Trades bandwidth for latency

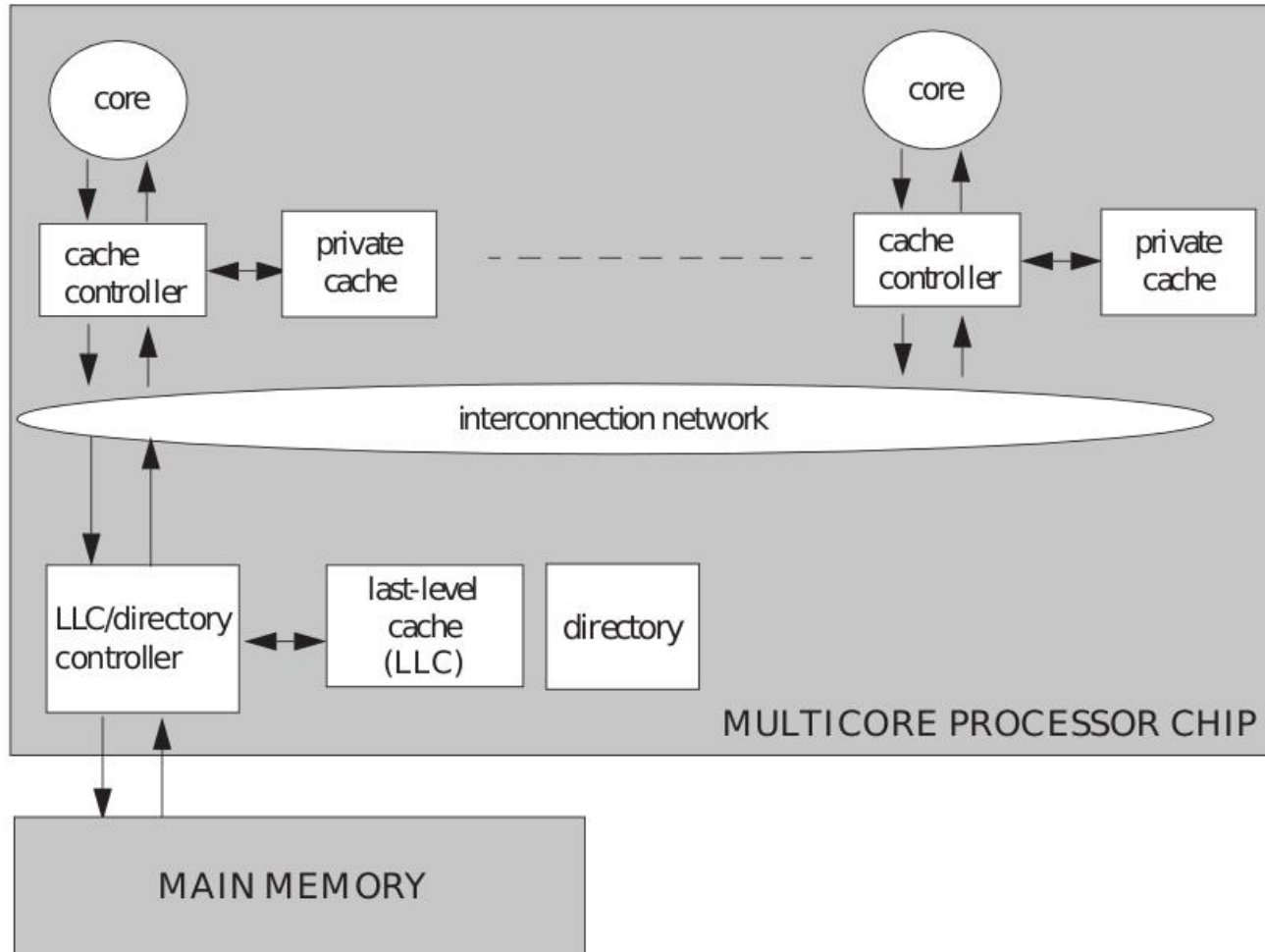


Families of Coherence Protocols

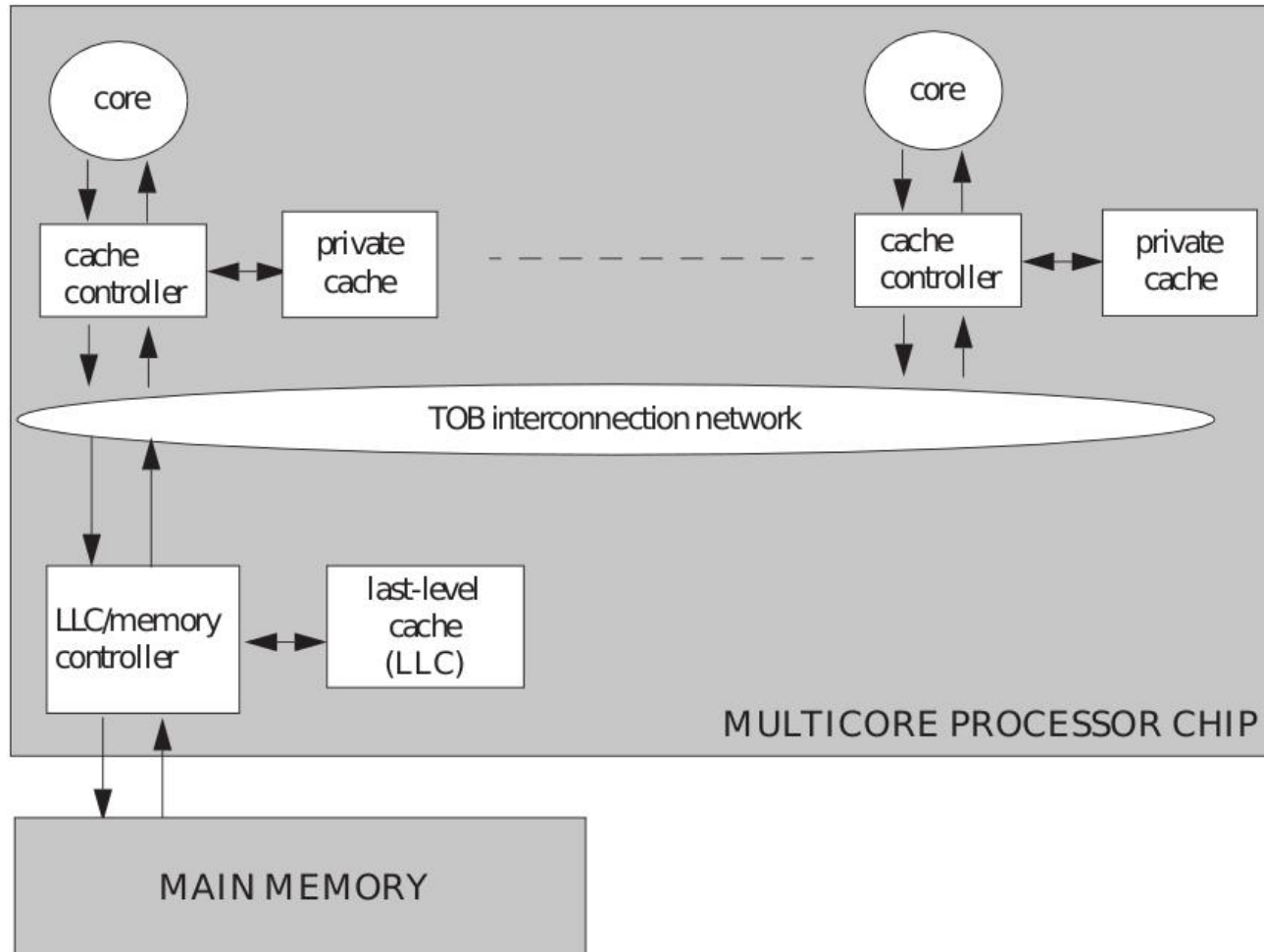
- **Snooping Cache:**
 - Coherence requests for a block are broadcast to all controllers
 - Require an interconnection layer which can total-order requests
 - Arbitration on the bus is the serialization point of requests
 - Fast, but not scalable
- **Directory Based:**
 - Coherence requests for a block are unicast to a directory
 - The directory forwards each request to the appropriate core
 - Require no assumptions on the interconnection layer
 - Arbitration at the directory is the serialization point of requests
 - Scalable, but not fast



Directory System Model



Snooping-Cache System Model



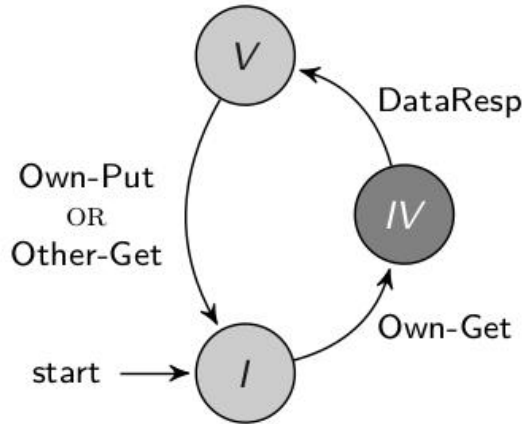
The VI Protocol

- Only one cache controller can read and/or write the block in any epoch
- Supported transactions:
 - Get: to request a block in read-write mode from the LLC controller
 - Put: to write the block's data back to the LLC controller
- List of events:
 - Own-Get: Get transaction issued from local cache controller
 - Other-Get: Get transaction issued from remote cache controller
 - Any-Get: Get transaction issued from any controller
 - Own-Put: Put transaction issued from local cache controller
 - Other-Put: Put transaction issued from remote cache controller
 - Any-Put: Put transaction issued from any controller
 - DataResp: the block's data has been successfully received



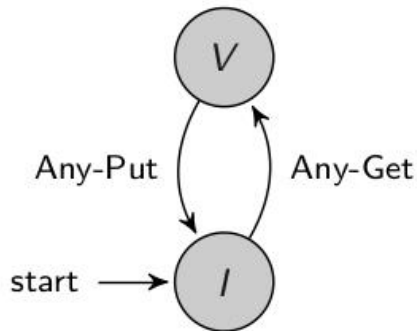
The VI Protocol

Cache controllers



V	Valid read-write copy
IV	Waiting for an up-to-date copy (transient state)
I	Invalid copy

LLC/Memory controller



V	One valid copy in L1
I	No valid copies in L1



The VI Protocol

- It has an implicit notion of *dirtiness* of a block
 - When in state V , the L1 controller can either read-write or just read the block (can't distinguish between the two usages)
- It has an implicit notion of *exclusiveness* for a block
 - When in state V , the L1 controller has exclusive access to that block (no one else has a valid copy)
- It has an implicit notion of ownership of a block
 - When in state V , the L1 controller is responsible for transmitting the updated copy to any other controller requesting it
 - In all other states, the LLC is responsible for the data transfer
- This protocol has minimal space overhead (only a few states), but it is quite inefficient—why?



What a CC Protocol should offer

- We are interested in capturing more aspects of a cache block
 - *Validity*: A valid block has the most up-to-date value for this block. The block can be read, but can be written only if it is exclusive.
 - *Dirtiness*: A block is dirty if its value is the most up-to-date value, and it differs from the one stored in the LLC/Memory.
 - *Exclusivity*: A cache block is exclusive if it is the only privately cached copy of that block in the system (except for the LLC/Memory).
 - *Ownership*: A cache controller is the owner of the block if it is responsible for responding to coherence requests for that block.
- In principle, the more properties are captured, the more aggressive is the optimization (and the space overhead!)

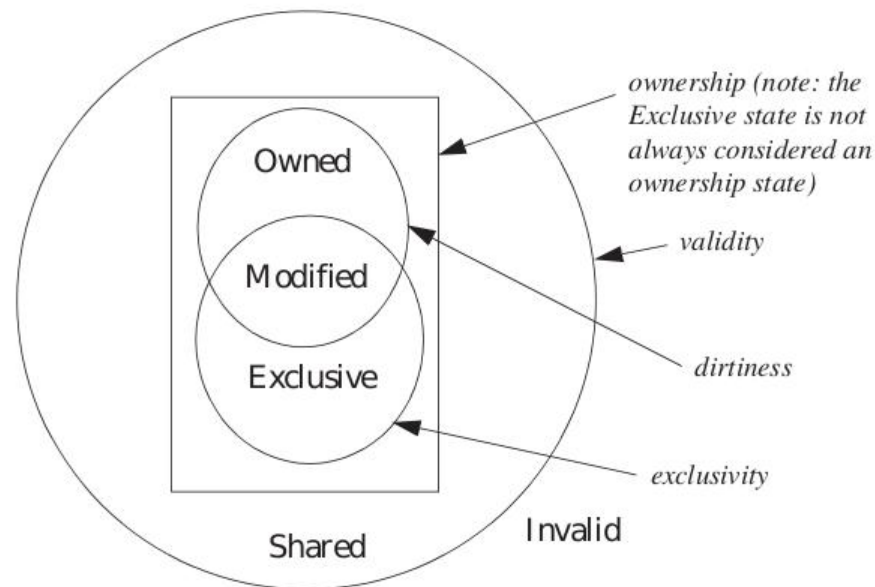


MOESI Stable States

- **Modified (M):** The block is valid, exclusive, owned and potentially dirty. It can be read or written. The cache has the only valid copy of the block.
- **Owned (O):** The block is valid, owned, and potentially dirty, but not exclusive. It can be only read, and the cache must respond to block requests.
- **Exclusive (E):** The block is valid, exclusive and clean. It can be only read. No other caches have a valid copy of the block. The LLC/Memory block is up-to-date.
- **Shared (S):** The block is valid but not exclusive, not dirty, and not owned. It can be only read. Other caches may have valid or read-only copies of the block.
- **Invalid (I):** The block is invalid. The cache either does not contain the block, or the block is potentially stale. It cannot be read nor written.



MOESI Stable States



- Many protocols spare one bit and drop the Owned state (MESI)
- Simpler protocols drop the Exclusive state (MSI)



CC and Write-Through Caches

- Stores immediately propagate to the LLC
 - States M and S collapse into V (no dirty copies)
 - Eviction only requires a transition from V to I (no data transfer)
- Write-through requires more bandwidth and power to write data



CC and False Cache Sharing

- This problem arises whenever two cores access different data items that lie on the same cache line (e.g., 64B–256B granularity)
- It produces an invalidation although accessed data items are different

```
1 struct foo {  
2     int x;  
3     int y;  
4 };  
5  
6 struct foo f;
```

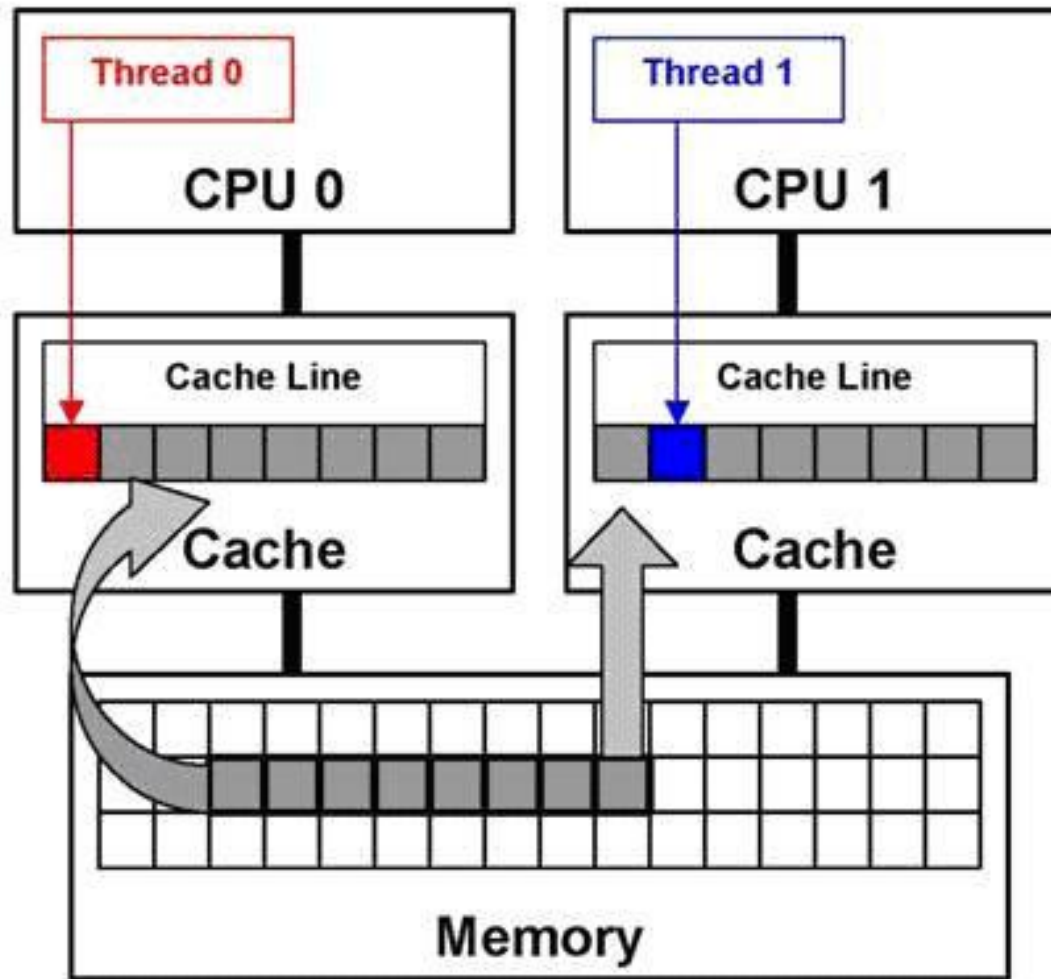
```
1 void inc_x(void)  
2 {  
3     int i;  
4     for(i = 0; i <  
5         1000000; i++)  
6         f.x++;  
7 }
```

```
1 int sum_y(void) {  
2     int s = 0;  
3     int i;  
4     for (i = 0; i <  
5         1000000; i++)  
6         s += f.y;  
7     return s;  
8 }
```

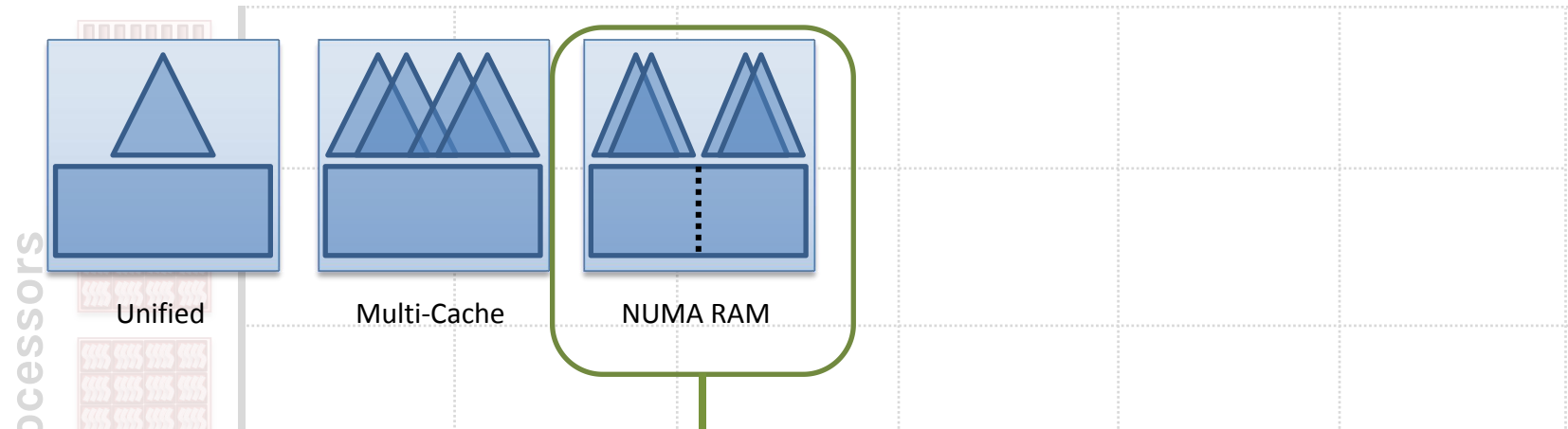
- Can be solved using sub-block coherence or speculation
- Better if prevented by good programming practices



The False Cache Sharing Problem



Charting the landscape: Memory



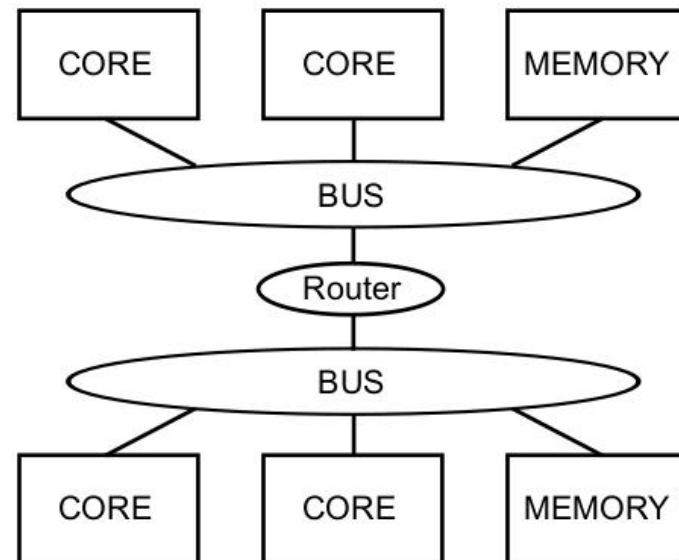
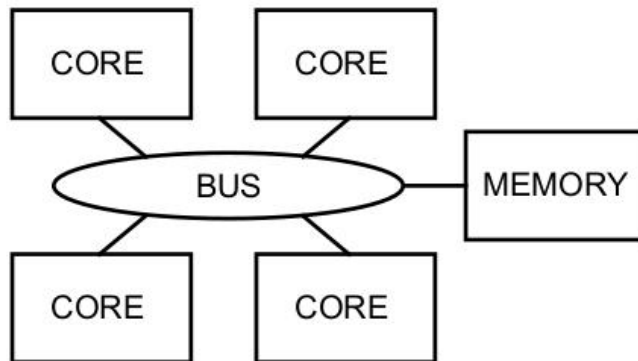
- We have multiple chunks of RAM, yet still a single address space
- The *interconnect* plays an important role: some memory is closer to some CPU, farther to others
- Conscious programmers should care about *copying*: having the possibility to point a byte could become expensive performance-wise

Memory



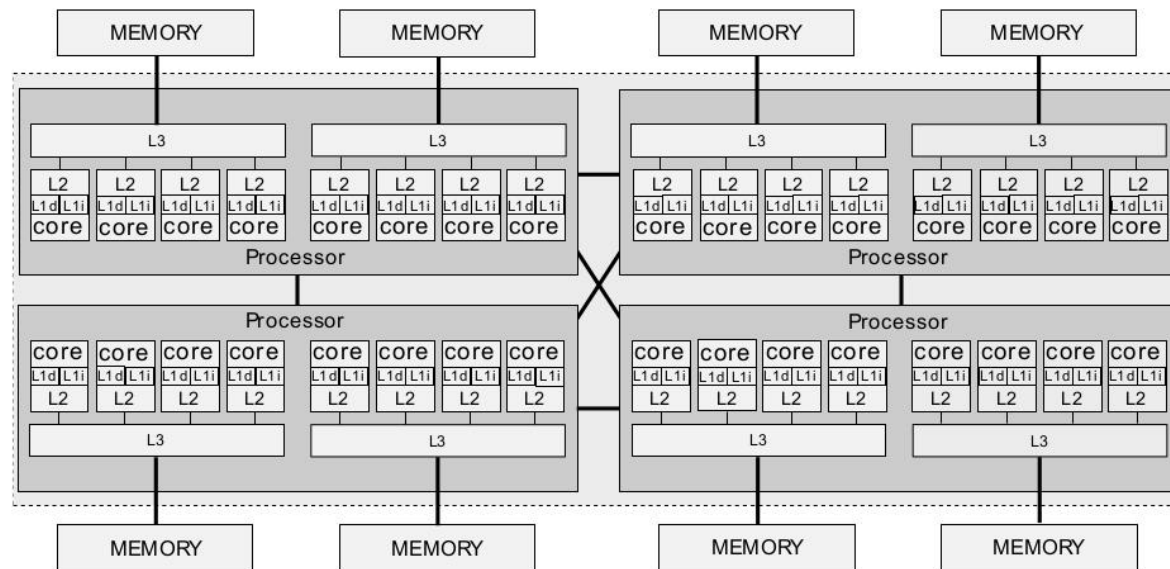
UMA vs NUMA

- In Symmetric Multiprocessing (SMP) Systems, a single memory controller is shared among all CPUs (Uniform Memory Access—UMA)
- To scale more, Non-Uniform Memory Architectures (NUMA) implement multiple buses and memory controllers



Non-Uniform Memory Access

- Each CPU has its own local memory which is accessed faster
- Shared memory is the union of local memories
- The latency to access remote memory depends on the 'distance'

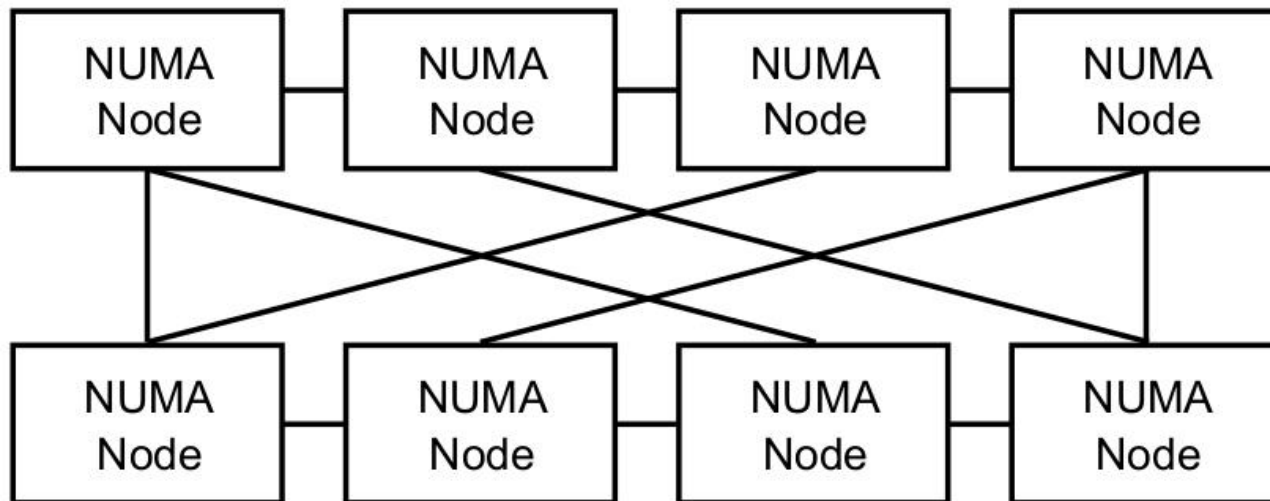


[NUMA organization with 4 AMD Opteron 6128 (2010)]



Non-Uniform Memory Access

- A processor (made of multiple cores) and the memory local to it form a *NUMA node*
- There are commodity systems which are not fully meshed: remote nodes can be only accessed with multiple hops
- The effect of a hop on commodity systems has been shown to produce a performance degradation of even 100%—but it can be even higher with increased load on the interconnect



NUMA Policies

- *Local* (first touch): allocation happens on the node the process is currently running on. Local allocation is the default policy
- *Preferred*: a set of nodes is specified. Allocation is first tried on these nodes. If memory is not available, the next closest node is selected
- *Bind*: similarly to the preferred policy, a set of nodes can be specified. If no memory is available, the allocation fails
- *Interleaved*: an allocation can span multiple nodes. Pages are allocated in a round-robin fashion across several specified nodes.
- Policies can be specified for processes, threads, or particular regions of virtual memory



NUMA System Calls (Linux)

- **Require `#include<numaif.h>` and must be linked with `-lnuma`**
- `int set_mempolicy(int mode, unsigned long *nodemask, unsigned long maxnode)`
 - `mode` is one of `MPOL_DEFAULT`, `MPOL_BIND`, `MPOL_INTERLEAVE`, `MPOL_PREFERRED`
 - `nodemask` is a bitmask specifying what nodes are affected by the policy
 - `maxnode` tells what is the most significant bit in `nodemask` which is valid
- `int get_mempolicy(int *mode, unsigned long *nodemask, unsigned long maxnode, unsigned long addr, unsigned long flags)`
 - If `flags` is 0, then information about the calling process's default policy is returned in `nodemask` and `mode`
 - If `flags` is `MPOL_F_MEMS_ALLOWED`, `mode` is ignored and `nodemask` is set accordingly all available NUMA nodes
 - If `flags` is `MPOL_F_ADDR`, the policy governing memory at `addr` is returned
- `int mbind(void *addr, unsigned long len, int mode, unsigned long *nodemask, unsigned long maxnode, unsigned flags)`
 - Can be used to set a policy for a memory region defined by `addr` and `len`



NUMA Page Migration (Linux)

- The kernel itself does not perform automatic memory migration of pages that are not allocated optimally
- `int numa_migrate_pages(int pid, struct bitmask *fromnodes, struct bitmask *tonodes)`
 - This system call can be used to migrate *all* pages that a certain process allocated on a specific set of nodes to a different set of nodes
- `long move_pages(int pid, unsigned long count, void **pages, const int *nodes, int *status, int flags)`
 - Allows to move specific pages to any specified node
 - The system call is synchronous
 - `status` allows to check the outcome of the move operation



How to Move Pages

```
1 static char *pages[TOTAL_MEMORY / PAGE_SIZE];
2 static int nodes[TOTAL_MEMORY / PAGE_SIZE];
3 static int status[TOTAL_MEMORY / PAGE_SIZE];
4
5 void do_move(int pid, char *buffer, unsigned target_node) {
6     int count;
7     int i;
8     char *segment_addr = buffer;
9
10    count = TOTAL_MEMORY / PAGE_SIZE;
11
12    for (i = 0; i < count; i++){
13        pages[i] = segment_addr + i * PAGE_SIZE;
14        nodes[i] = target_node;
15        status[i] = target_node+1;
16    }
17
18    move_pages(0, count, (void **)pages, nodes, status, MPOL_MF_MOVE);
19 }
```



libnuma

- This library offers an abstracted interface
- It is the preferred way to interact with a NUMA-aware kernel
- Requires `#include<numa.h>` and linking with `-lnuma`
- Some symbols are exposed through `numaif.h`
- `numactl` is a command line tool to run processes with a specific NUMA policy without changing the code, or to gather information on the NUMA system



Checking for NUMA

```
1 #include <numa.h>
2
3 ...
4
5 if(numa_available() < 0) {
6     printf("Your system does not support NUMA API\n");
7     exit();
8 }
9
10 int nodes = numa_max_node();
11 int cpus = numa_num_configured_cpus();
```



Allocation and Policies

- Different memory allocation APIs for different nodes:
 - `void *numa_alloc_onnode(size_t size, int node)`
 - `void *numa_alloc_local(size_t size)`
 - `void *numa_alloc_interleaved(size_t size)`
 - `void *numa_alloc_interleaved_subset(size_t size, struct nodemask_t *nodemask)`
- The counterpart is `numa_free(void *start, size_t size)`
- How to set a process policy:

```
1 nodemask_t oldmask = numa_get_interleave_mask();
2 numa_set_interleave_mask(&numa_all_nodes);
3 /* run memory-intensive code */
4 numa_set_interleave_mask(&oldmask);
```



Dealing with Nodemasks

- `nodemask_t` defines a nodemask
- A nodemask can be cleared with `nodemask_zero(nodemask_t *)`
- A specific node can be set with `nodemask_set(nodemask_t *, int node)`, cleared with `nodemask_clear(nodemask_t *, int node)`, and checked with `nodemask_isset(nodemask_t *, int node)`
- `nodemask_equal(nodemask_t *, nodemask_t *)` compares two different nodemasks



Binding to CPUs

- Run current thread on node 1 and allocate memory on node 1:

```
1 numa_run_on_node(1);  
2 numa_set_preferred(1);
```

- Bind process CPU and memory allocation to node 1:

```
1 nodemask_t mask;  
2 nodemask_zero(&mask);  
3 nodemask_set(&mask, 1);  
4 numa_bind(&mask);
```

- Allow the thread to run on all CPUs again:

```
1 numa_run_on_node_mask(&numa_all_nodes);
```

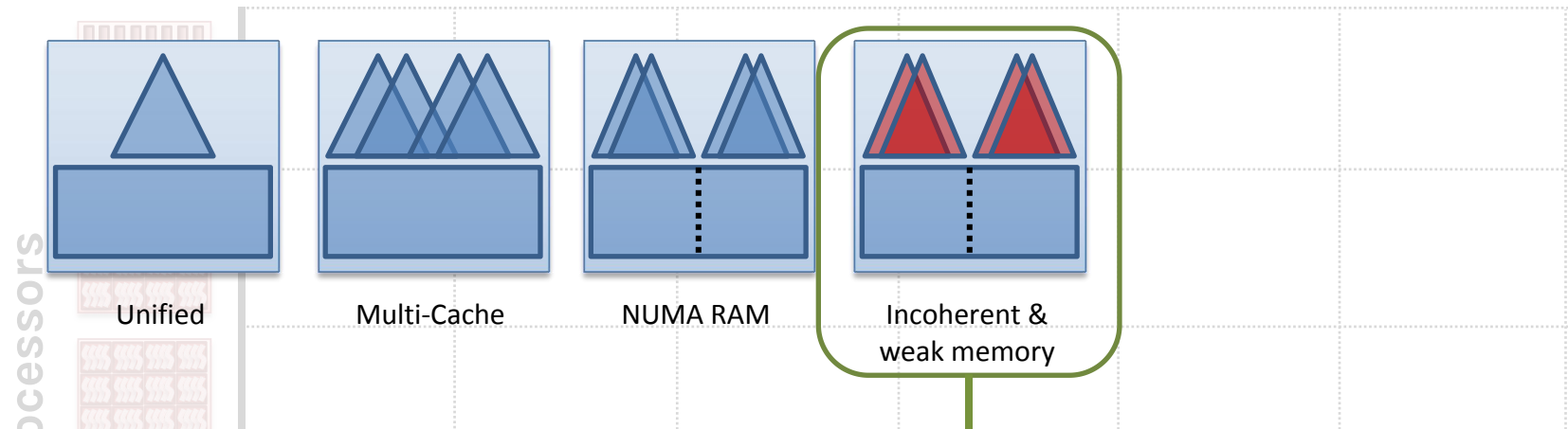


numactl

- `numactl --cpubin=0 --membind=0,1 program`
 - Run program on CPUs of node 0 and allocate memory from nodes 0 and 1
- `numactl --preferred=1 numactl --show`
 - Allocate memory preferably from node 1 and show the resulting state
- `numactl --interleave=all program`
 - Run program with memory interleaved over all available nodes
- `numactl --offset=1G --length=1G --membind=1 --file /dev/shm/A --touch`
 - Bind the second gigabyte in the tempfs file `/dev/shm/A` to node 1
- `numactl --localalloc /dev/shm/file`
 - Reset the policy for the shared memory file `/dev/shm/file`
- `numactl --hardware`
 - Print an overview of the available nodes



Charting the landscape: Memory



- The cache subsystem no longer ensures *consistency*, or ensures “reduced” consistency
 - Different CPUs could see different values at the same byte
- Programmers must take care of *synchronization* explicitly
- Many of these are performance experiments that are failing in the marketplace
- Yet, all mainstream architectures have some form of weak memory

Memory



Weaker Coherence

- Weaker forms of coherence may exist for performance purposes
 - Caches can respond faster to memory read/write requests
- The SWMR invariant might be completely dropped
 - Multiple cores might write to the same location A
 - One core might read A while another core is writing to it
- The DV invariant might hold only *eventually*
 - Stores are guaranteed to propagate to all cores in d epochs
 - Loads see the last value written only after d epochs
- The effects of weak coherency are usually visible to programmers
 - Might affect the memory consistency model (see later)



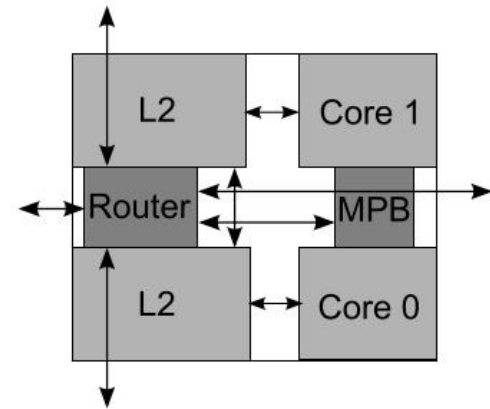
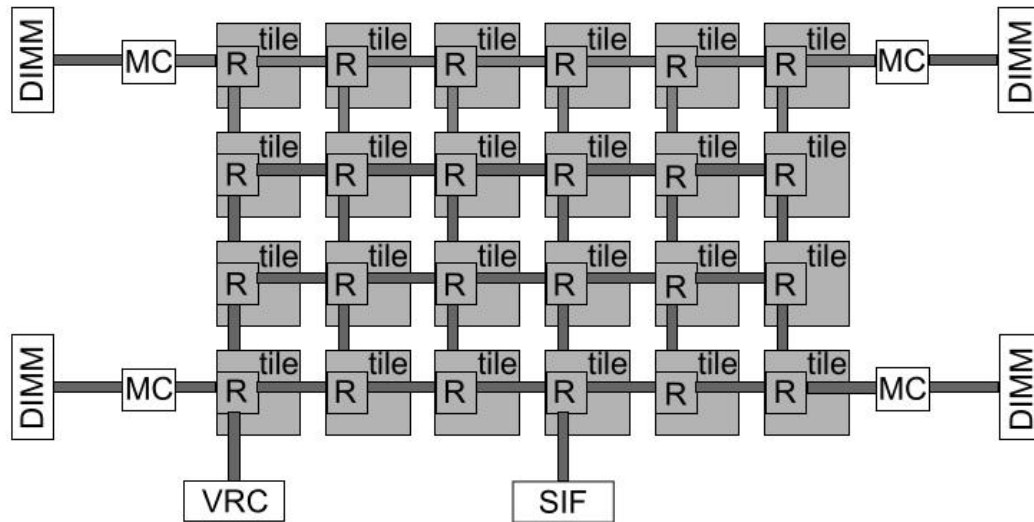
No Coherence

- The fastest cache is *non-coherent*
 - All read/write operations by all cores can occur simultaneously
 - No guarantees on the value observed by a read operation
 - No guarantees on the propagation of values from write operations
- Programmers must explicitly coordinate caches across cores
 - Explicit invocation of coherency requests via C/Assembly APIs



Intel Single-Chip Cloud Computing (SCC)

- 48 Intel cores on a single die
- Power 125W cores @ 1GHz, Mesh @ 2Ghz
- Message Passing Architecture
- No coherent shared memory
- Proof of Concept of a scalable many-core solution



Memory Consistency (MC)

- MC defines the correct behaviour of shared-memory subsystems, *regardless of how they are implemented*
- Programmers know what to expect, implementors know what to provide

Core C1	Core C2
S1: store data = NEW S2: store flag = SET	L1: load r1 = flag B1: if (r1 \neq SET) goto L1 L2: load r2 = data

- What is the value of r2?



Reordering of Memory Accesses

- Reordering occurs when two memory R/W operations:
 - Are committed by a core in order
 - Are seen by other cores in a different order
- Mainly for performance reasons
 - Out-of-order execution/retirement
 - Speculation (e.g., branch prediction)
 - Delayed/combined stores
- Four possible reorderings
 - Store-store reordering
 - Load-load reordering
 - Store-load reordering
 - Load-store reordering



An example

Core C1	Core C2
S1: $x = \text{NEW}$	S2: $y = \text{NEW}$
L1: $r1 = y$	L2: $r2 = x$

- Multiple reorderings are possible:

- $(r1, r2) = (0, \text{NEW})$ [S1, L1, S2, L2]

- $(r1, r2) = (\text{NEW}, 0)$ [S2, L2, S1, L1]

- $(r1, r2) = (\text{NEW}, \text{NEW})$ [S1, S2, L1, L2]

- $(r1, r2) = (0, 0)$ [L1, L2, S1, S2]

Allowed by most real hardware architectures (also x86!)



Program and Memory Orders

- A program order \prec_p is a per-core total order that captures the order in which each core logically executes memory operations
- A memory order \prec_m is a system-wide total order that captures the order in which memory logically serializes memory operations from all cores
- Memory consistency can be defined imposing constraints on how \prec_p and \prec_m relate to each other



Sequential Consistency

- Let $L(a)$ and $S(a)$ be a Load and a Store to address a , respectively
- A *Sequentially Consistent* execution requires that:
 - All cores insert their loads and stores into \prec_m respecting their program order regardless of whether they are to the same or different address (i.e., $a = b$ or $a \neq b$):
 - If $L(a) \prec_p L(b) \Rightarrow L(a) \prec_m L(b)$ (Load/Load)
 - If $L(a) \prec_p S(b) \Rightarrow L(a) \prec_m S(b)$ (Load/Store)
 - If $S(a) \prec_p S(b) \Rightarrow S(a) \prec_m S(b)$ (Store/Store)
 - If $S(a) \prec_p L(b) \Rightarrow S(a) \prec_m L(b)$ (Store/Load)
 - Every load gets its value from the last store before it (as seen in memory order) to the same address:
 - value of $L(a) = \text{value of } \max_{\prec_m} \{S(a) | S(a) \prec_m L(a)\}$ where \max_{\prec_m} is the latest in memory order



Sequential Consistency in Practice

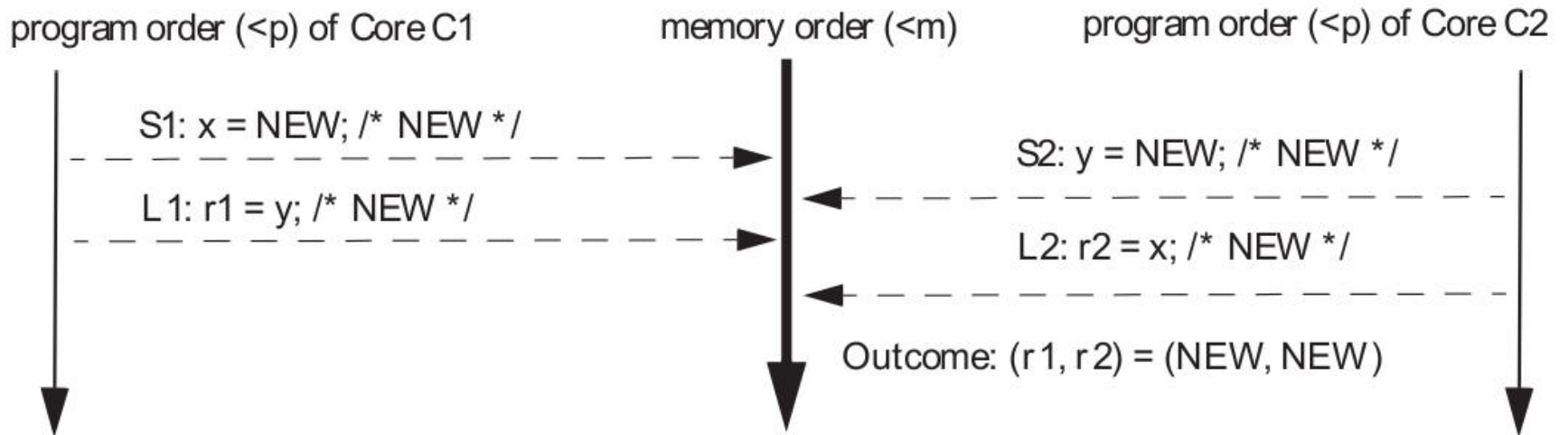
Core C1	Core C2
S1: $x = \text{NEW}$	S2: $y = \text{NEW}$
L1: $r1 = y$	L2: $r2 = x$

- We can globally reorder the execution in four different ways
- Only three of them are sequentially consistent



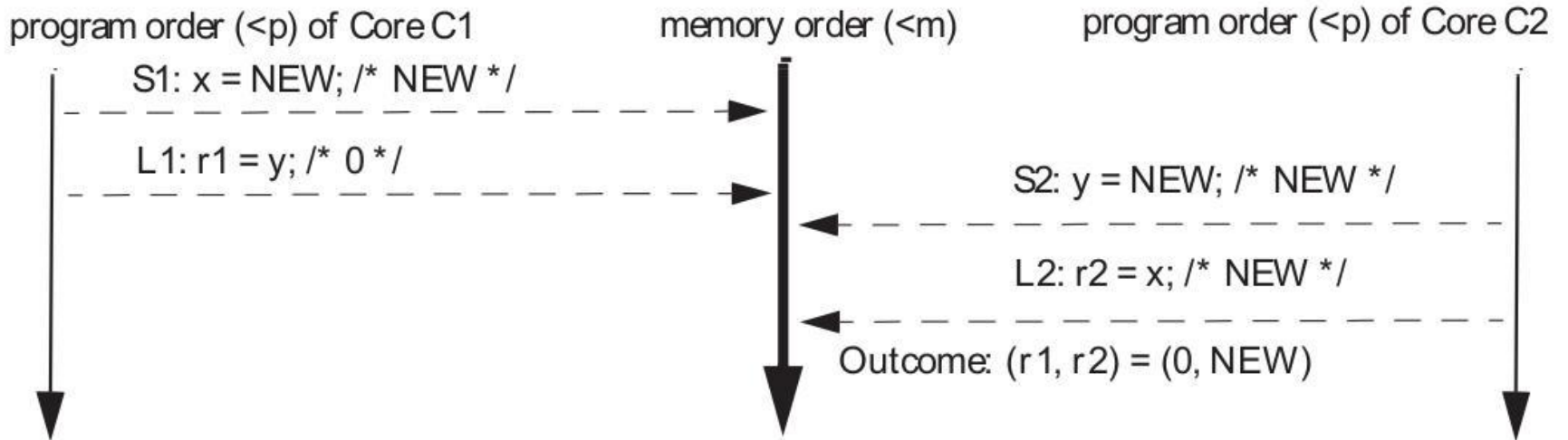
Sequential Consistency in Practice

- This is a SC execution



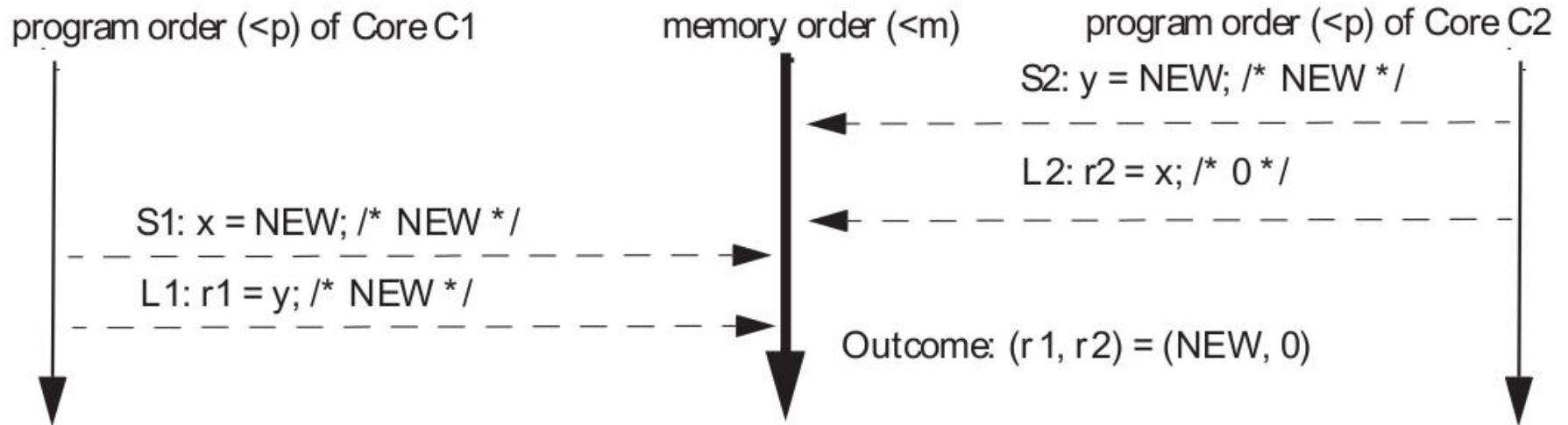
Sequential Consistency in Practice

- This is a SC execution



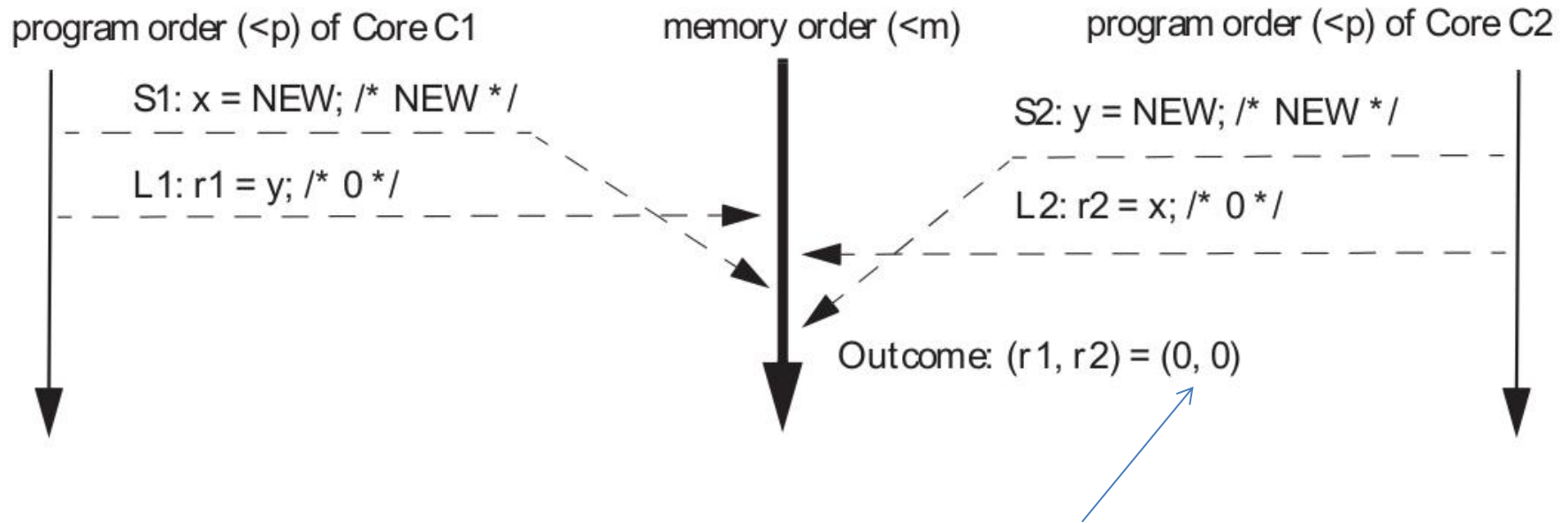
Sequential Consistency in Practice

- This is a SC execution



Sequential Consistency in Practice

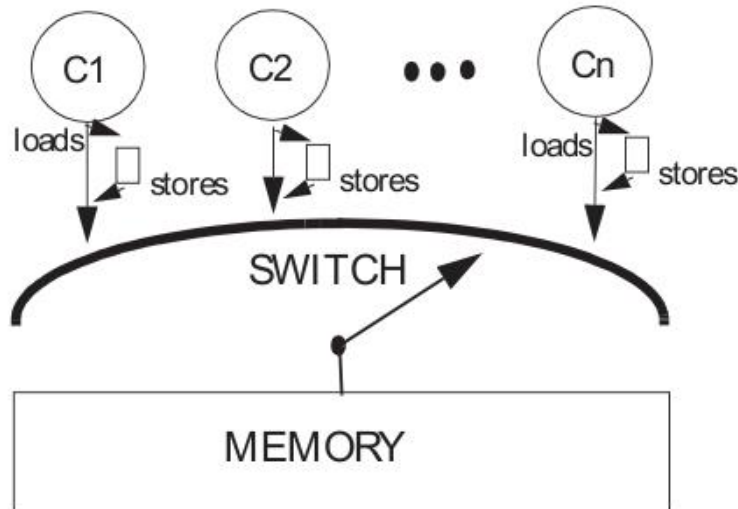
- This is not a SC execution



Why did we say that this outcome is allowed on common hardware architectures?



Weaker Consistency: Total Store Order

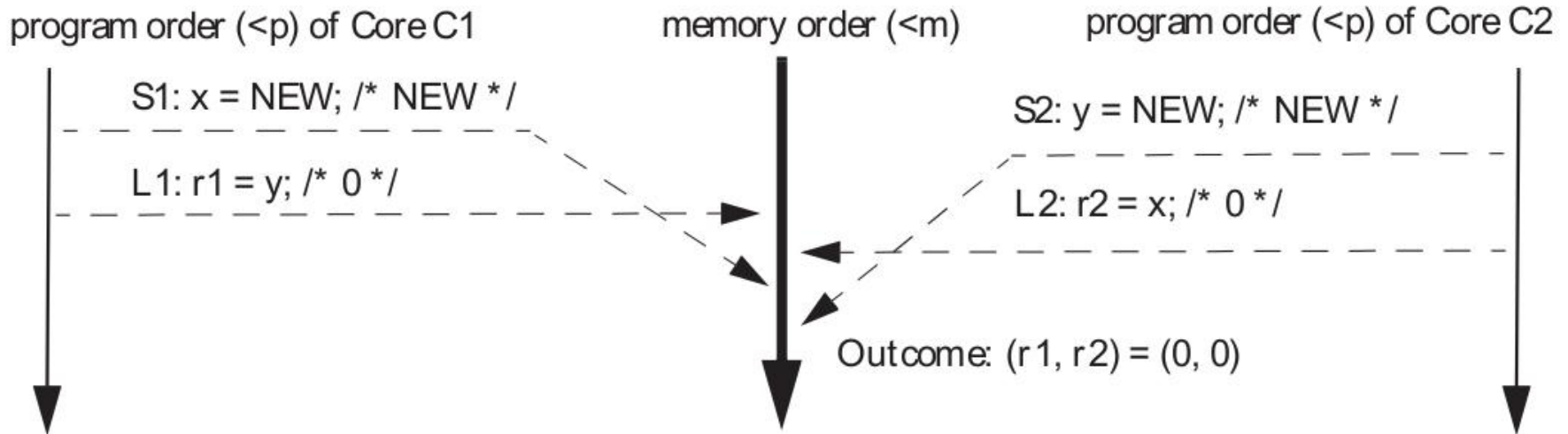


- A FIFO *store buffer* is used to hold committed stores until the memory subsystem can process it
- When a load is issued by a core, the store buffer is looked up for a matching store
 - if found, the load is served by the store buffer (*forwarding*)
 - otherwise it is served by the memory subsystem (*bypassing*)



Sequential Consistency in Practice

- This is a valid TSO execution



- A programmer might want to avoid the result $(r1, r2) = (0, 0)$



Memory Reordering in the Real World

Type	Alpha	ARMv7	POWER	SPARC			
				PSO	x86	AMD64	IA-64
LOAD/LOAD	✓	✓	✓				✓
LOAD/STORE	✓	✓	✓				✓
STORE/STORE	✓	✓	✓	✓			✓
STORE/LOAD	✓	✓	✓	✓	✓	✓	✓
ATOMIC/LOAD	✓	✓	✓				✓
ATOMIC/STORE	✓	✓	✓	✓			✓
Dependent LOADs	✓						
Incoherent I-cache	✓	✓	✓	✓	✓		✓



The Effect Seen by Programmers

```
struct foo {  
    int a;  
    int b;  
    int c;  
};  
struct foo *gp = NULL;
```

```
/* . . . */
```

```
p = malloc(sizeof(*p));  
p->a = 1;  
p->b = 2;  
p->c = 3;  
gp = p;
```

Is this always correct?



The Effect Seen by Programmers

```
struct foo {
    int a;
    int b;
    int c;
};
struct foo *gp = NULL;

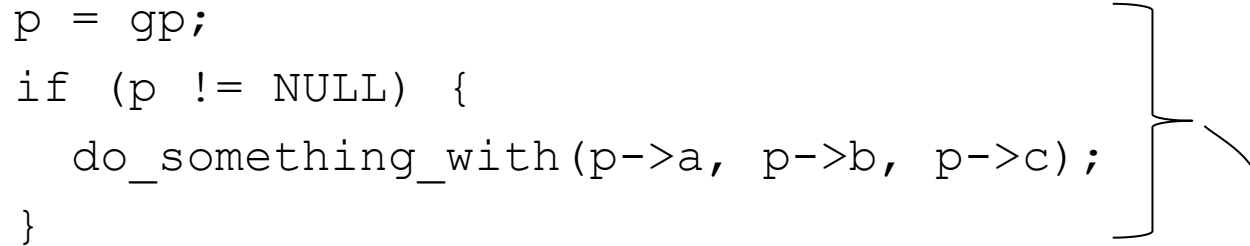
/* . . . */

p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
gp = p;
SFENCE ← "publish" the value
```



The Effect Seen by Programmers

```
p = gp;  
if (p != NULL) {  
    do_something_with(p->a, p->b, p->c);  
}
```



Is this always correct?



The Effect Seen by Programmers

```
p = gp; ←———— Memory/compiler barriers here
if (p != NULL) {
    do_something_with(p->a, p->b, p->c);
}
```



Memory Fences

- Let $X(a)$ be either a load or a store operation to a
- *Memory fences* force the memory order of load/store operations:
 - If $X(a) \prec_p \text{FENCE} \Rightarrow X(a) \prec_m \text{FENCE}$
 - If $\text{FENCE} \prec_p X(a) \Rightarrow \text{FENCE} \prec_m X(a)$
 - If $\text{FENCE} \prec_p \text{FENCE} \Rightarrow \text{FENCE} \prec_m \text{FENCE}$
- With fences it is possible to implement SC over TSO (with a significant performance penalty)

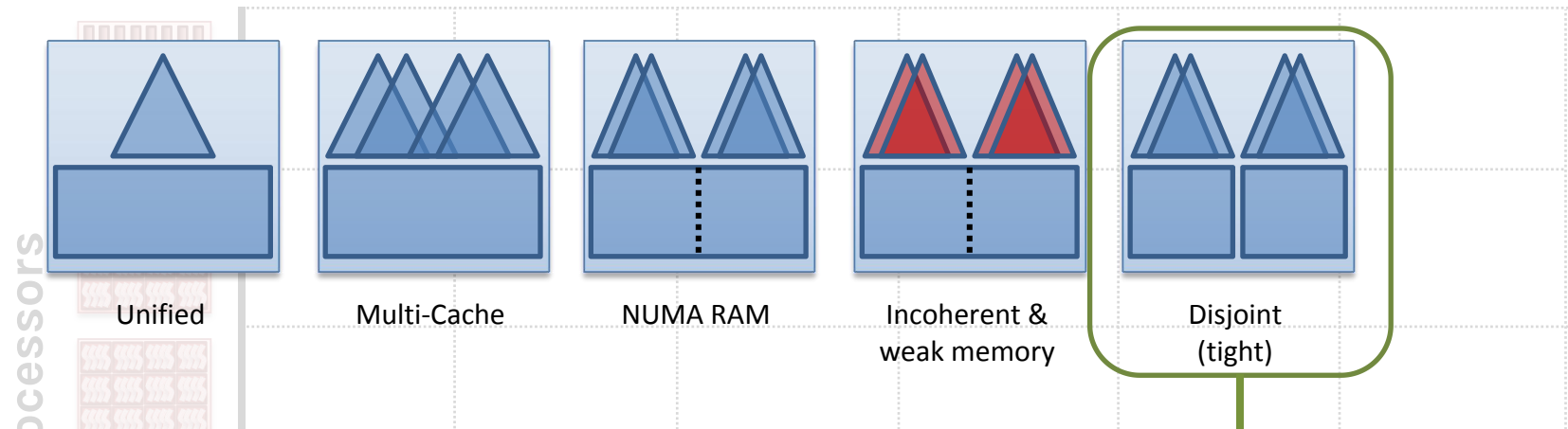


x86 Fences

- **MFENCE: Full barrier**
 - If $X(a) \prec_p \text{MFENCE} \prec_p X(b) \Rightarrow X(a) \prec_m \text{MFENCE} \prec_m X(b)$
- **SFENCE: Store/Store barrier**
 - If $S(a) \prec_p \text{SFENCE} \prec_p S(b) \Rightarrow S(a) \prec_m \text{SFENCE} \prec_m S(b)$
- **LFENCE: Load/Load and Load/Store barrier**
 - If $L(a) \prec_p \text{LFENCE} \prec_p X(b) \Rightarrow L(a) \prec_m \text{LFENCE} \prec_m X(b)$
- Both MFENCE and SFENCE drain the store buffer



Charting the landscape: Memory

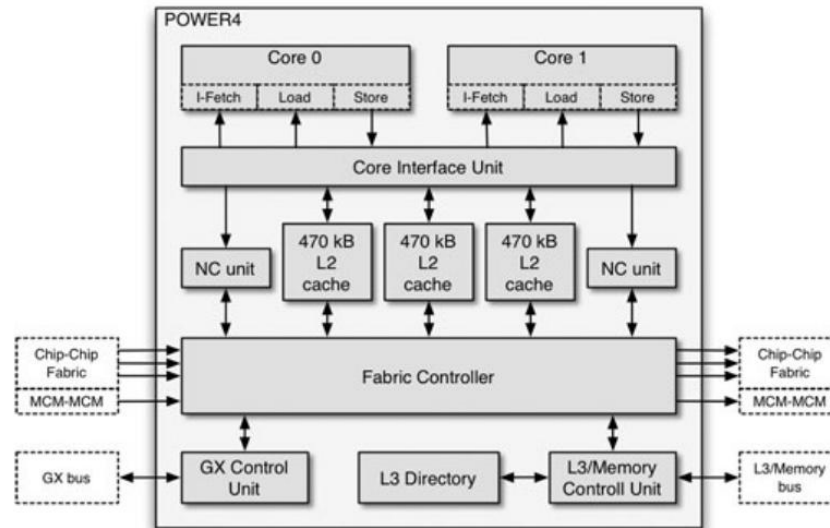


- Different cores see different memory, connected over a shared busnetwork
- Reliability is still evaluated as a single unit
- Typical of ~2010 vintage GPU systems, or in general of systems which require offloading
- Removed all precedent burdens from programmers
 - replaced with that of *copying* data

Memory



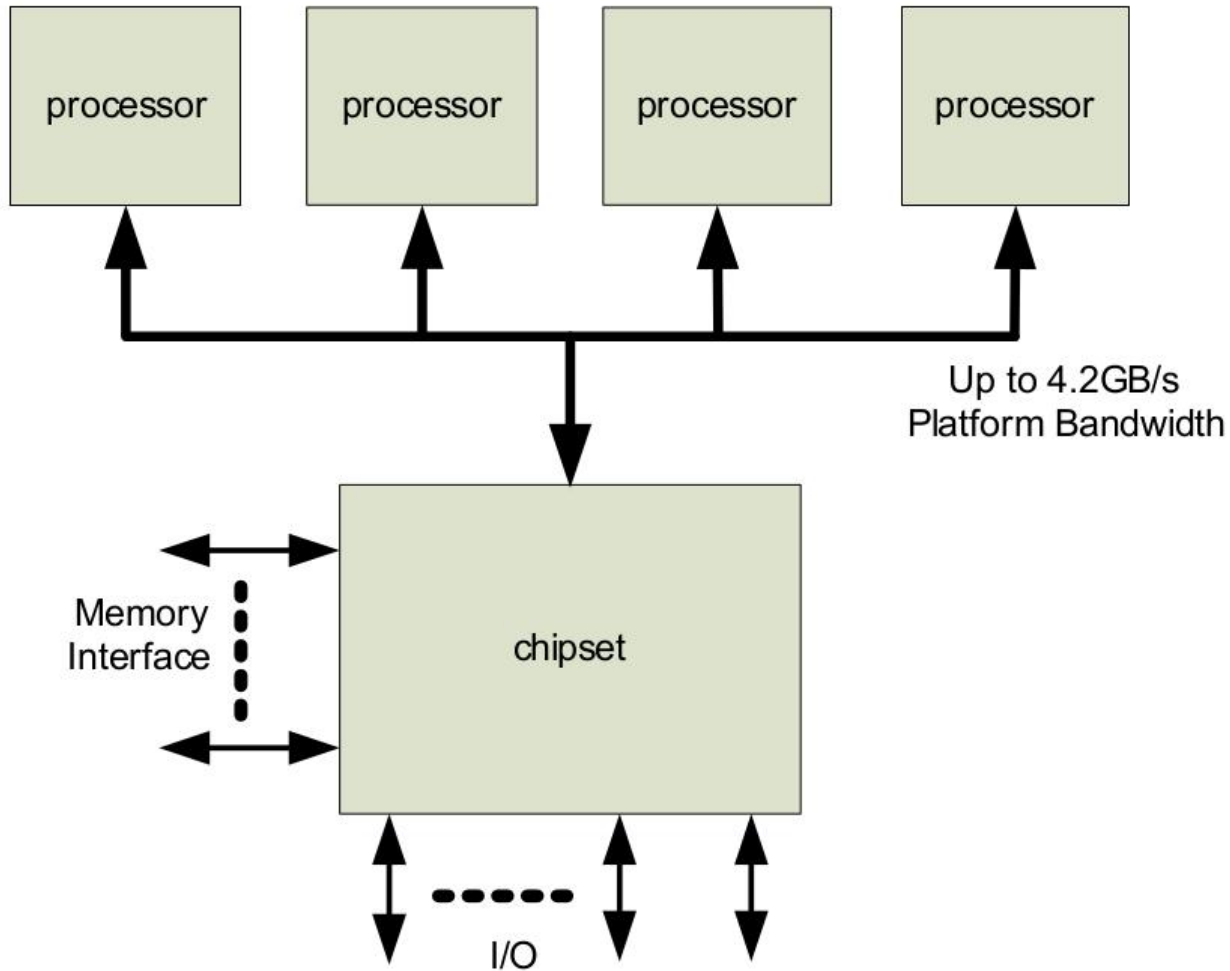
Inter-Core Connection



- What is the importance of inter-core connection?

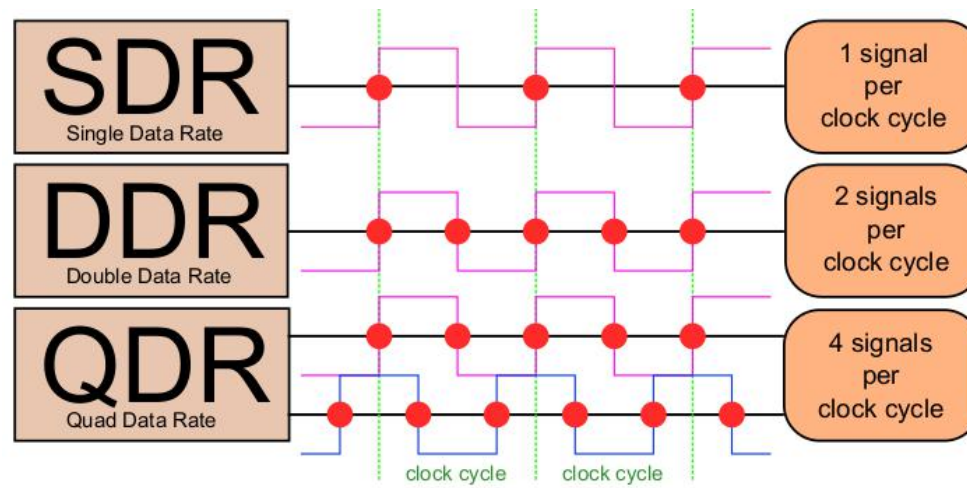


Front-Side Bus (up to 2004)

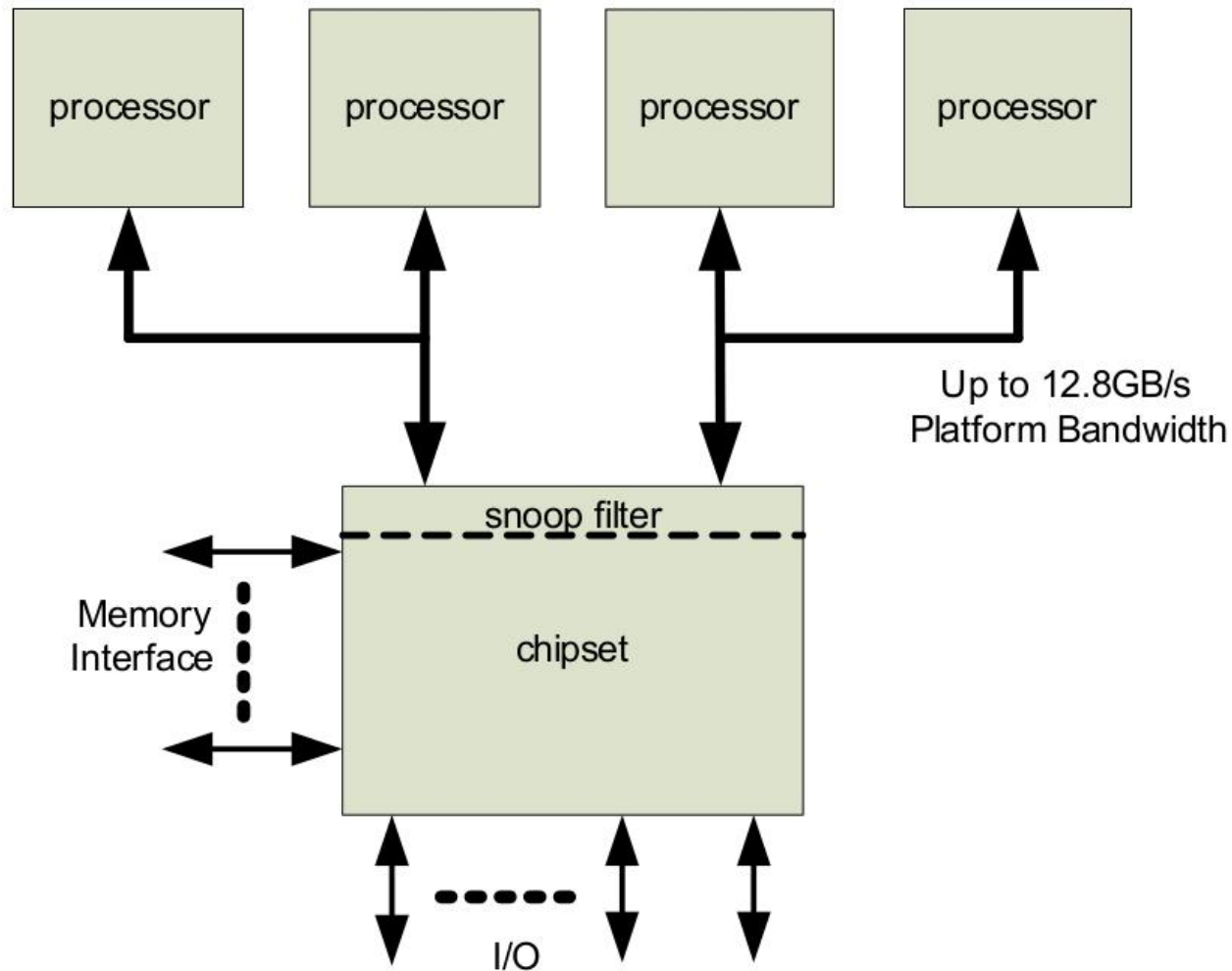


Front-Side Bus (up to 2004)

- All traffic is sent across a single shared bi-directional bus
- Common width: 64 bits, 128 bits — multiple data bytes at a time
- To increase data throughput, data has been clocked in up to 4x the bus clock
 - *double-pumped* or *quad-pumped* bus



Dual Independent Buses (2005)

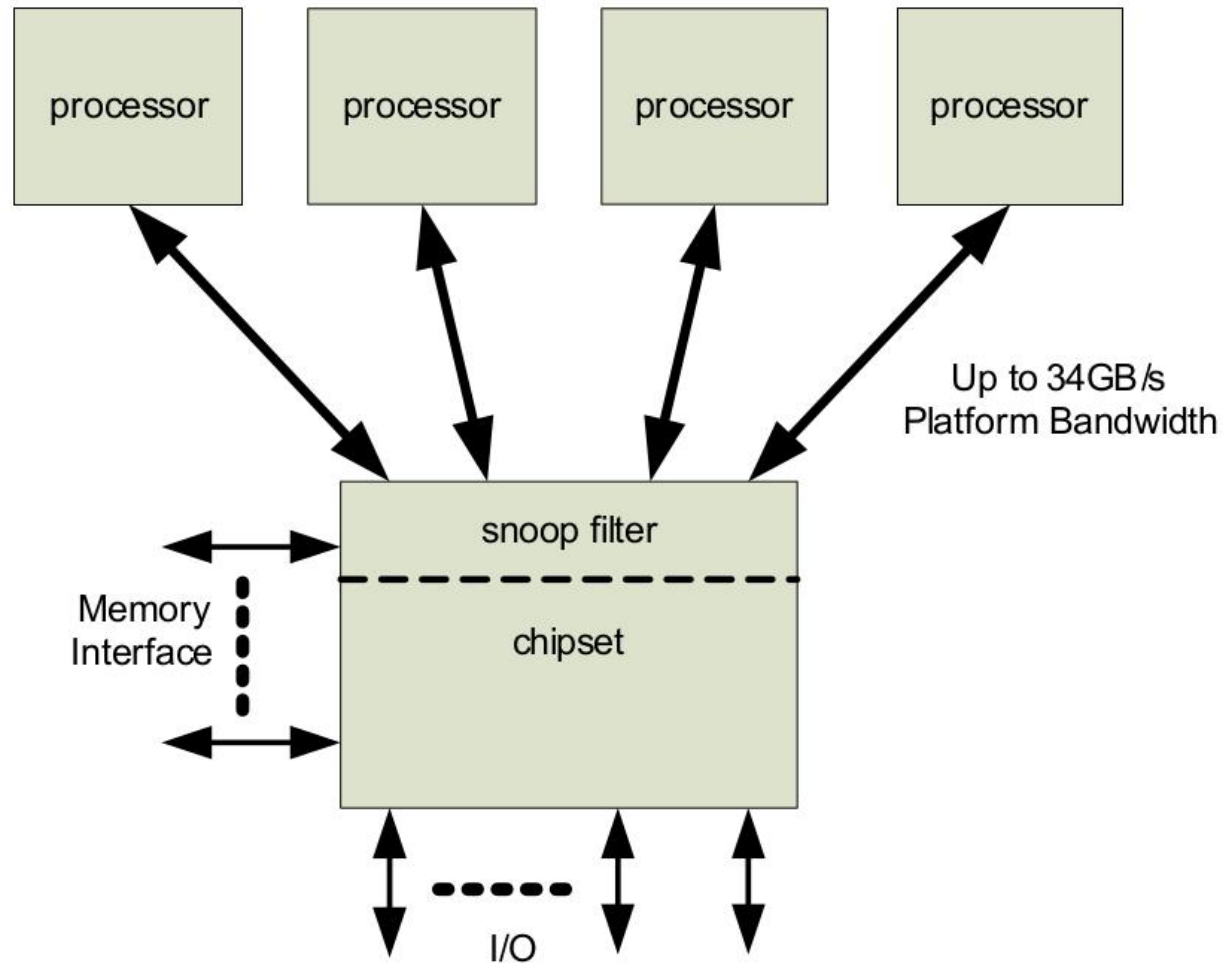


Dual Independent Buses (2005)

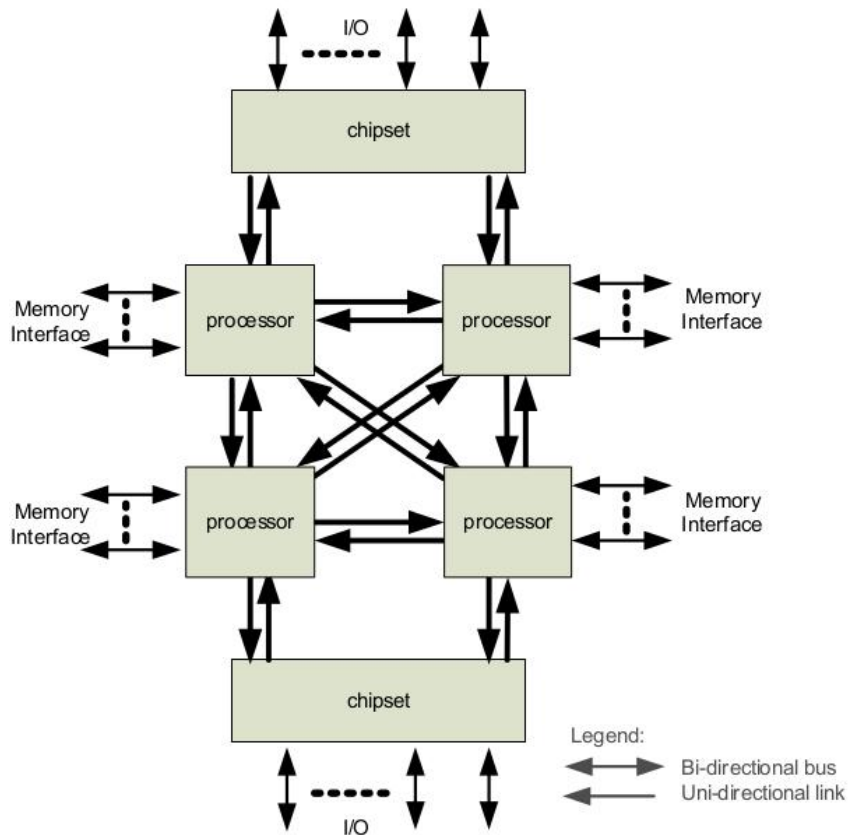
- The single bus is split into two separate buffers
- This doubles the available bandwidth, in principles
- All snoop traffic had to be broadcast on both buses
- This would reduce the effective bandwidth
 - Snoop filters are introduced in the chipset
 - They are used as a cache of snoop messages



Dedicated High-Speed Interconnects (2007)



QuickPath Interconnect (2009)

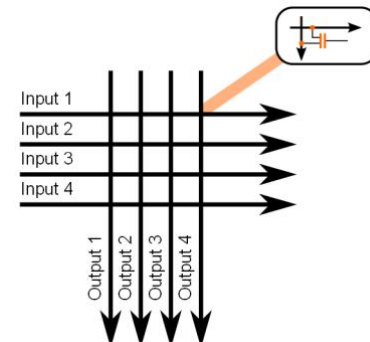
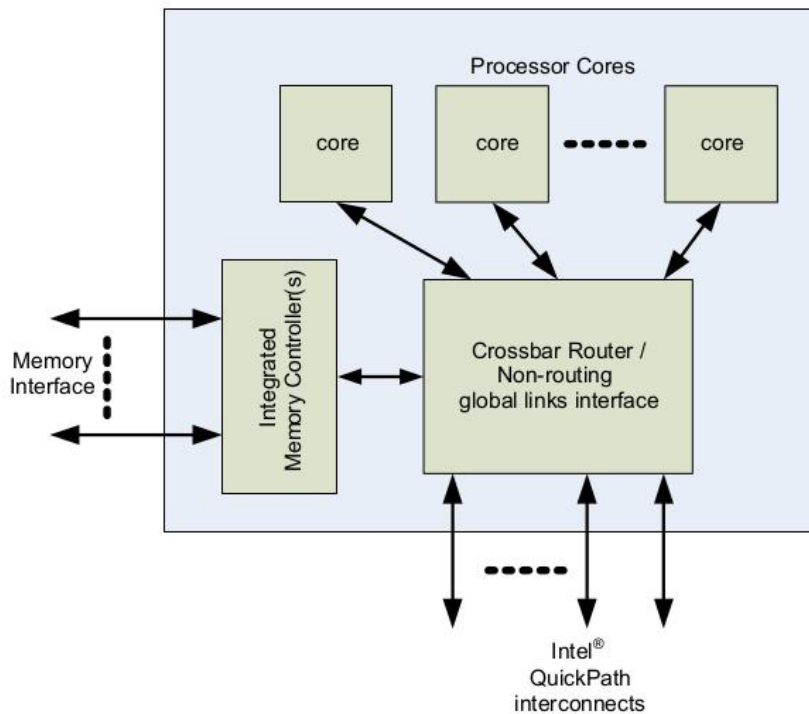


- Migration to a distributed shared memory architecture chipset
- Inter-CPU communication based on high-speed uni-directional point-to-point links
- Data can be sent across multiple lanes
- Transfers are packetized: data is broken into multiple transfers



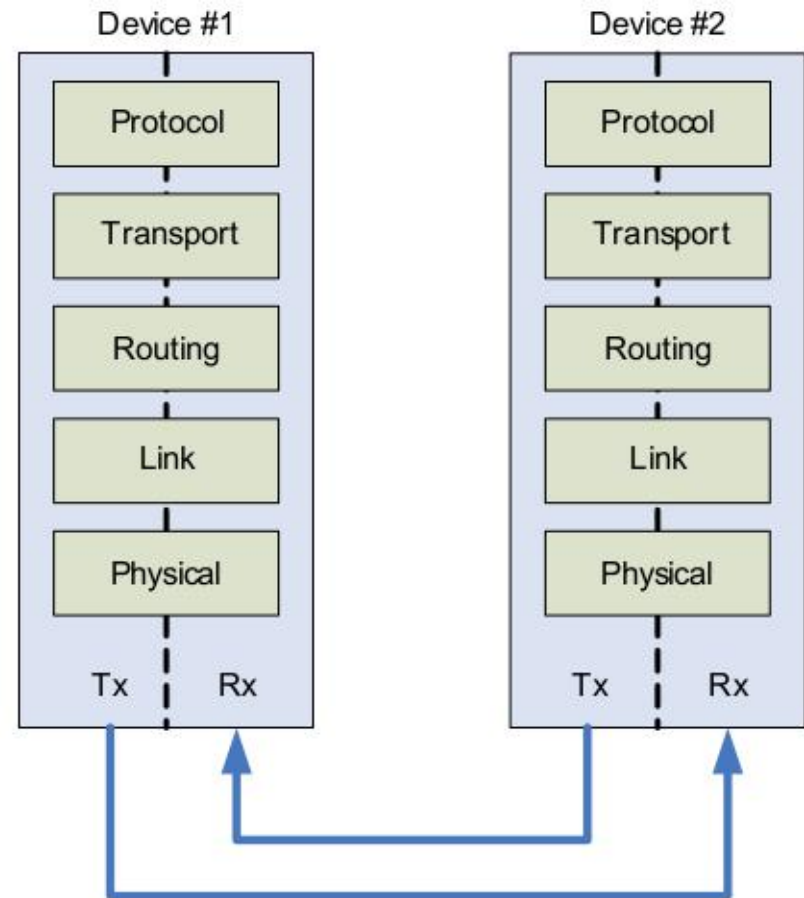
QPI and Multicores

- The connection between a core and QPI is realized using a crossbar router:

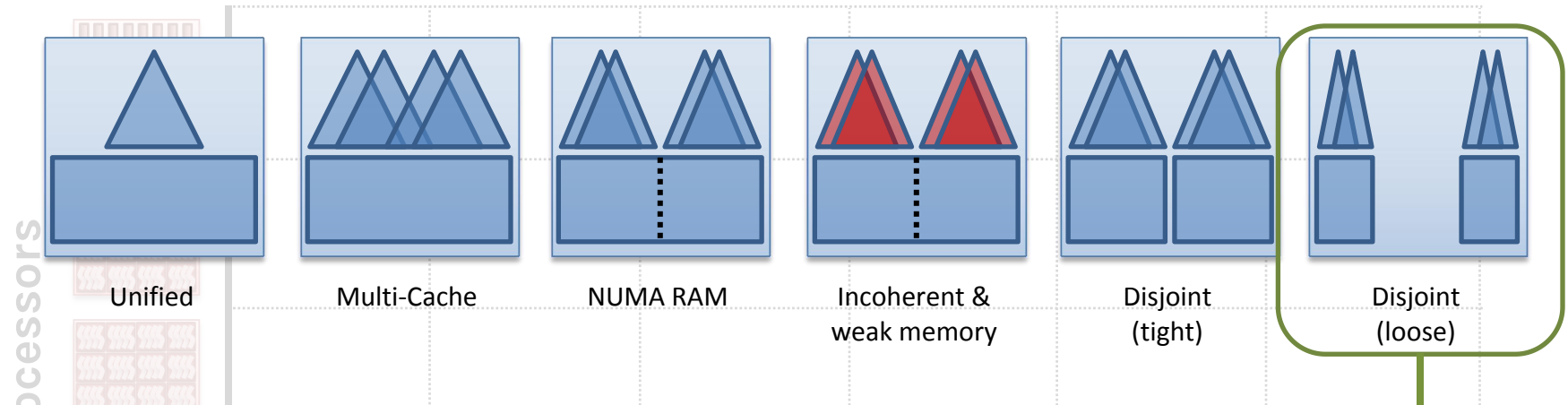


QPI Layers

- Each link is made of 20 signal pairs and a forwarded clock
- Each port has a link pair with two uni-directional link
- Traffic is supported simultaneously in both directions



Charting the landscape: Memory

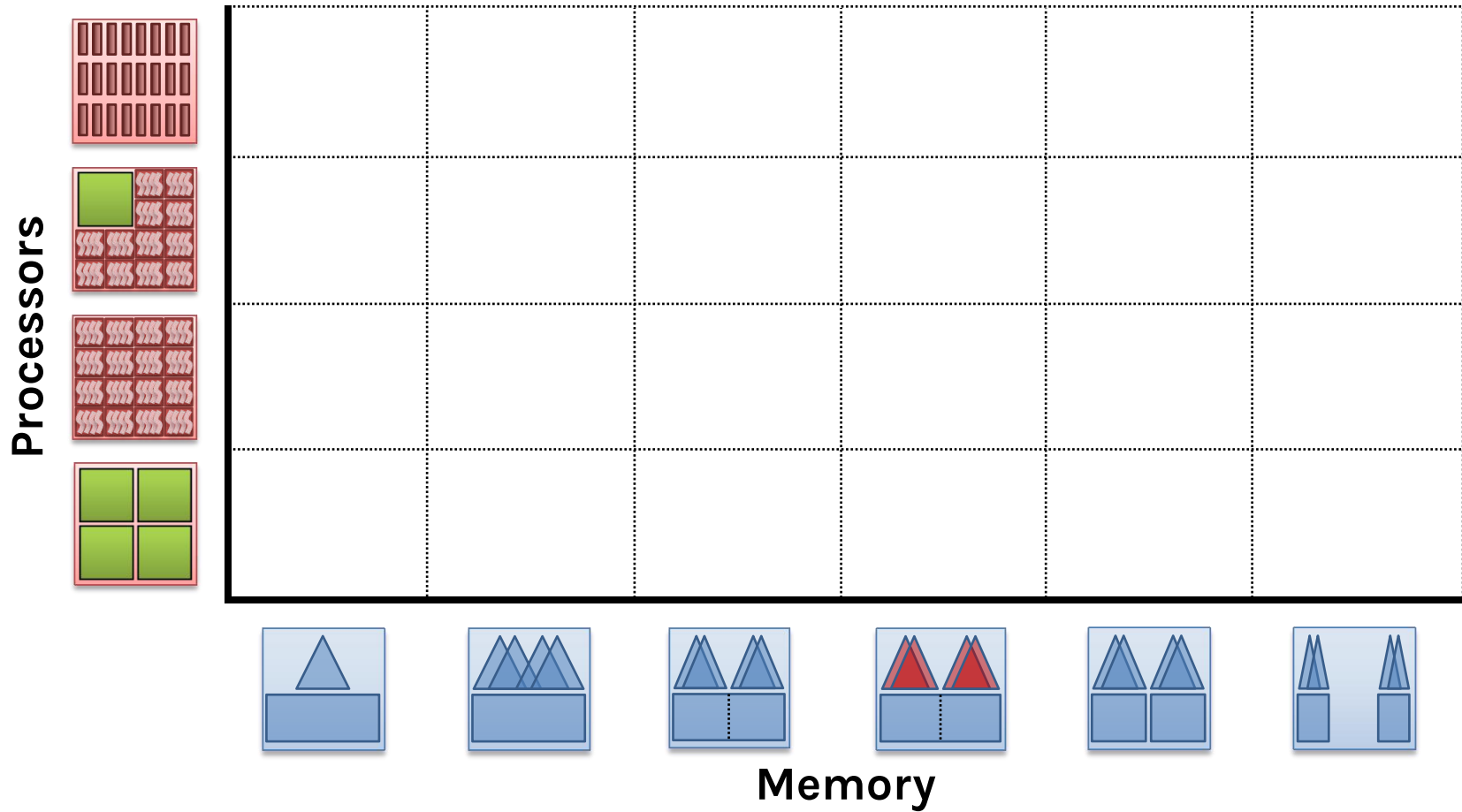


- The memory bus is replaced with a network
- We have islands of computing power, which can be closer or farther apart
- Typical of cloud computing, and the idea of cloud-core
- Two additional concerns for programmers (often abstracted by libraries):
 - *reliability*: nodes come and go
 - *latency*: the network congestion and the distance play an important role

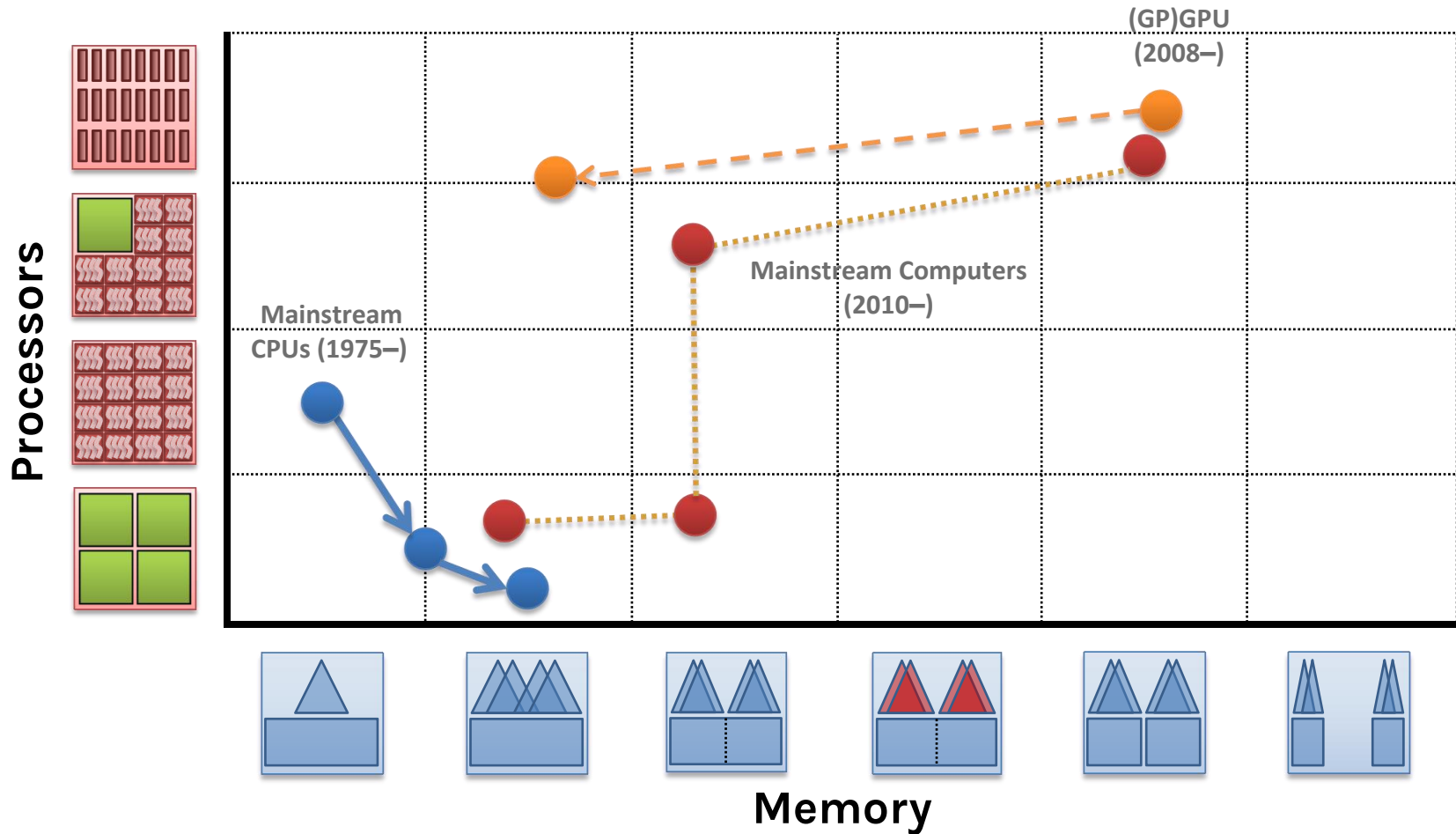
Memory



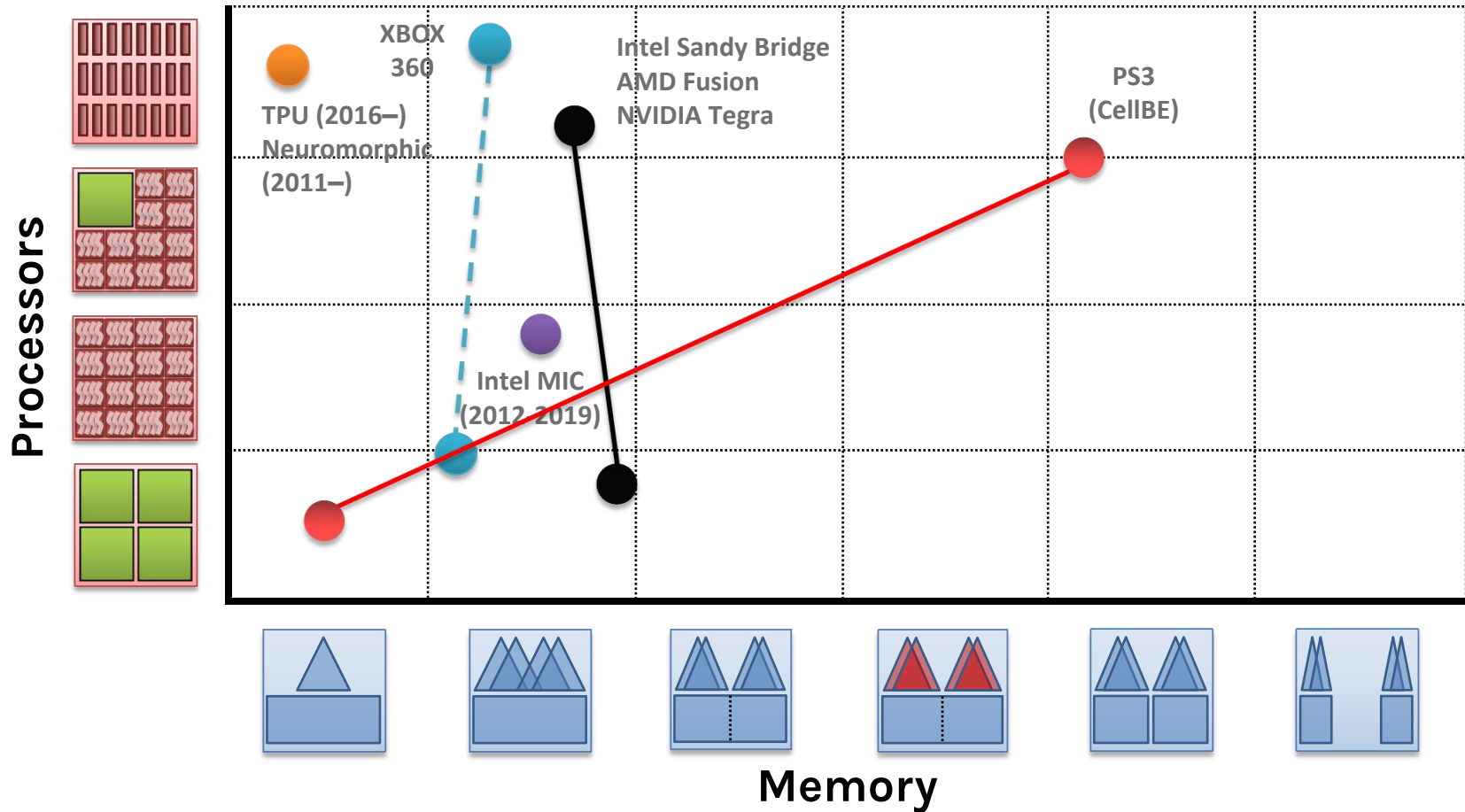
Charting the landscape



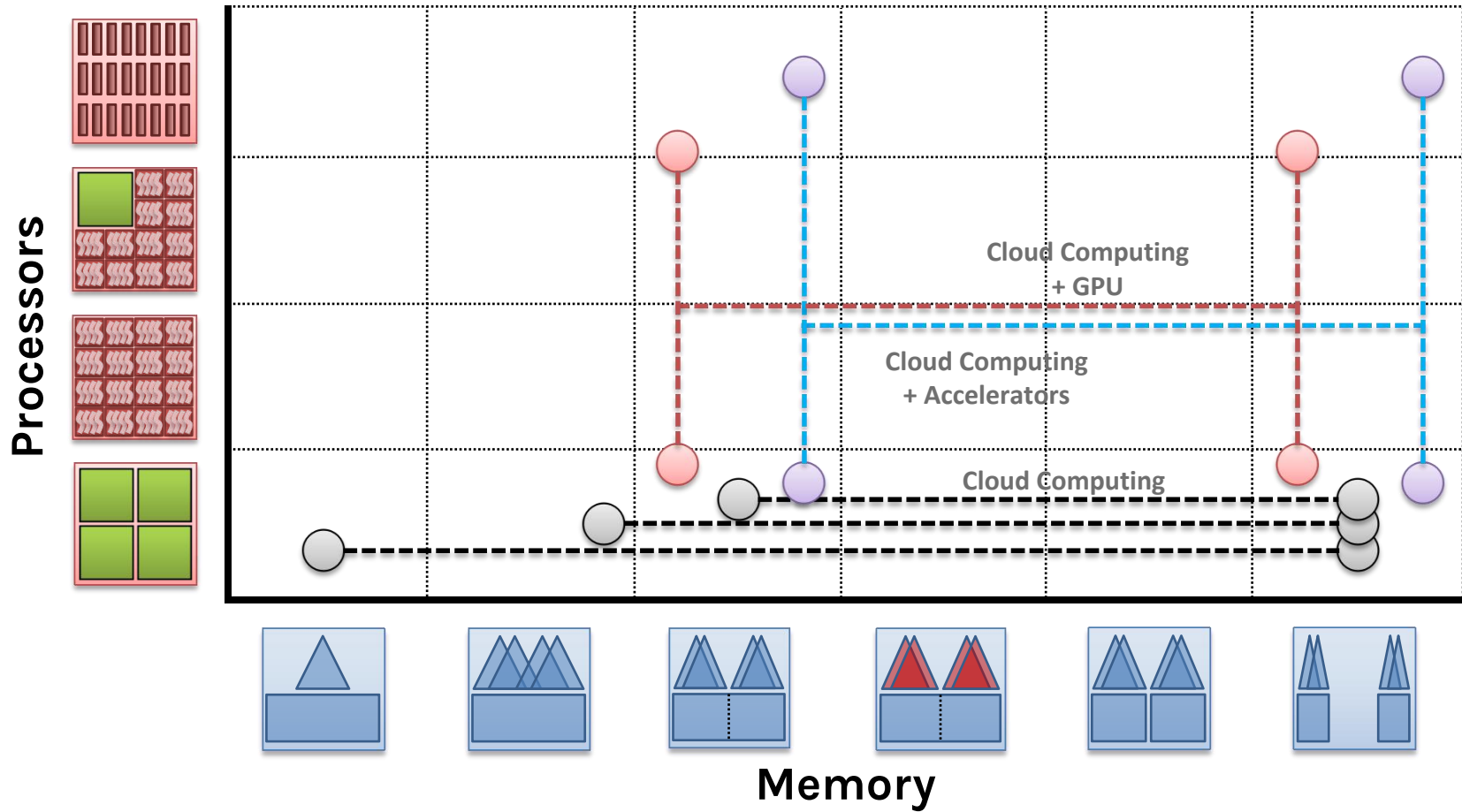
Mainstream Hardware: 1970s—today



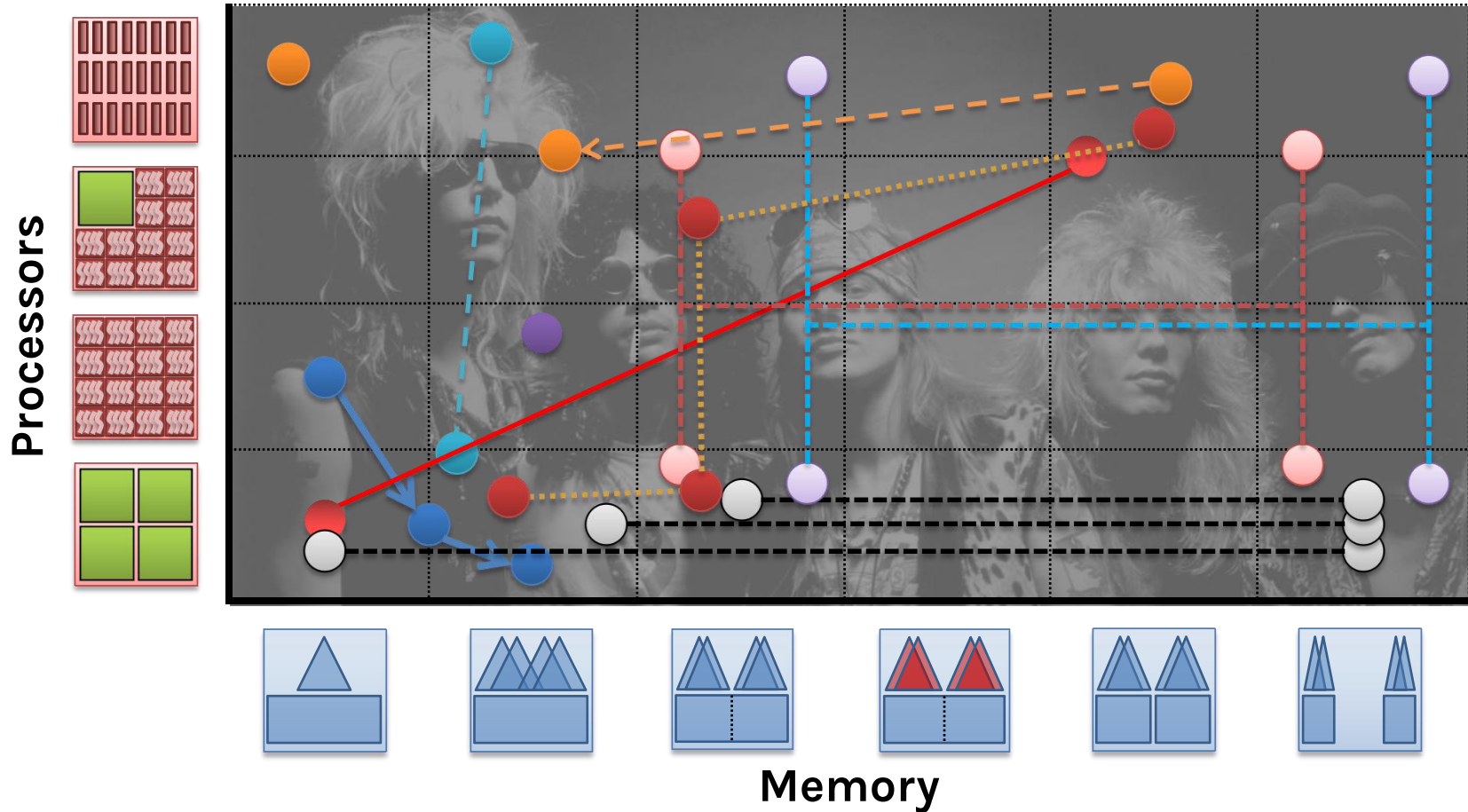
Recent Hardware Trends



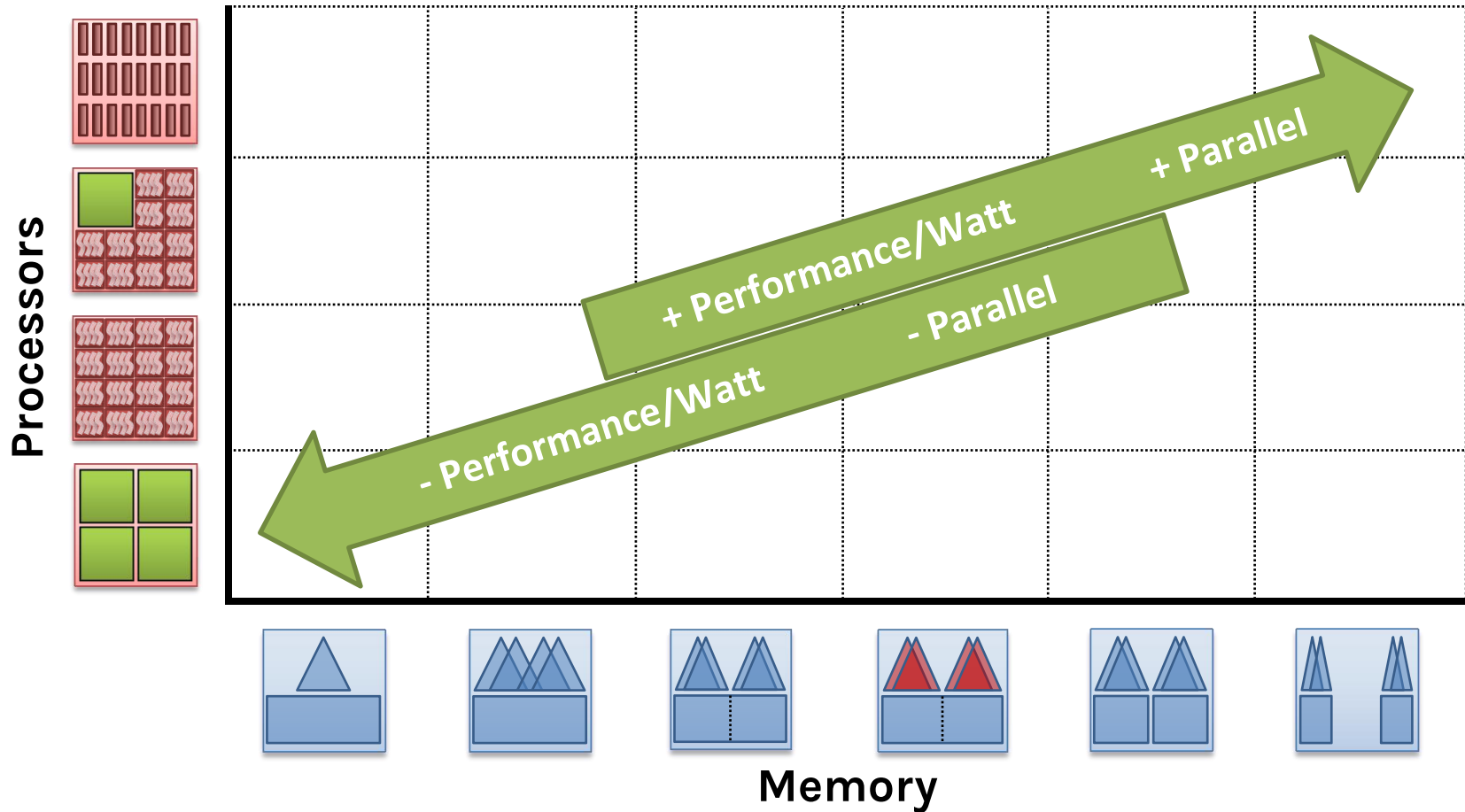
The Cloud



Welcome to the jungle...



Charting the landscape



Petascale Computing

- Petascale: a computer system capable of reaching performance in excess of one petaflops, i.e. one *quadrillion floating point operations per second*.
- Level already reached in ~2007
 - Several in the US
 - Four in China
 - One in Russia
 - Some in Europe

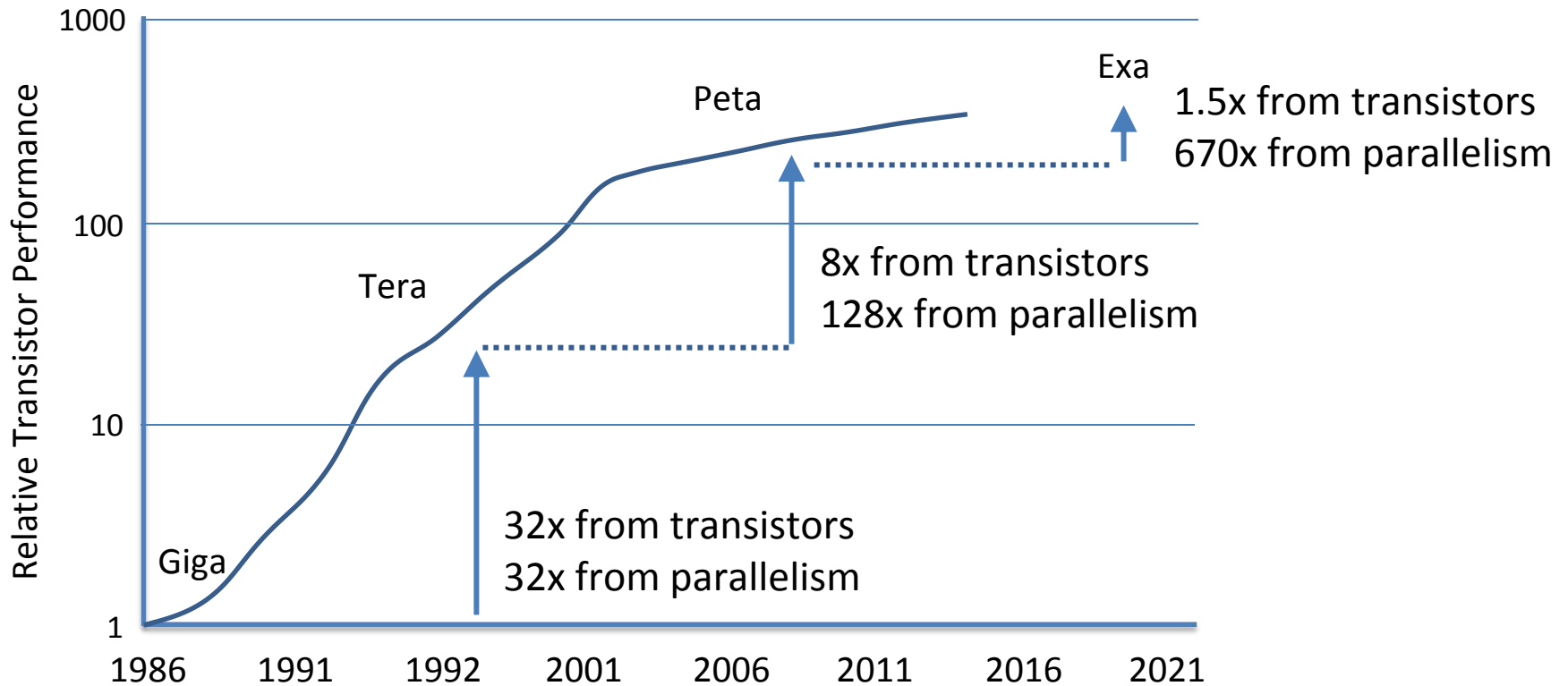


Exascale Computing

- Computing systems capable of at least one exaFLOPS, or a billion billion (i.e. a quintillion) calculations per second
- Expected to enter the market in 2020/2021
 - Involved both research and industrial partners
 - Huge effort put by US, Europe, and China

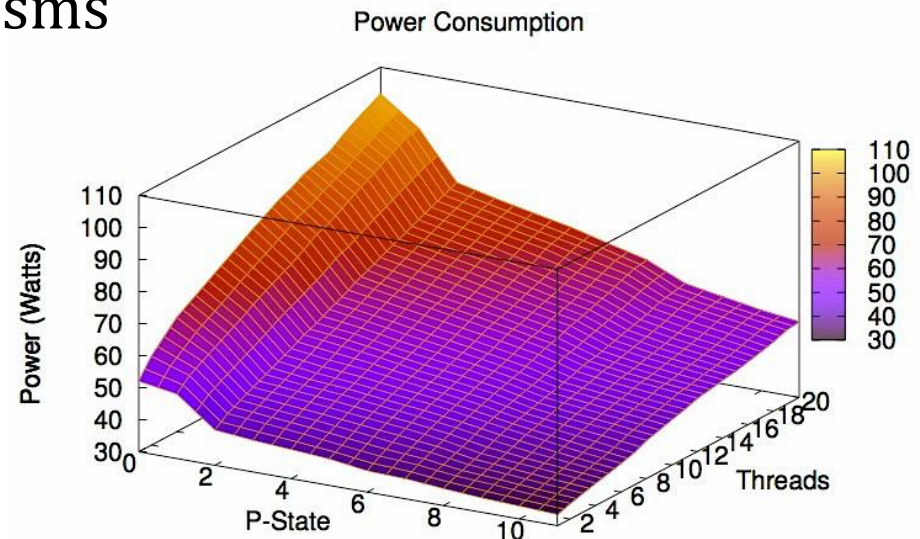


From Petascale to Exascale



Complicating the Picture: Dark Silicon

- **Dark silicon is increasing** (portion of circuitry that cannot be powered off for TDP constraints)
- We won't be able to run all computing units powered on and at the max performance state
- **Power management** mechanisms are required to **fine-tune allocation** of the power budget

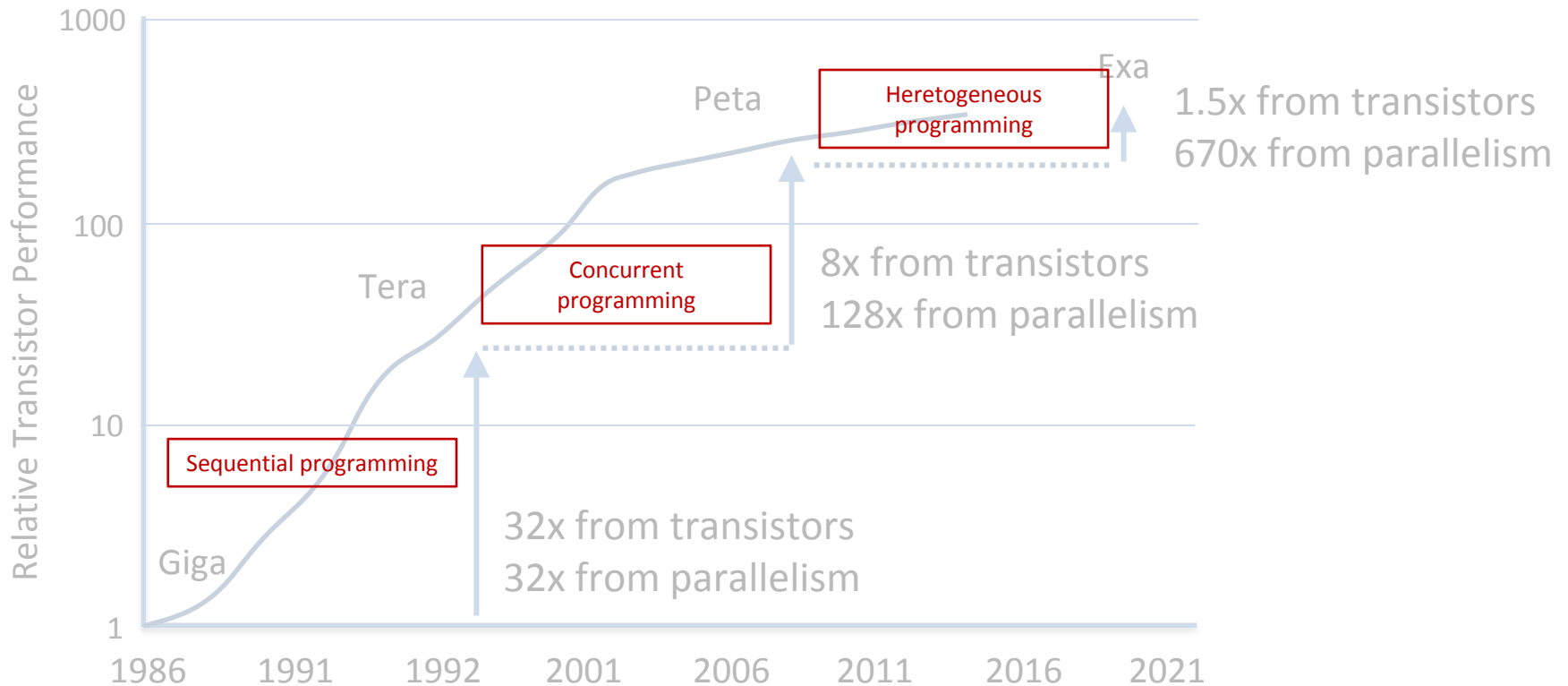


Recap: The Hardware Perspective

- **Parallelism** is the **de-facto standard** to support the exascale transition
- **General purpose** computing units even through parallelism **cannot reach exascale** with tolerable power consumption
- **Heterogeneity of specialized** computing units is key
 - FPGA
 - GPUs
 - Same ISA heterogeneous chips
 - Co-processors (Xeon Phi)



The Software Perspective



The Software Perspective

- Applications will need to be at least massively parallel, and ideally able to use non-local cores and heterogeneous cores
- Efficiency and performance optimization will get more, not less, important
- Programming languages and systems will increasingly be forced to deal with heterogeneous distributed parallelism



Complexity in Plain Sight

- The hardware is **no longer hiding** its complexity
- Software libraries (e.g. OpenCL) require **understanding** of the **underlying hardware** infrastructure
- Who will be using heterogeneous architectures?
 - Computer scientists
 - Biologists
 - Medical scientists
 - Economists
 - Genetic engineers
 - Etc.



Questions?

