Prefetching

ETH 263-2210-00L Computer Architecture : Guest Lecture 11-22-2019
Stephan Meier
Some slides authored by Tyler Huberty, Onur Mutlu (used with permission)

Outline

A few definitions before we start…

Memory Hierarchy: the way memory is organized, typically includes a series of smaller cache memories in addition to main memory (DRAM)

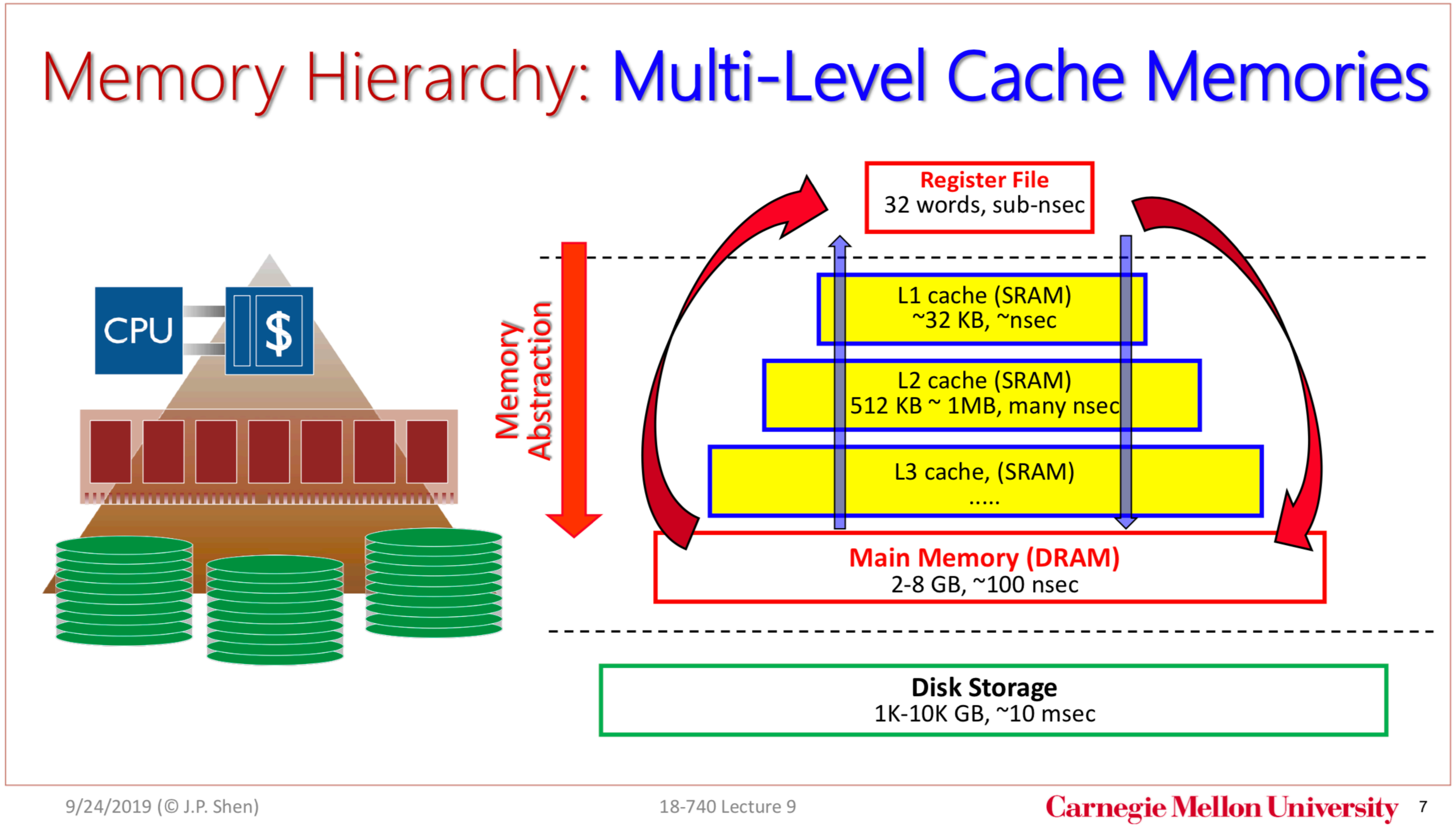
Memory Latency: time to access memory (clock cycles or ns)

Read/Write Latency: time to read data from/write data to memory

Memory Bandwidth (BW): the rate at which data can be moved to/from a memory (MB/s)

Memory Footprint: the total space occupied by an application in memory. The active footprint or working set refers to the set actively being used

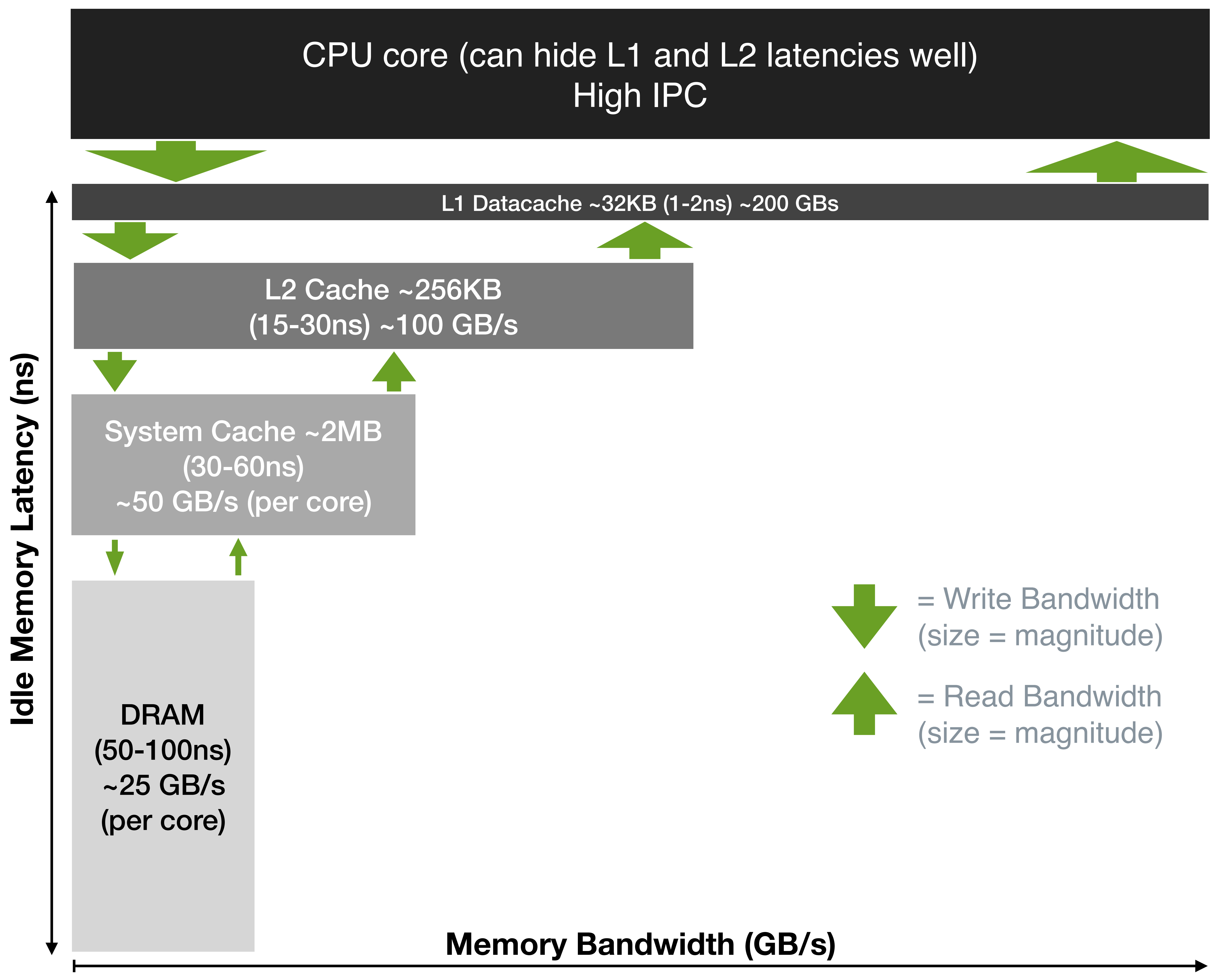
Memory Hierarchy - What you've already seen…



Memory Hierarchy: Multi-Level Cache Memories

9/24/2019 (© J.P. Shen) 18-740 Lecture 9 Carnegie Mellon University 7

Memory Hierarchy - Viewed another way (Ideal)



Upper layers are low latency and high bandwidth

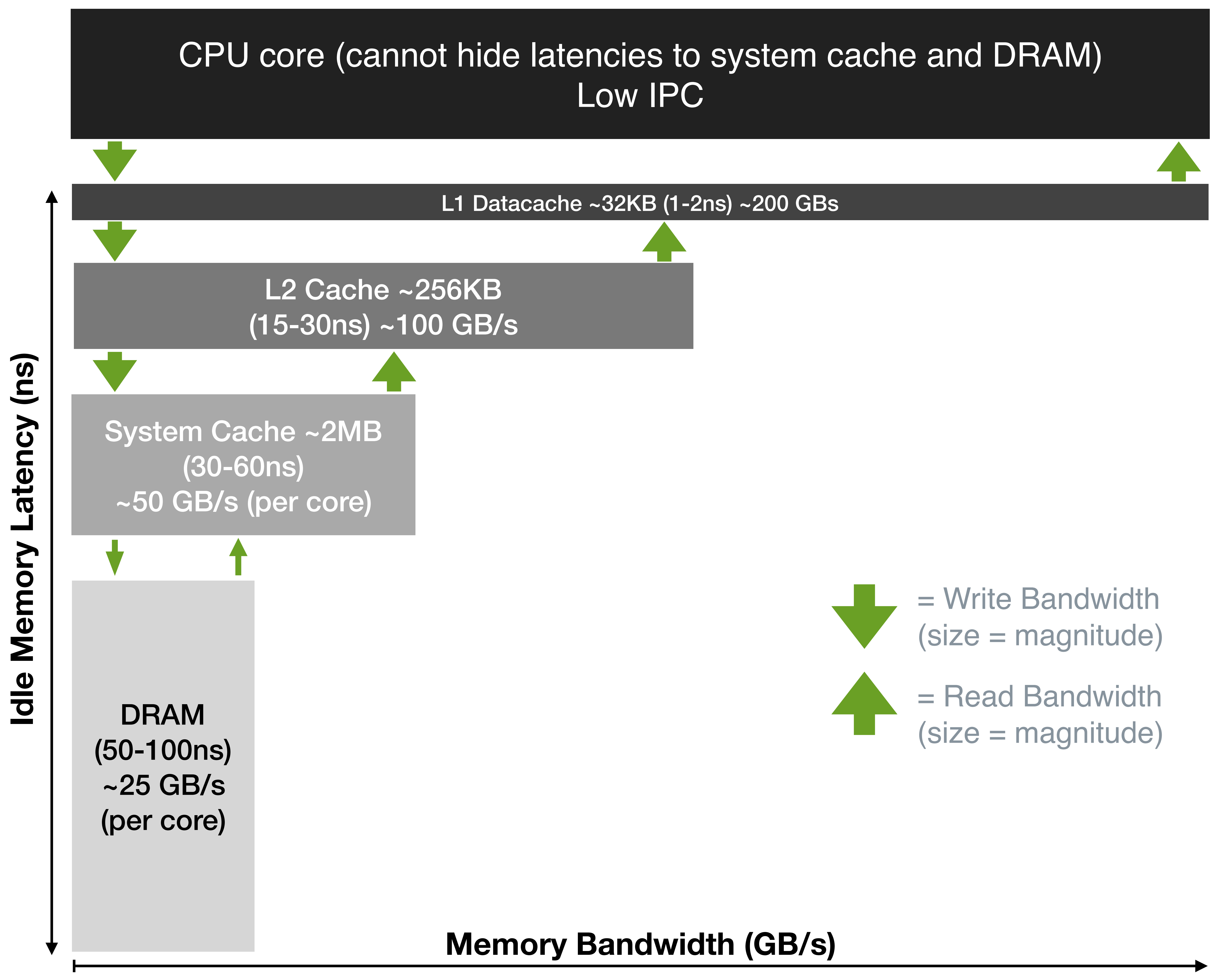
Most requests hit in L1 and L2 caches

Each layer of the hierarchy acts as a bandwidth filter

Slow DRAM sees little bandwidth.
Some perf loss due to read latency

Memory Hierarchy - Viewed another way (large active footprint)

(eg big dot product)



Filtering does not work effectively…

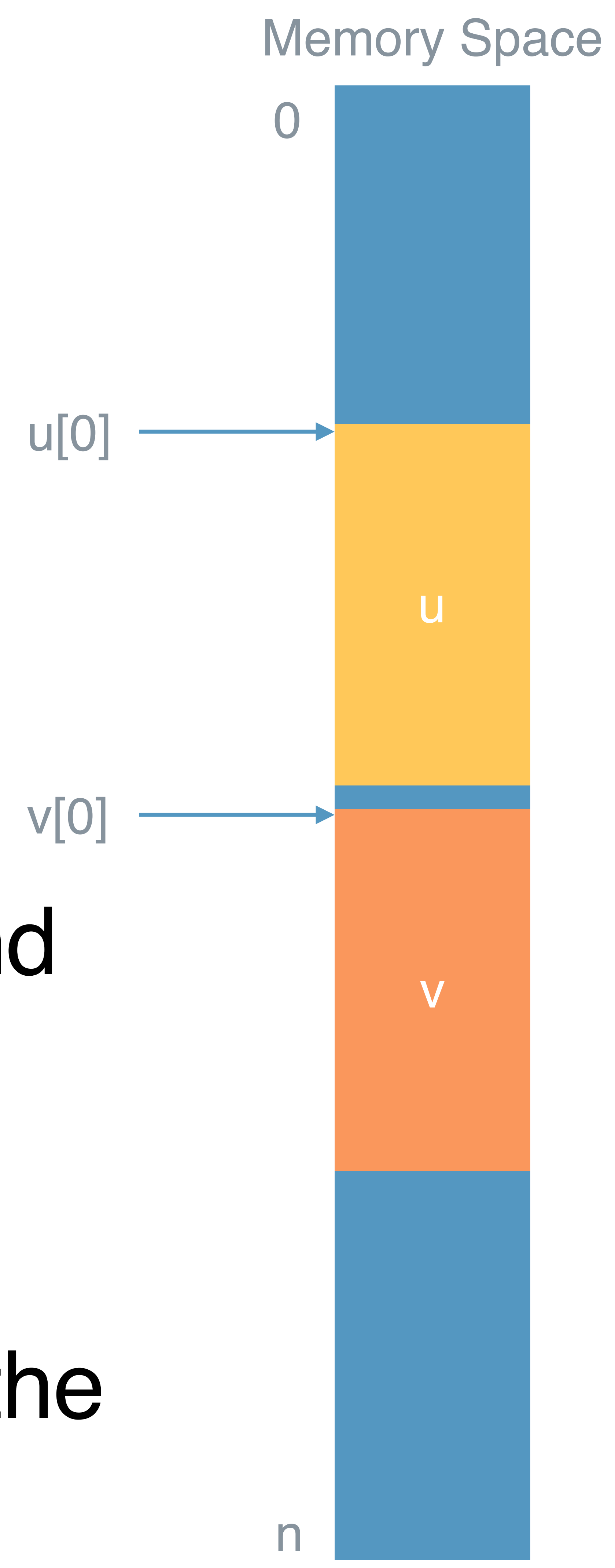
Most requests miss in L1 and L2 caches

Majority of requests go to system cache or DRAM. BW drops

Long latencies exposed that CPU cannot hide. Read BW drops even more!

Latency is costly: a simple dot product function in c

```
double dot_product(double v[], double u[], int n)
{
    double result = 0.0;
    for (int i = 0; i < n; i++)
        result += v[i]*u[i];
    return result;
}
```



- If the data is stale/cold (not yet touched) it will likely be in DRAM
- If u and v are in DRAM (100ns latency) and are each 1MB in size and reads are done with a 64B fill granule:
  - time = 1MB/64B * 100ns = 1638 us
  - If data is in the L1 cache for a modern 4Ghz CPU it could process the data at 1 loop iteration/cycle ie 1MB/8B/4 = 32.536 us (50X faster)

How to hide latency? Latency tolerance techniques

- **Caches** hide latency much of the time (via hits)
  - Work well (high hit rate) when the working set fits and spatial and temporal locality are favorable
  - 64B fetch granule helps hide some latency (to rest of the line) even on misses!
- **Reordering code** (compiler or hand optimized) to hoist loads and block operations to fit in caches (can fault, still need resources)
- **Software prefetch instructions** (non-faulting, no destination register)

Back to our simple example - ways to hide latency

- Dot product loop in pseudo assembly

```
loop:
      ld r1, [r3]  # load u[n]
      ld r2, [r4]  # load v[n]
      add r3, #4   # n++ for u
      add r4, #4   # n++ for v
      mul r1, r2   # r1 <- u[n] * v[n]
      add r6, r1   # sum += r1
      cmp r3, r5   # check if done
      bne loop
```

- Dot product loop with SW prefetch

```
loop:
    sw_pf [r3+64] # Start next u fill
    sw_pf [r4+64] # Start next v fill
    ld r1, [r3]  # load u[n]
    ld r2, [r4]  # load v[n]
    add r3, #4   # n++ for u
    add r4, #4   # n++ for v
    mul r1, r2   # r1 <- u[n] * v[n]
    add r6, r1   # sum += r1
    cmp r3, r5   # check if done
    bne loop
```

- Assume u[0:n-1] and v[0:n-1] are in DRAM
- A simple in-order machine will only issue u[0] and v[0] and block waiting on 2 data cache fills
- The next loop iteration will not even get to issue as the first 2 loads block further issue
- When the fills complete, another 8 iterations will be performed since each 64B fill will satisfy 64/8=8 loads (assuming 8B double precision floats)
- Restructuring code does not help due to blocking loads
- Software prefetch (non-blocking fill) helps. Consumes fetch BW. Requires human/compiler tuning. How many ahead? Loop unroll 8x for better performance.

More latency tolerance techniques

- **Caches** hide latency much of the time (via hits)
  - Work well (high hit rate) when the working set fits and spatial and temporal locality are favorable
  - 64B fetch granule helps hide some latency (to rest of the line) even on misses!
- **Reordering code** (compiler or hand optimized) to hoist loads and block operations to fit in caches (can fault, still need resources)
- **Software prefetch instructions** (non-faulting, no destination register)
- **Speculative execution** of loads and stores along with multiple outstanding misses (non-blocking caches) help as well

The large out of order machine

- How about a large out of order architecture with many outstanding misses?
  - Core complexity increases, resources are expensive (eg physical registers)
  - Limited by ability to scale complex core structures like the load and store queues, reservation stations, and register files.
  - Younger loads may need to speculate past older unresolved stores.   More cost and complexity!
  - Modern high performance cores do all of the above.
  - Compiler techniques like loop unrolling and load hoisting can now help too.
  - In practice limited to hiding L1 and L2 cache latencies.

Size of instruction window

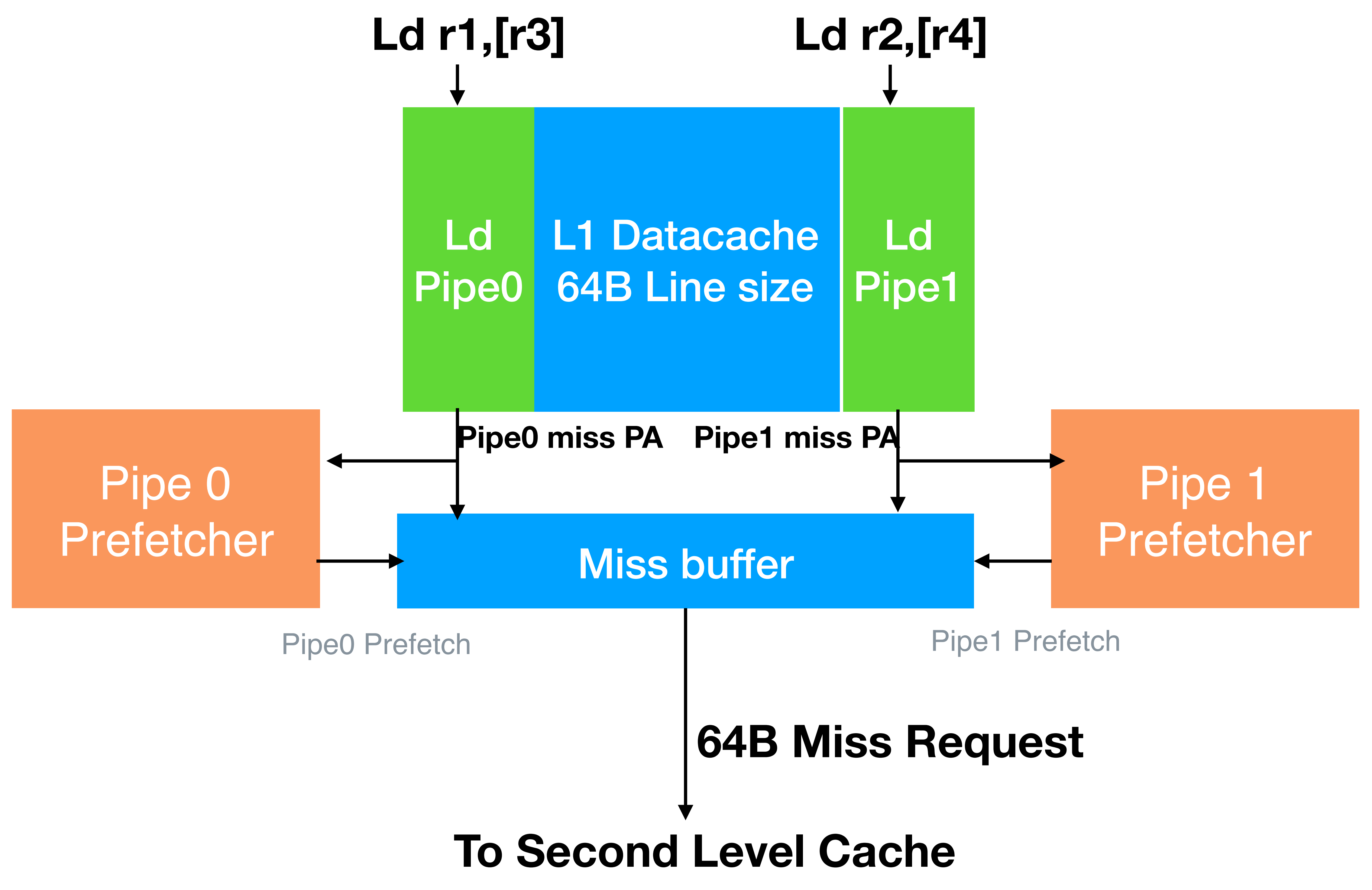
Little’s Law (occupancy = latency * throughput).

- throughput = window size / latency
- High latency really hurts! For a fixed window size, high latency reduces throughput (IPC)

More latency tolerance techniques

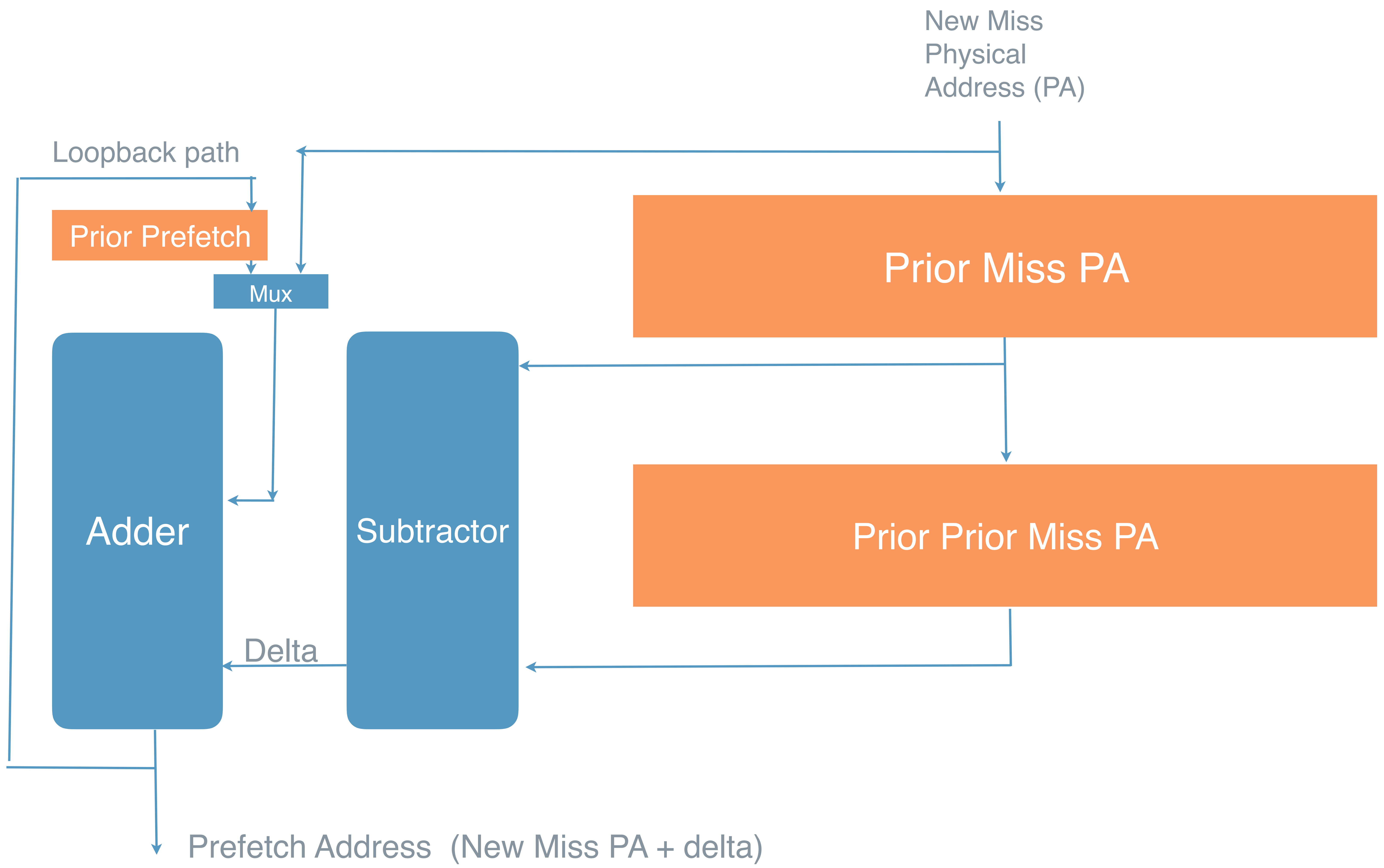
- **Caches** hide latency much of the time (via hits)
  - Work well (high hit rate) when the working set fits and spatial and temporal locality are favorable
  - 64B fetch granule helps hide some latency (to rest of the line) even on misses!
- **Reordering code** (compiler or hand optimized) to hoist loads and block operations to fit in caches (can fault, still need resources)
- **Software prefetch instructions** (non-faulting, no destination register)
- **Speculative execution** of loads and stores along with multiple outstanding misses (non-blocking caches) help as well
- **Hardware prefetching…**

What about adding hardware?



- Consider a simple in order dual issue machine
- The two loads in our dot product can dual issue as shown
- Capture last 2 miss addresses in each pipe
- Compute a delta between these addresses
- Add delta to current miss address
- If it is to a new cacheline, issue a prefetch fill request

Toy hardware prefetcher



- A simple datapath captures two back to back miss addresses
- The delta between these is computed
- This delta is then added to to the incoming miss address to generate a prefetch
- The prefetch degree here is 1. Limited latency hiding.
- Consider looping n times through adder adding delta each time. Degree then n.

Prefetcher metrics

- Given a hardware prefetcher, how do we evaluate it?
- We can characterize the trivial prefetcher with via a set of metrics:
  - Even for this simple workload it may not be accurate. It assumes a static pipe assignment for instance
    - Accuracy = useful prefetches / total prefetches
  - It also does not cover all accesses. At startup there are no prefetches
    - Coverage = total prefetches / total unique accesses
  - Finally the prefetches may be accurate but arrive too late to help much
    - ***Timeliness = number of prefetches arriving on time / total prefetches***

What to prefetch? Instruction vs Data prefetching

- So far we have assumed that we are prefetching program data
  - This reduces the latency of load/store operations
- What about the program instructions?
  - Small programs will reside in the instruction cache (eg Dhrystone benchmark!)
  - Larger programs (think UI frameworks, etc) will not even fit in the instruction and L2 cache
  - For these we want to anticipate the instructions needed next and bring them closer to the processor to reduce fetch latency

Outline

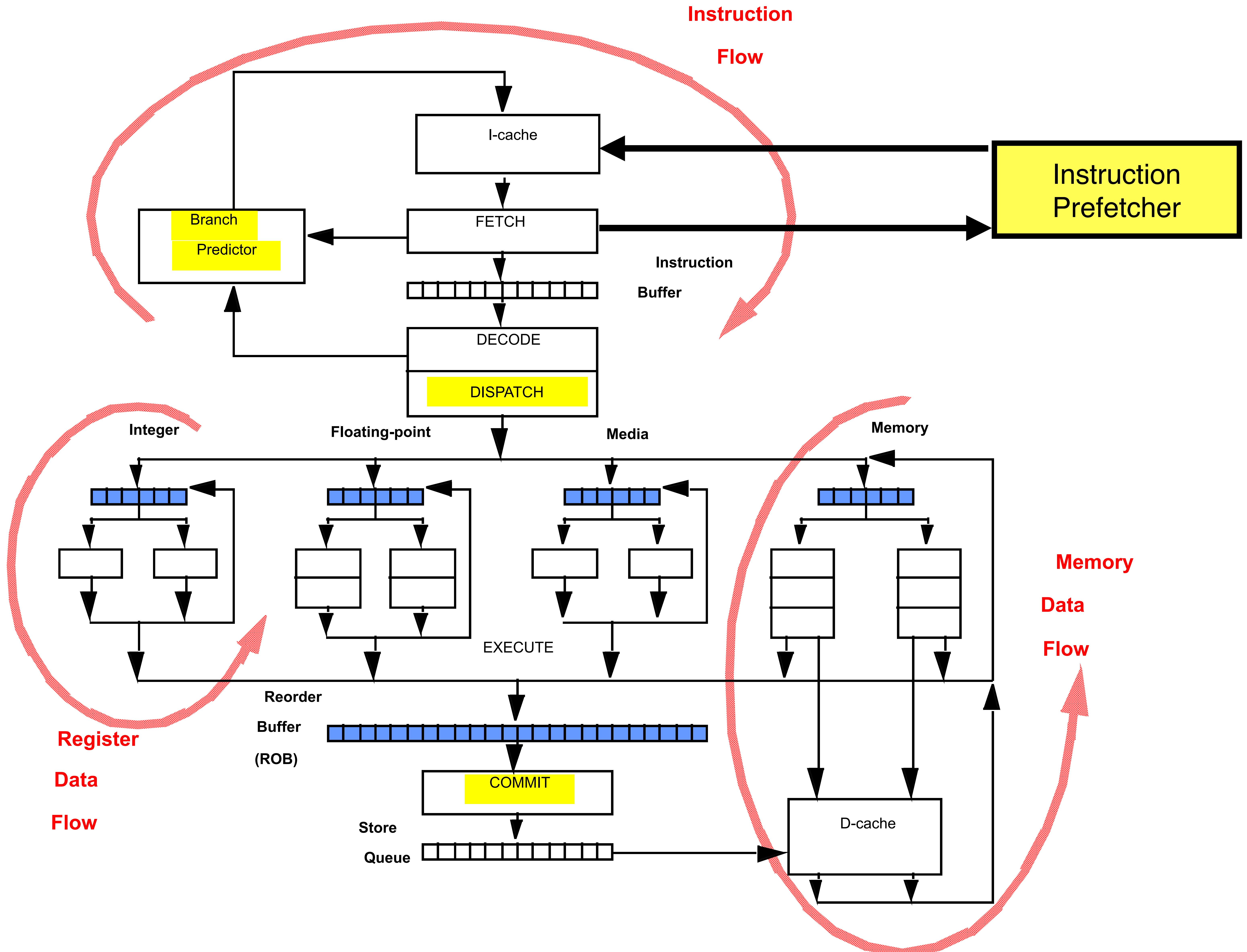
Motivation

Instruction prefetching

Data prefetching

Research directions

Instruction Prefetching Overview



Basic Schemes. Heuristic-Based Next-N-Line Prefetching

Prefetch next N sequential cachelines from current cacheline

- Example:
  - Intel 8086 lacked an instruction cache but had a Prefetch Input Queue (PIQ), which read up to six bytes ahead of current instruction
  - [Challenge] Flushed PIQ on encountering a branch or mode change
- Considerations:
  - Prefetch on miss only? Value of N?

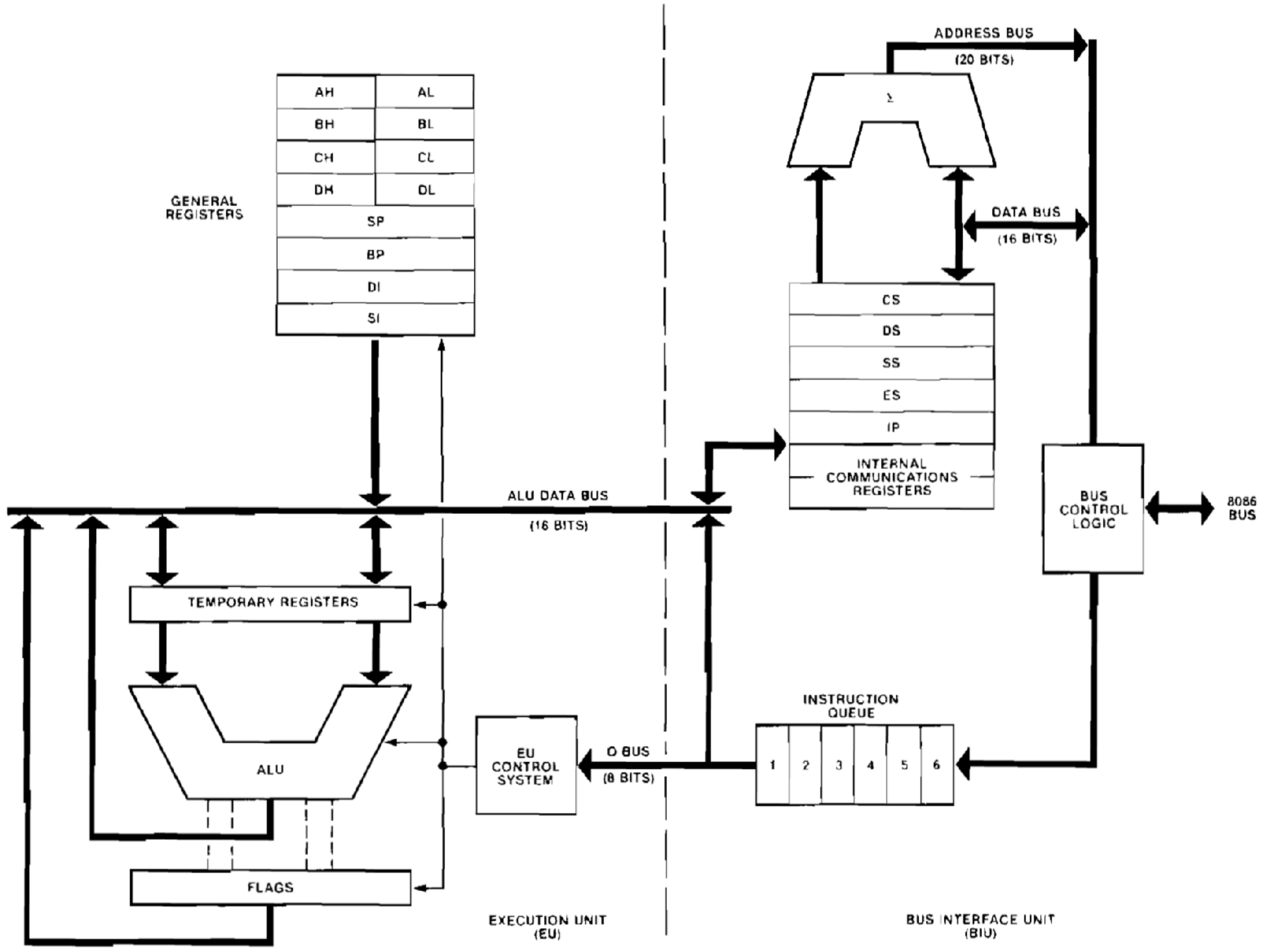


Figure 4-3. 8086 Elementary Block Diagram

Source: Intel 8086 Family User's Manual (1979)

Basic Schemes. History-Based Target Line Prefetching

Prefetch next cacheline executed historically following current cacheline

- Maintain table of {current line, next line} pairs in hardware

+Prefetch coverage for non-sequential program flow

- [Challenge] Lacks coverage for compulsory misses
- [Challenge] Storage costs to cover large program footprints
- [Challenge] Assumes historic conditional branch direction consistent with future direction

Basic Schemes. Heuristic-Based Wrong-Path Prefetching

Prefetch cacheline containing target of conditional branch once decoded (even if not taken)

+ Greater than zero probability branch may be taken in near future even if not taken now
- [Challenge] Prefetches for immediately taken branches may not be timely

Basic Schemes. Hybrid Prefetching

Combine Next-N-Line Prefetching with Target Prefetching and/or Wrong-Path Prefetching

+ Broader coverage than individual policies, including for compulsory misses; taken and not taken branches
- [Challenge] Incomplete or poor coverage for some branch types (i.e. conditional; subroutine call; register indirect)

• Further reading: Pierce and Mudge, “Wrong-Path Instruction Prefetching,” MICRO 1996.

Can one do better?

Advanced Directions in Instruction Prefetching

Recall presence of Branch Predictor, including Branch Target Buffer (BTB)

- In BTB-directed instruction prefetching schemes, a second or decoupled Branch Predictor looks for predecoded branches in cacheline fills and uses BTB to predict target addresses to prefetch

+ Allows construction of predicted control flow without storage costs beyond those already incurred by Branch Predictor

+ Enables prefetch degree >1

- [Challenge] BTB misses can stall instruction prefetching. Possible solution: BTB prefetching :-)

- Further reading: Kumar et. al., “Boomerang: a Metadata-Free Architecture for Control Flow Delivery,” HPCA 2017.

Outline

Motivation

Instruction prefetching

Data prefetching

Research directions

Data Prefetching

The following slides adapted (with permission) from Professor Onur Mutlu's ETH Zürich Computer Architecture course

https://people.inf.ethz.ch/omutlu/

https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=onur-comparch-fall2017-lecture18-prefetching-afterlecture.pdf

Next-Line Prefetchers

Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)

Next-line prefetcher (or next sequential prefetcher)

Tradeoffs:

+ Simple to implement. No need for sophisticated pattern detection
+ Works well for sequential/streaming access patterns (instructions?)

-- Can waste bandwidth with irregular patterns

-- And, even regular patterns:

- What is the prefetch accuracy if access stride = 2 and N = 1?
- What if the program is traversing memory from higher to lower addresses?
- Also prefetch “previous” N cache lines?

Stride Prefetchers

Two kinds

- Instruction program counter (PC) based
- Cache block address based

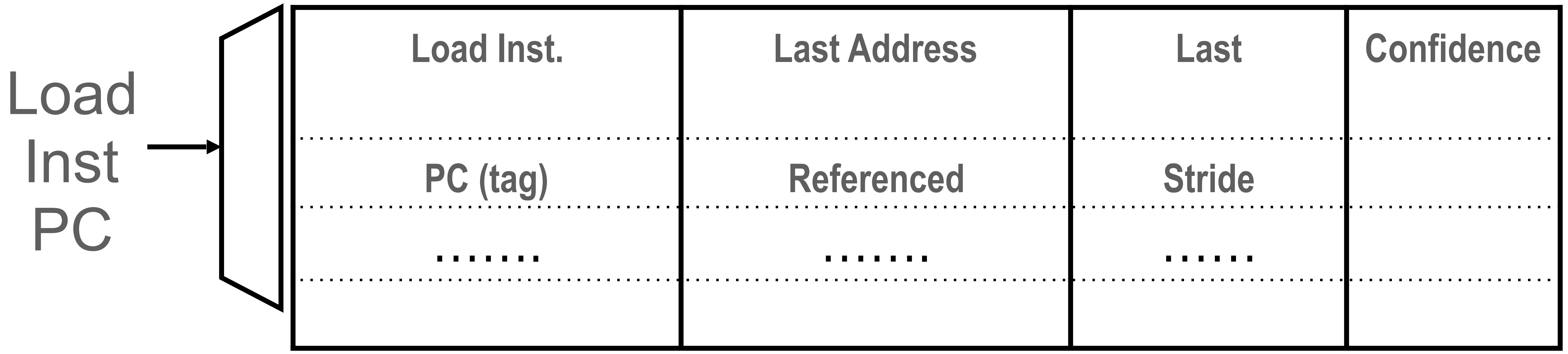
Instruction based:

Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.

Idea:

- Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
- Next time the same load instruction is fetched, prefetch last address + stride

Instruction Based Stride Prefetching



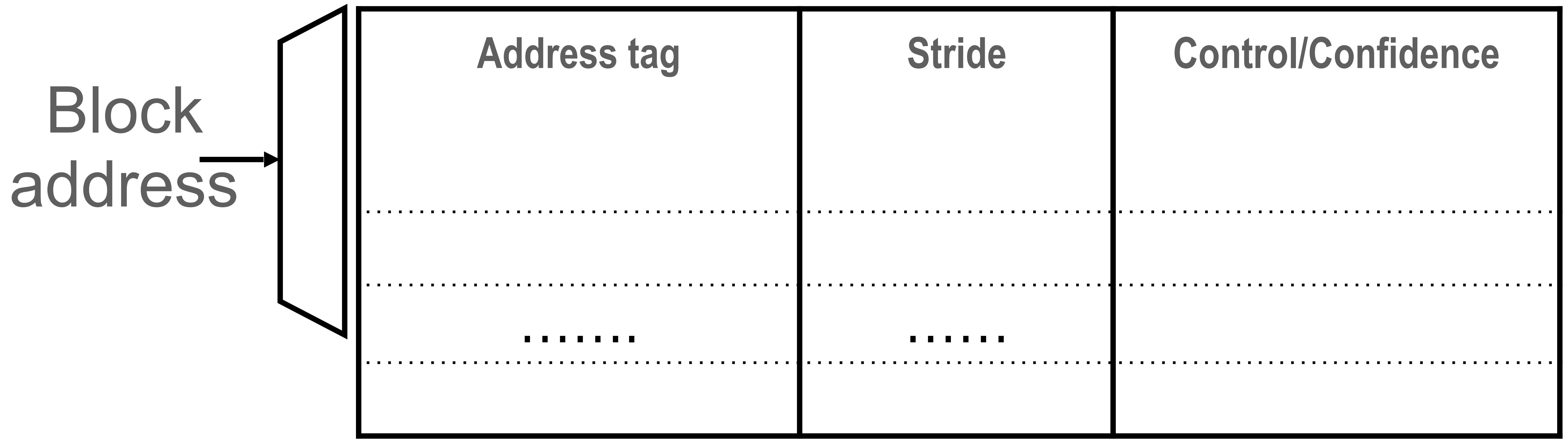
What is the problem with this?

- How far can the prefetcher get ahead of the demand access stream?
- Initiating the prefetch when the load is fetched the next time can be too late
  - Load will access the data cache soon after it is fetched!

Solutions:

- Use lookahead PC to index the prefetcher table (decouple frontend of the processor from backend)
- Prefetch ahead (last address + N*stride)
- Generate multiple prefetches

Cache-Block Address Based Stride Prefetching



Can detect

- A, A+N, A+2N, A+3N, …
- Stream buffers are a special case of cache block address based stride prefetching where N = 1

Tradeoffs in Stride Prefetching

Instruction based stride prefetching vs. cache block address based stride prefetching

The latter can exploit strides that occur due to the interaction of multiple instructions

The latter can more easily get further ahead of the processor access stream

- No need for lookahead PC

The latter is more hardware intensive

- Usually there are more data addresses to monitor than instructions

How to Prefetch More Irregular Access Patterns?

Regular patterns: Stride, stream prefetchers do well

More irregular access patterns

- Indirect array accesses
- Linked data structures
- Multiple regular strides (1,2,3,1,2,3,1,2,3,…)
- Random patterns?
- Generalized prefetcher for all patterns?

Correlation based prefetchers

Content-directed prefetchers

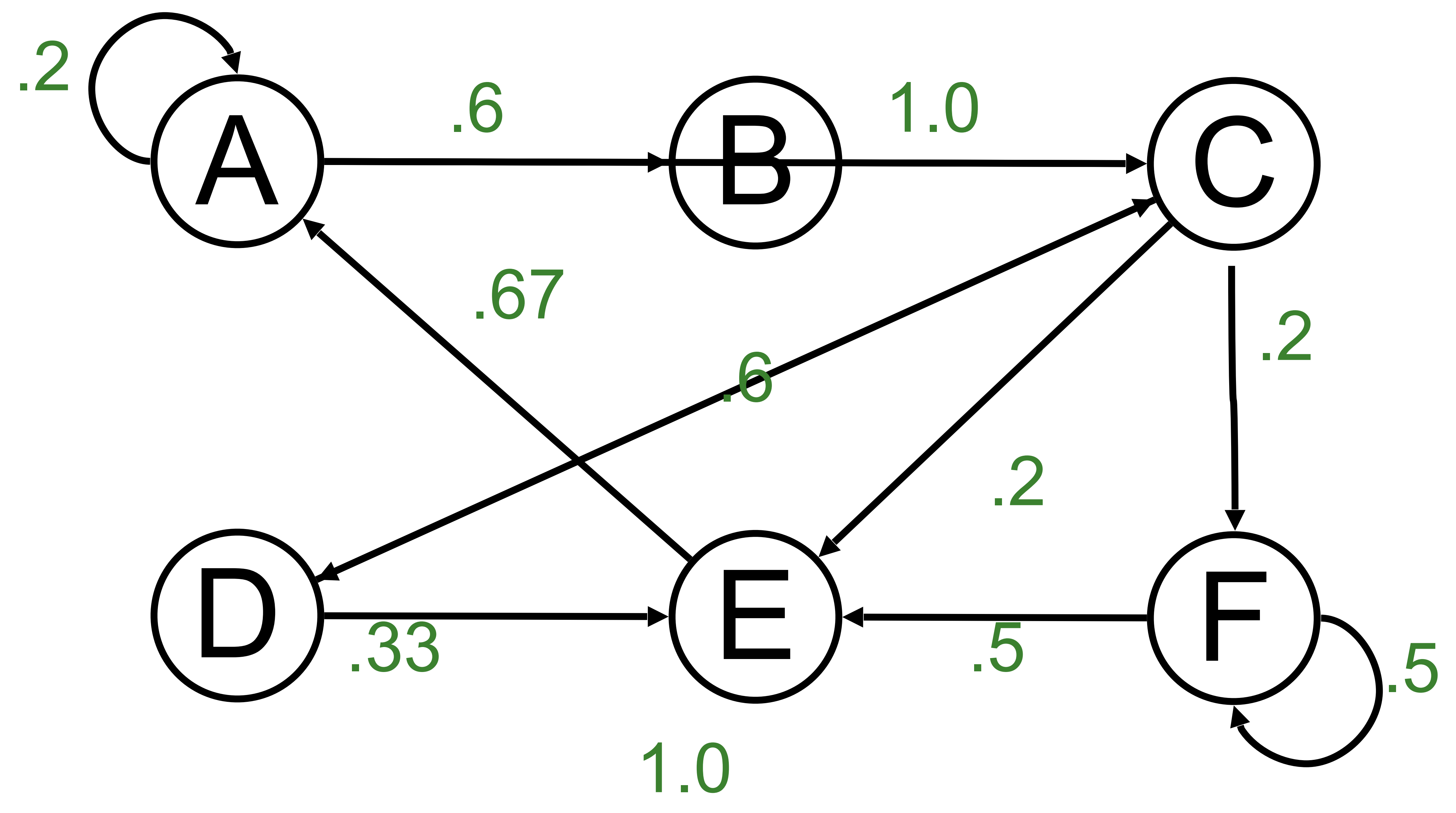
Precomputation or execution-based prefetchers

Address Correlation Based Prefetching (I)

Consider the following history of cache block addresses

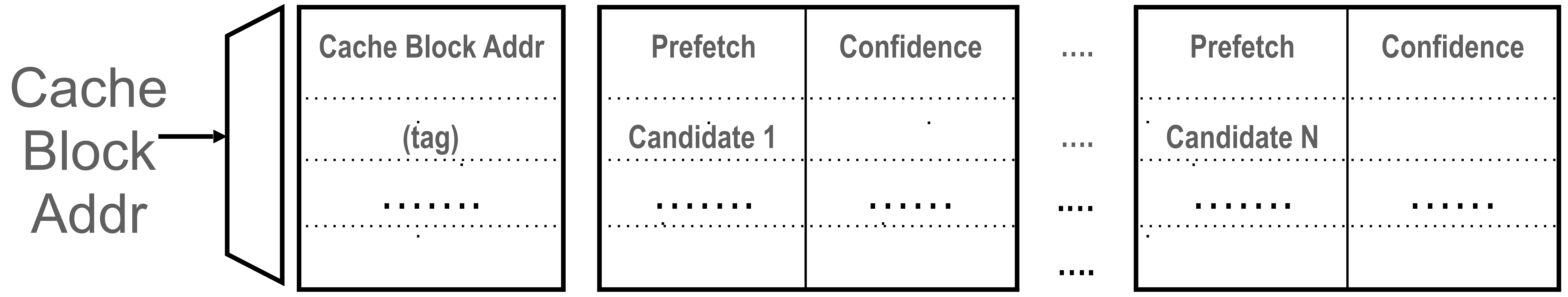
A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C

After referencing a particular address (say A or E), some addresses are more likely to be referenced next



Markov Model

Address Correlation Based Prefetching (II)



Idea:

- Record the likely-next addresses (B, C, D) after seeing an address A
- Next time A is accessed, prefetch B, C, D
- A is said to be correlated with B, C, D

Prefetch up to N next addresses to increase coverage

Prefetch accuracy can be improved by using multiple addresses as key for the next address: (A, B) → (C)

(A,B) correlated with C

Joseph and Grunwald, "Prefetching using Markov Predictors," ISCA 1997.

- Also called "Markov prefetchers"

Address Correlation Based Prefetching (III)

Advantages:

- Can cover arbitrary access patterns
  - Linked data structures
  - Streaming patterns (though not so efficiently!)

Disadvantages:

- Correlation table needs to be very large for high coverage
  - Recording every miss address and its subsequent miss addresses is infeasible
- Can have low timeliness: Lookahead is limited since a prefetch for the next access/miss is initiated right after previous
- Can consume a lot of memory bandwidth
- Especially when Markov model probabilities (correlations) are low
- Cannot reduce compulsory misses

Hybrid Hardware Prefetchers

Many different access patterns

- Streaming, striding
- Linked data structures
- Localized random

Idea: Use multiple prefetchers to cover all patterns

+ Better prefetch coverage

-- More complexity

-- More bandwidth-intensive

-- Prefetchers start getting in each other’s way (contention, pollution)

  - Need to manage accesses from each prefetcher

Execution-based Prefetchers (I)

Idea: Pre-execute a piece of the (pruned) program solely for prefetching data

- Only need to distill pieces that lead to cache misses

Speculative thread: Pre-executed program piece can be considered a “thread”

Speculative thread can be executed

- On a separate processor/core
- On a separate hardware thread context (think fine-grained multithreading)
- On the same thread context in idle cycles (during cache misses)

Execution-based Prefetchers (II)

How to construct the speculative thread:

- Software based pruning and “spawn” instructions
- Hardware based pruning and “spawn” instructions
- Use the original program (no construction), but
  - Execute it faster without stalling and correctness constraints

Speculative thread

- Needs to discover misses before the main program
  - Avoid waiting/stalling and/or compute less
- To get ahead, uses
  - Perform only address generation computation, branch prediction, value prediction (to predict “unknown” values)
- Purely speculative so there is no need for recovery of main program if the speculative thread is incorrect

Prefetching in Multi-Core (I)

Prefetching shared data

- Coherence misses

Prefetch efficiency is a lot more important

- Bus bandwidth more precious
- Cache space more valuable

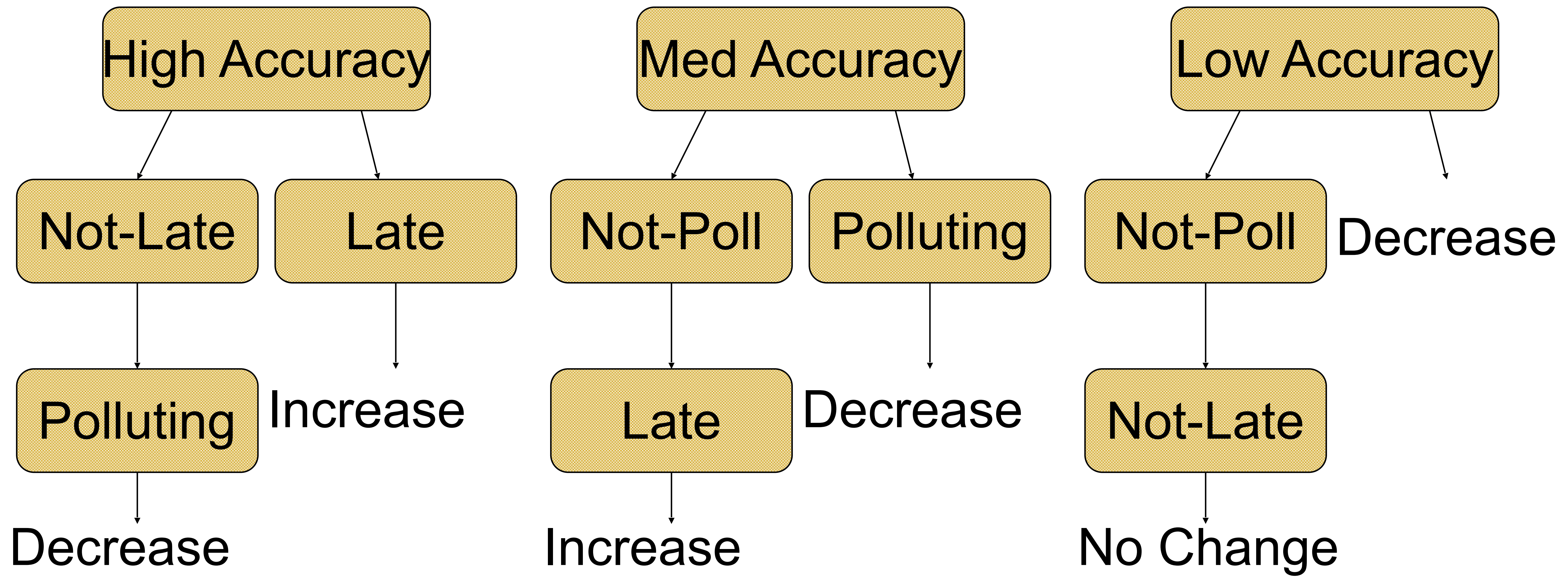
One cores’ prefetches interfere with other cores’ requests

- Cache conflicts
- Bus contention
- DRAM bank and row buffer contention

Feedback-Directed Prefetcher Throttling (I)

Idea:

- Dynamically monitor prefetcher performance metrics
- Throttle the prefetcher aggressiveness up/down based on past performance
- Change the location prefetches are inserted in cache based on past performance



Feedback-Directed Prefetcher Throttling (III)

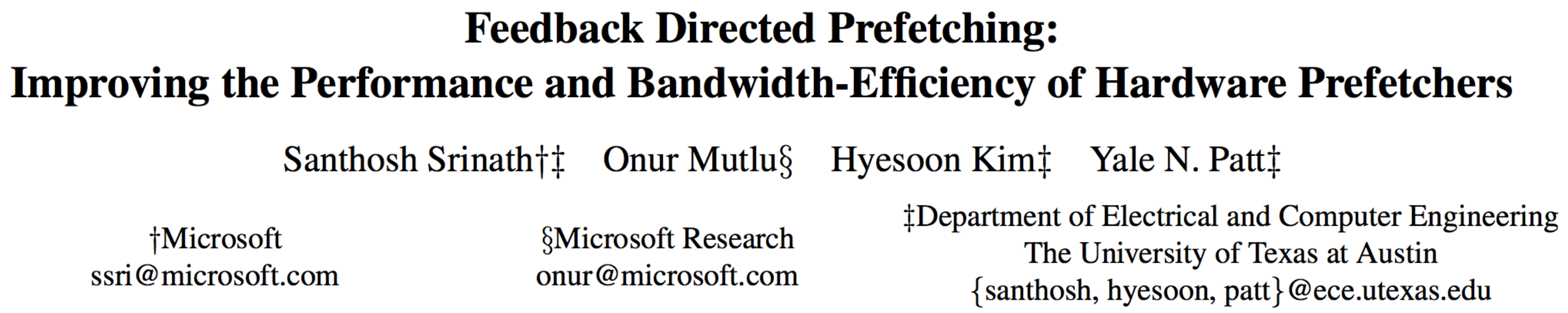
BPKI - Memory Bus Accesses per 1000 retired Instructions

- Includes effects of L2 demand misses as well as pollution induced misses and prefetches
- A measure of bus bandwidth usage

	No. Pref.	Very Cons	Mid	Very Aggr	FDP
IPC	0.85	1.21	1.47	1.57	1.67
BPKI	8.56	9.34	10.60	13.38	10.88

More on Feedback Directed Prefetching

Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers"
Proceedings of the 13th International Symposium on High-Performance Computer Architecture (**HPCA**), pages 63-74, Phoenix, AZ, February 2007. Slides (ppt)



**Feedback Directed Prefetching:
Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers**

Santhosh Srinath[†‡] Onur Mutlu[§] Hyesoon Kim[‡] Yale N. Patt[‡]

[†]Microsoft
ssri@microsoft.com

[§]Microsoft Research
onur@microsoft.com

[‡]Department of Electrical and Computer Engineering
The University of Texas at Austin
{santhosh, hyesoon, patt}@ece.utexas.edu

Outline

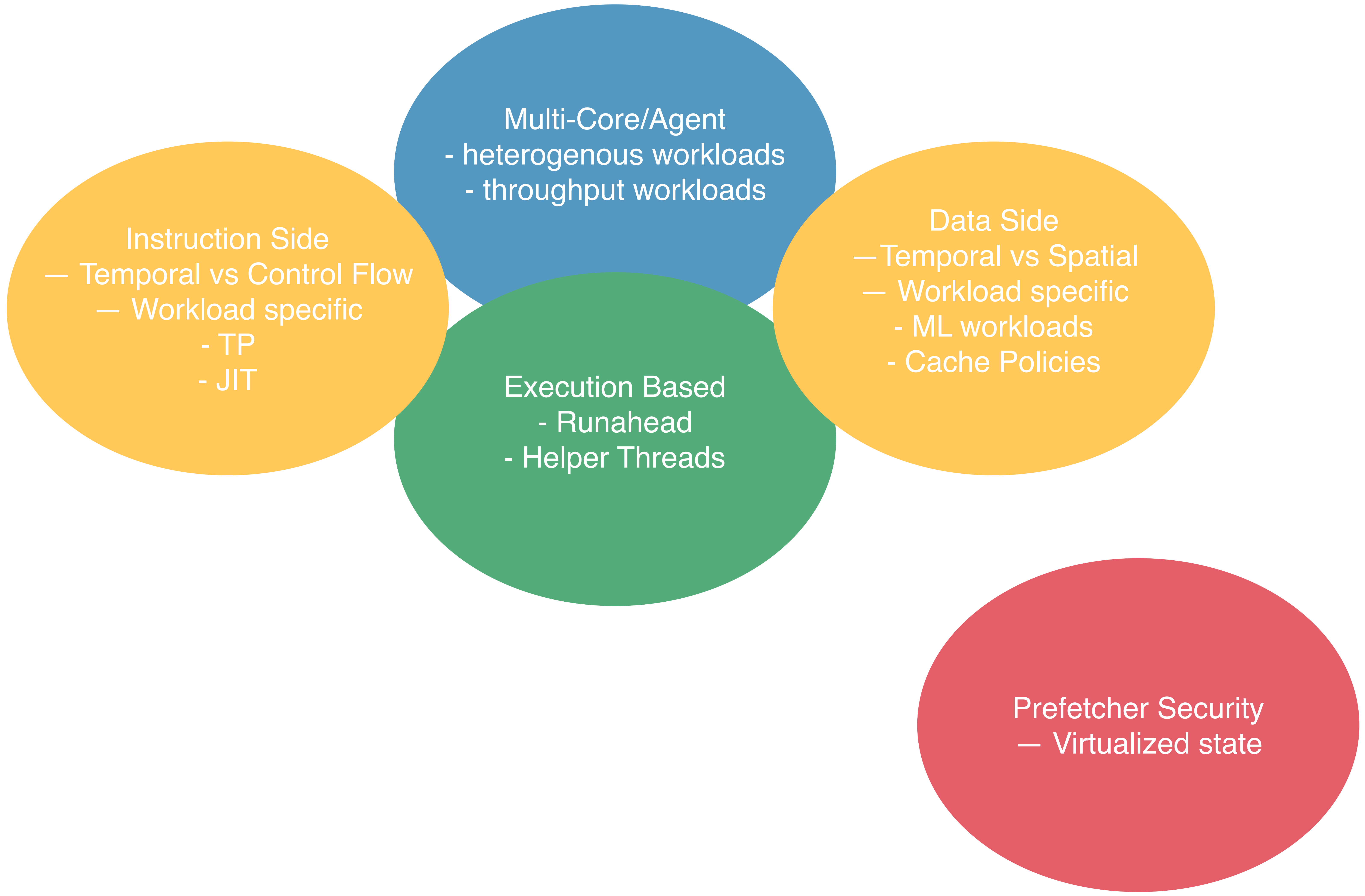
Motivation

Instruction prefetching

Data prefetching

Research directions

Directions in Prefetcher Design Exploration


Multi-Core/Agent
- heterogenous workloads
- throughput workloads
Instruction Side
— Temporal vs Control Flow
— Workload specific
- TP
- JIT
Data Side
—Temporal vs Spatial
— Workload specific
- ML workloads
- Cache Policies
Execution Based
- Runahead
- Helper Threads
Prefetcher Security
— Virtualized state