

18-447: Computer Architecture

Lecture 29: Prefetching (and Exam Review)

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2012, 4/15/2013

Midterm II this Wednesday

- April 17, in class, 12:30-2:20pm
- Similar format and spirit as Midterm I
- Suggestion: Do Homework 6 to prepare for the Midterm
- All topics we have covered so far can be included
- Much of the focus will be on topics after Midterm I
- Two cheat sheets allowed
- Please come to class on time: 12:25pm

Suggestions

- Solve past midterms (and finals) on your own...
 - And, check your solutions vs. the online solutions
 - Questions will be similar in spirit
- <http://www.ece.cmu.edu/~ece447/s12/doku.php?id=wiki:exams>
- <http://www.ece.cmu.edu/~ece447/s12/lib/exe/fetch.php?media=wiki:18447-midterm2.pdf>
- <http://www.ece.cmu.edu/~ece447/s12/lib/exe/fetch.php?media=wiki:18447-final.pdf>

Suggestions

- Attend office hours tomorrow
- Get help from Justin, Yoongu, Jason

Last Lecture

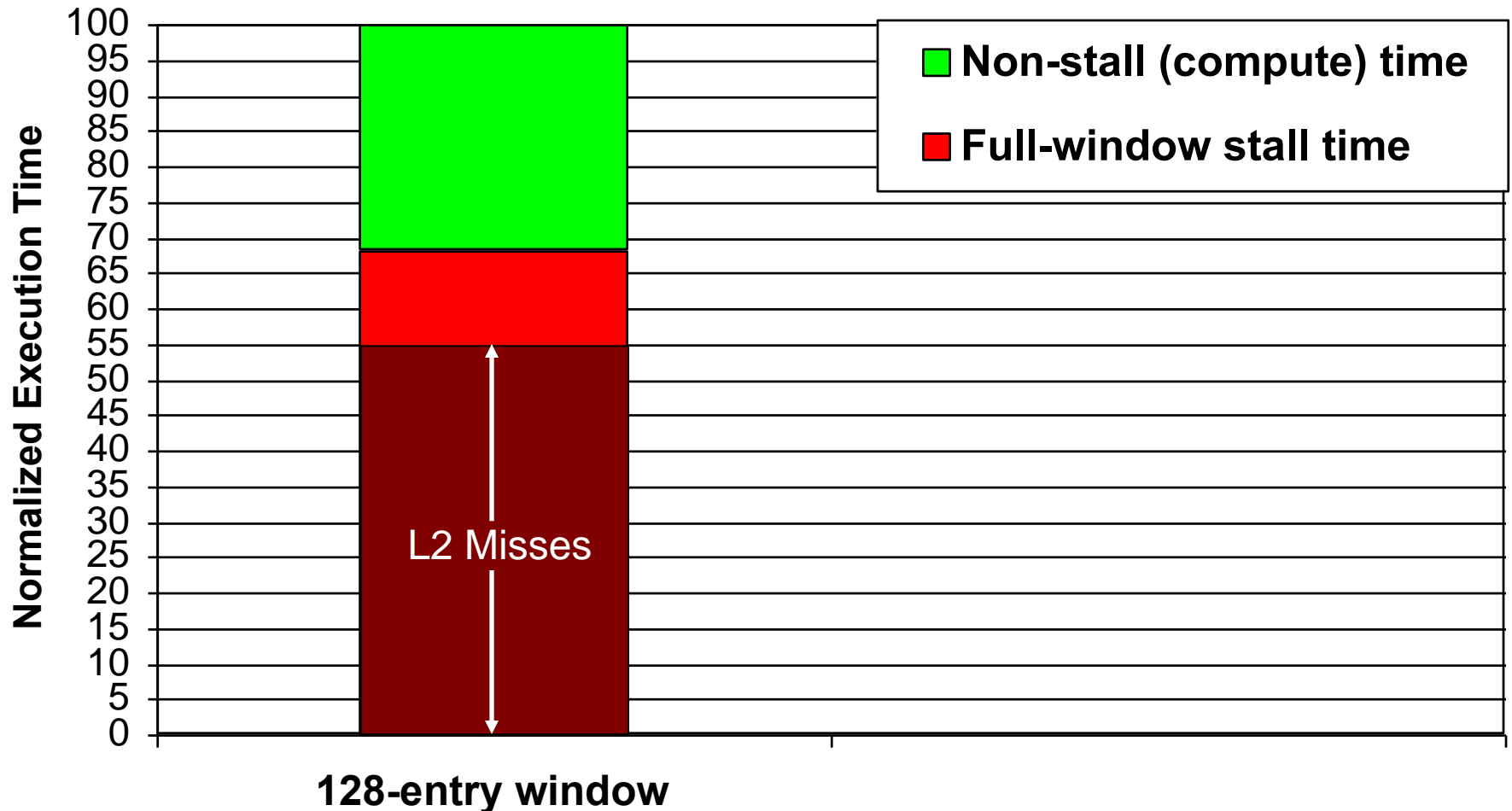
- Wrap up Memory Interference Handling Techniques
- Start Memory Latency Tolerance
- Runahead Execution and Enhancements
 - Efficient Runahead Execution
 - Address-Value Delta Prediction

Today

- Basics of Prefetching
- If time permits and if there are questions: Midterm review
 - Please ask questions

Tolerating Memory Latency

Cache Misses Responsible for Many Stalls



512KB L2 cache, 500-cycle DRAM latency, aggressive stream-based prefetcher
Data averaged over 147 memory-intensive benchmarks on a high-end x86 processor model

Review: Memory Latency Tolerance Techniques

- **Caching** [initially by Wilkes, 1965]
 - Widely used, simple, effective, but inefficient, passive
 - Not all applications/phases exhibit temporal or spatial locality
- **Prefetching** [initially in IBM 360/91, 1967]
 - Works well for regular memory access patterns
 - Prefetching irregular access patterns is difficult, inaccurate, and hardware-intensive
- **Multithreading** [initially in CDC 6600, 1964]
 - Works well if there are multiple threads
 - Improving single thread performance using multithreading hardware is an ongoing research effort
- **Out-of-order execution** [initially by Tomasulo, 1967]
 - Tolerates irregular cache misses that cannot be prefetched
 - Requires extensive hardware resources for tolerating long latencies
 - **Runahead execution** alleviates this problem (as we will see in a later lecture)

Prefetching

Outline of Prefetching Lectures

- Why prefetch? Why could/does it work?
- The four questions
 - What (to prefetch), when, where, how
- Software prefetching
- Hardware prefetching algorithms
- Execution-based prefetching
- Prefetching performance
 - Coverage, accuracy, timeliness
 - Bandwidth consumption, cache pollution
- Prefetcher throttling (if we get to it)
- Issues in multi-core (if we get to it)

Prefetching

- Idea: Fetch the data before it is needed (i.e. pre-fetch) by the program
- Why?
 - ❑ Memory latency is high. If we can prefetch accurately and early enough we can reduce/eliminate that latency.
 - ❑ Can eliminate compulsory cache misses
 - ❑ Can it eliminate all cache misses? Capacity, conflict?
- Involves predicting which address will be needed in the future
 - ❑ Works if programs have predictable miss address patterns

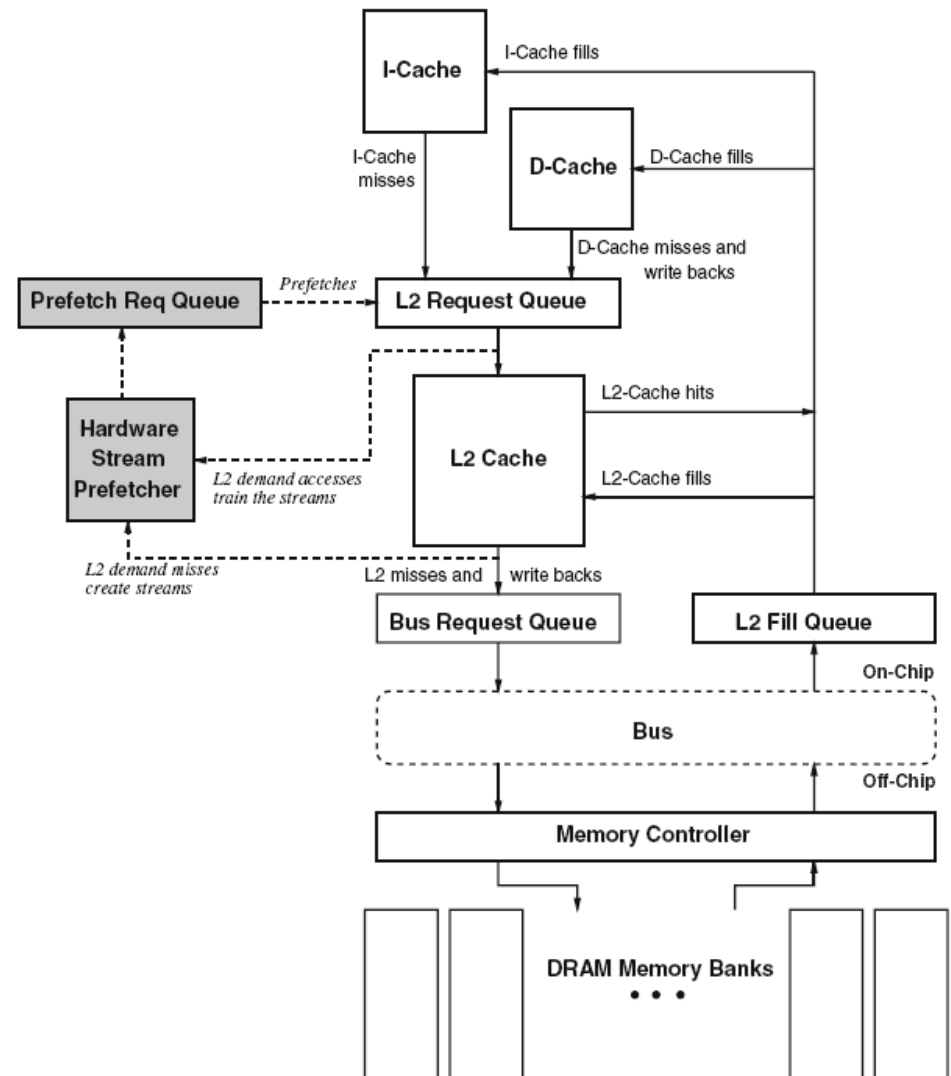
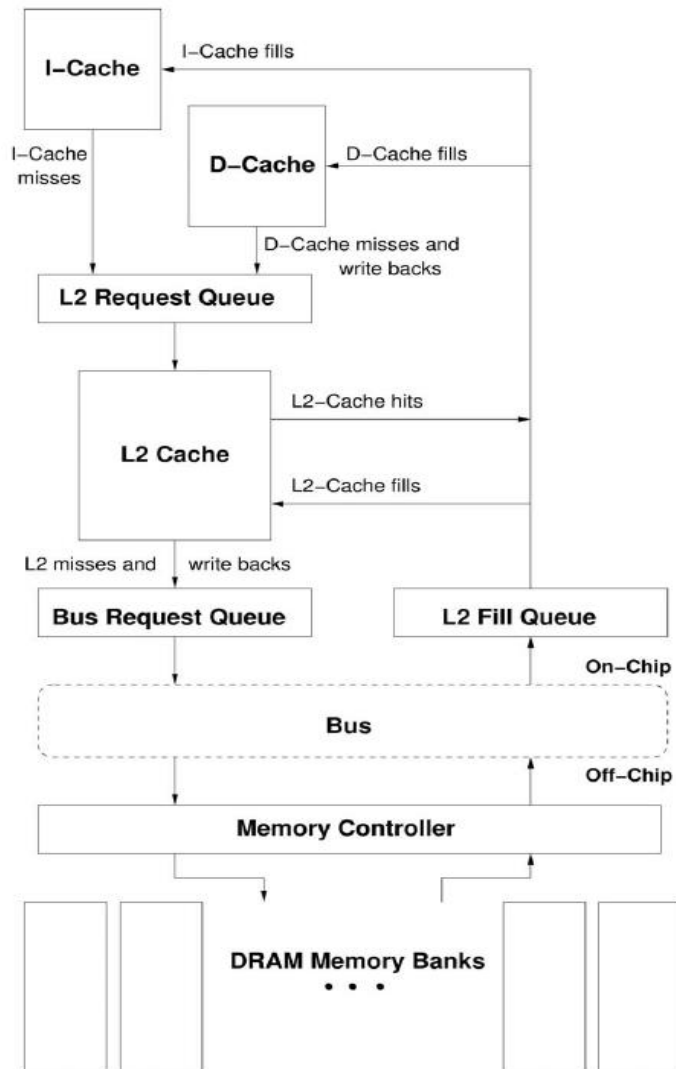
Prefetching and Correctness

- Does a misprediction in prefetching affect correctness?
- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery
- In contrast to branch misprediction or value misprediction

Basics

- In modern systems, prefetching is usually done in cache block granularity
- Prefetching is a technique that can reduce both
 - Miss rate
 - Miss latency
- Prefetching can be done by
 - hardware
 - compiler
 - programmer

How a HW Prefetcher Fits in the Memory System



Prefetching: The Four Questions

- What
 - **What** addresses to prefetch
- When
 - **When** to initiate a prefetch request
- Where
 - **Where** to place the prefetched data
- How
 - Software, hardware, execution-based, cooperative

Challenges in Prefetching: What

- **What** addresses to prefetch
 - Prefetching useless data wastes resources
 - Memory bandwidth
 - Cache or prefetch buffer space
 - Energy consumption
 - These could all be utilized by demand requests or more accurate prefetch requests
 - **Accurate** prediction of addresses to prefetch is important
 - Prefetch accuracy = used prefetches / sent prefetches
- **How do we know what to prefetch**
 - Predict based on past access patterns
 - Use the compiler's knowledge of data structures
- **Prefetching algorithm** determines what to prefetch

Challenges in Prefetching: When

- **When** to initiate a prefetch request
 - Prefetching too early
 - Prefetched data might not be used before it is evicted from storage
 - Prefetching too late
 - Might not hide the whole memory latency
- When a data item is prefetched affects the **timeliness** of the prefetcher
- Prefetcher can be made more timely by
 - Making it more **aggressive**: try to stay far ahead of the processor's access stream (hardware)
 - Moving the **prefetch instructions earlier in the code** (software)

Challenges in Prefetching: Where (I)

- **Where** to place the prefetched data
 - In cache
 - + Simple design, no need for separate buffers
 - Can evict useful demand data → cache pollution
 - In a separate **prefetch buffer**
 - + Demand data protected from prefetches → no cache pollution
 - More complex memory system design
 - Where to place the prefetch buffer
 - When to access the prefetch buffer (parallel vs. serial with cache)
 - When to move the data from the prefetch buffer to cache
 - How to size the prefetch buffer
 - Keeping the prefetch buffer coherent
- Many modern systems place prefetched data into the cache
 - Intel Pentium 4, Core2's, AMD systems, IBM POWER4,5,6, ...

Challenges in Prefetching: Where (II)

- Which level of cache to prefetch into?
 - Memory to L2, memory to L1. Advantages/disadvantages?
 - L2 to L1? (a separate prefetcher between levels)

- Where to place the prefetched data in the cache?
 - Do we treat prefetched blocks the same as demand-fetched blocks?
 - Prefetched blocks are not known to be needed
 - With LRU, a demand block is placed into the MRU position

- Do we skew the replacement policy such that it favors the demand-fetched blocks?
 - E.g., place all prefetches into the LRU position in a way?

Challenges in Prefetching: Where (III)

- **Where** to place the hardware prefetcher in the memory hierarchy?
 - ❑ In other words, what access patterns does the prefetcher see?
 - ❑ L1 hits and misses
 - ❑ L1 misses only
 - ❑ L2 misses only
- Seeing a more complete access pattern:
 - + Potentially better **accuracy** and **coverage** in prefetching
 - Prefetcher needs to examine more requests (bandwidth intensive, more ports into the prefetcher?)

Challenges in Prefetching: How

- **Software** prefetching
 - ❑ ISA provides prefetch instructions
 - ❑ Programmer or compiler inserts prefetch instructions (effort)
 - ❑ Usually works well only for “regular access patterns”
- **Hardware** prefetching
 - ❑ Hardware monitors processor accesses
 - ❑ Memorizes or finds patterns/strides
 - ❑ Generates prefetch addresses automatically
- **Execution-based** prefetchers
 - ❑ A “thread” is executed to prefetch data for the main program
 - ❑ Can be generated by either software/programmer or hardware

Software Prefetching (I)

- Idea: Compiler/programmer places prefetch instructions into appropriate places in code
- Mowry et al., “Design and Evaluation of a Compiler Algorithm for Prefetching,” ASPLOS 1992.
- Prefetch instructions prefetch data into caches
- Compiler or programmer can insert such instructions into the program

X86 PREFETCH Instruction

PREFETCHh—Prefetch Data Into Caches


Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
OF 18 /1	PREFETCHT0 <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using T0 hint.
OF 18 /2	PREFETCHT1 <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using T1 hint.
OF 18 /3	PREFETCHT2 <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using T2 hint.
OF 18 /0	PREFETCHNTA <i>m8</i>	Valid	Valid	Move data from <i>m8</i> closer to the processor using NTA hint.

Description


Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
 - Pentium III processor—1st- or 2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
 - Pentium III processor—2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
 - Pentium III processor—2nd-level cache.
 - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
 - Pentium III processor—1st-level cache
 - Pentium 4 and Intel Xeon processors—2nd-level cache

microarchitecture
dependent
specification

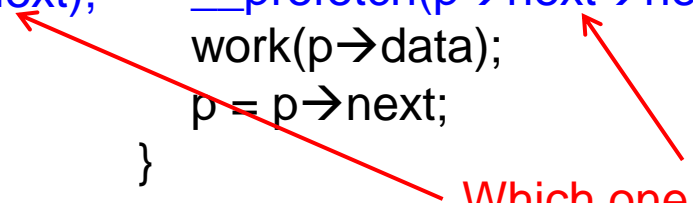


different instructions
for different cache
levels



Software Prefetching (II)

<pre>for (i=0; i<N; i++) { __prefetch(a[i+8]); __prefetch(b[i+8]); sum += a[i]*b[i]; }</pre>	<pre>while (p) { __prefetch(p->next); work(p->data); p = p->next; }</pre>	<pre>while (p) { __prefetch(p->next->next->next); work(p->data); p = p->next; }</pre>
---	--	--



Which one is better?

- Can work for very regular array-based access patterns. Issues:
 - Prefetch instructions take up processing/execution bandwidth
 - **How early to prefetch?** Determining this is difficult
 - Prefetch distance depends on hardware implementation (memory latency, cache size, time between loop iterations) → portability?
 - Going too far back in code reduces accuracy (branches in between)
 - Need “special” prefetch instructions in ISA?
 - Not really. Alpha load into register 31 treated as prefetch (r31==0)
 - PowerPC *dcbt* (data cache block touch) instruction
 - Not easy to do for pointer-based data structures

Software Prefetching (III)

- Where should a compiler insert prefetches?
 - Prefetch for every load access?
 - Too bandwidth intensive (both memory and execution bandwidth)
 - Profile the code and determine loads that are likely to miss
 - What if profile input set is not representative?
 - How far ahead before the miss should the prefetch be inserted?
 - Profile and determine probability of use for various prefetch distances from the miss
 - What if profile input set is not representative?
 - Usually need to insert a prefetch far in advance to cover 100s of cycles of main memory latency → reduced accuracy

Hardware Prefetching (I)

- Idea: Specialized hardware observes load/store access patterns and prefetches data based on past access behavior
- Tradeoffs:
 - + Can be tuned to system implementation
 - + Does not waste instruction execution bandwidth
 - More hardware complexity to detect patterns
 - Software can be more efficient in some cases

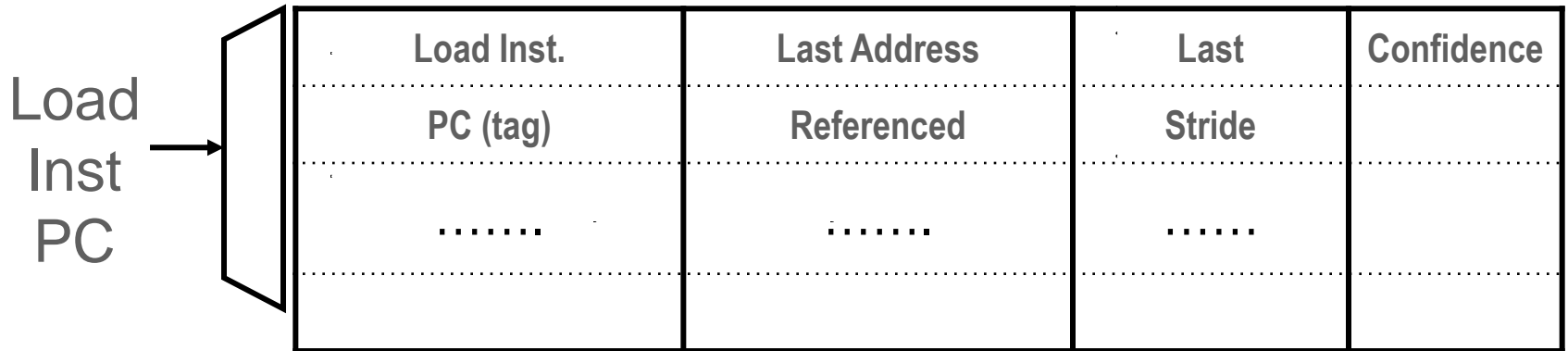
Next-Line Prefetchers

- 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

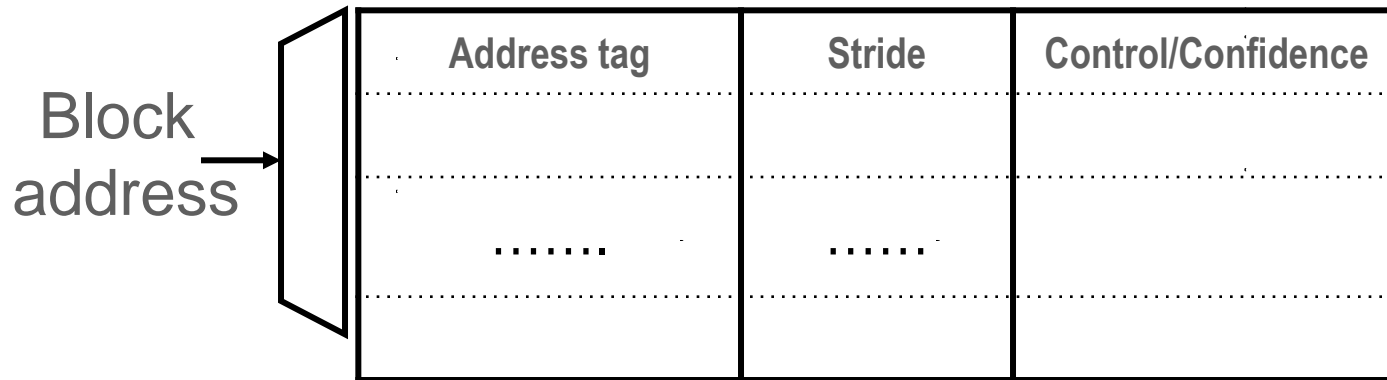
- Two kinds
 - Instruction program counter (PC) based
 - Cache block address based
- Instruction based:
 - Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.
 - Idea:
 - Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
 - Next time the same load instruction is fetched, prefetch $\text{last address} + \text{stride}$

Instruction Based Stride Prefetching



- What is the problem with this?
 - Hint: how far can this get ahead? How much of the miss latency can the prefetch cover?
 - 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
 - Prefetch ahead ($\text{last address} + N \times \text{stride}$)
 - Generate multiple prefetches

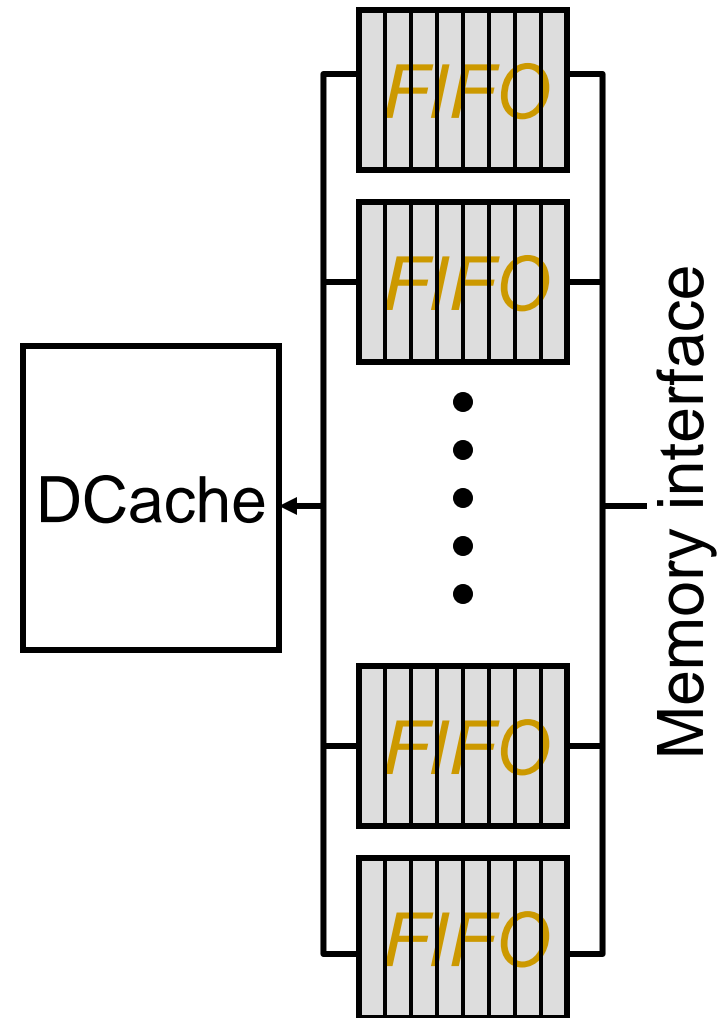
Cache-Block Address Based Stride Prefetching



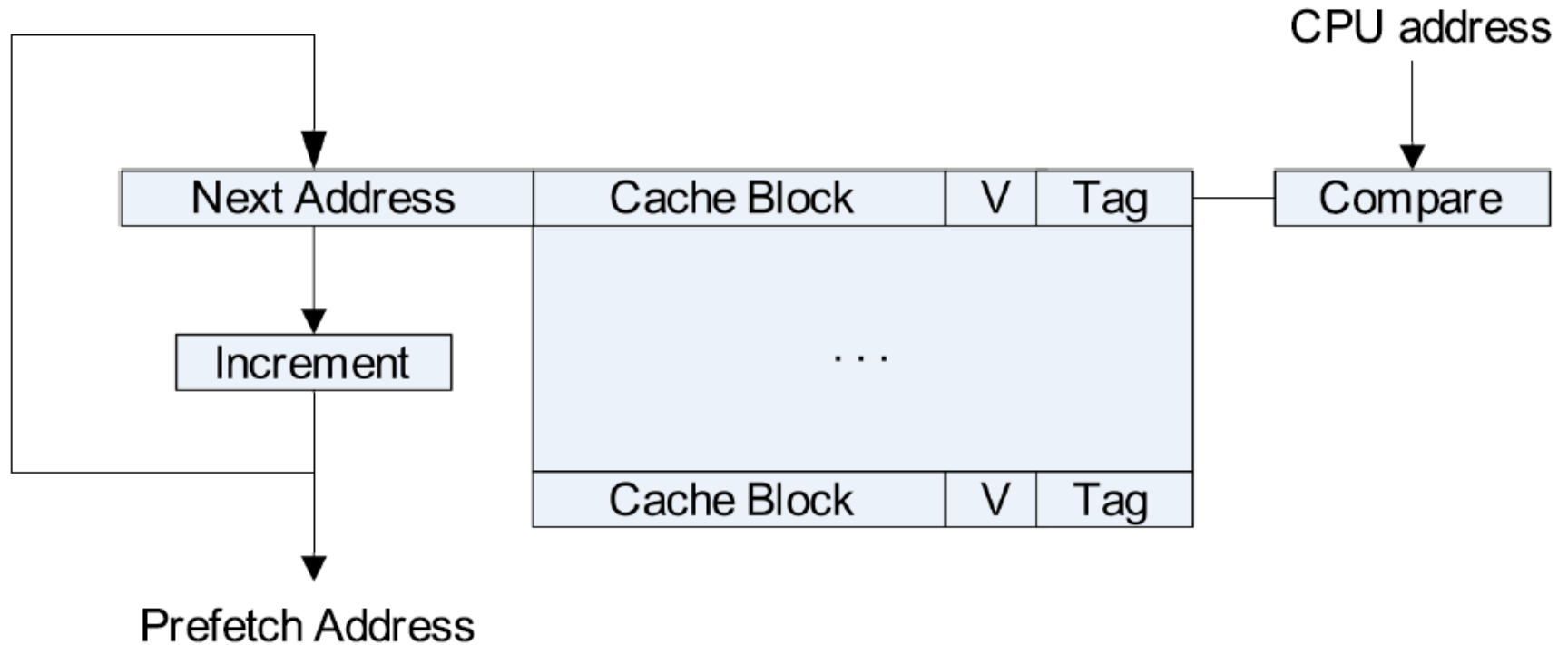
- Can detect
 - $A, A+N, A+2N, A+3N, \dots$
 - **Stream buffers** are a special case of cache block address based stride prefetching where $N = 1$
 - Read the Jouppi paper
 - Stream buffer also has data storage in that paper (no prefetching into cache)

Stream Buffers (Jouppi, ISCA 1990)

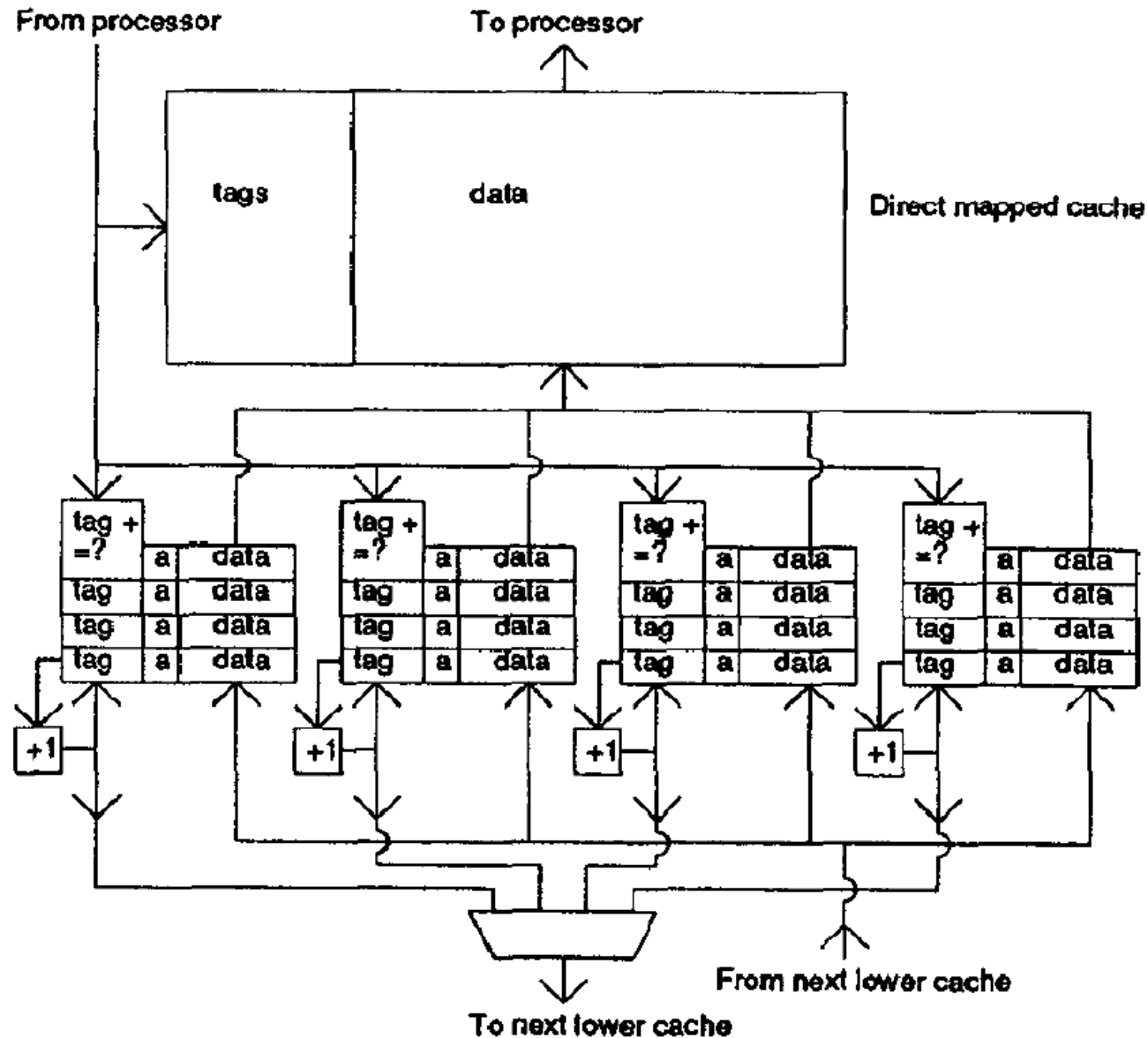
- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
 - if hit, pop the entry from FIFO, update the cache with data
 - if not, allocate a new stream buffer to the new miss address (may have to recycle a stream buffer following LRU policy)
- Stream buffer FIFOs are continuously topped-off with subsequent cache lines whenever there is room and the bus is not busy



Stream Buffer Design



Stream Buffer Design



Prefetcher Performance (I)

- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetches / used prefetches)

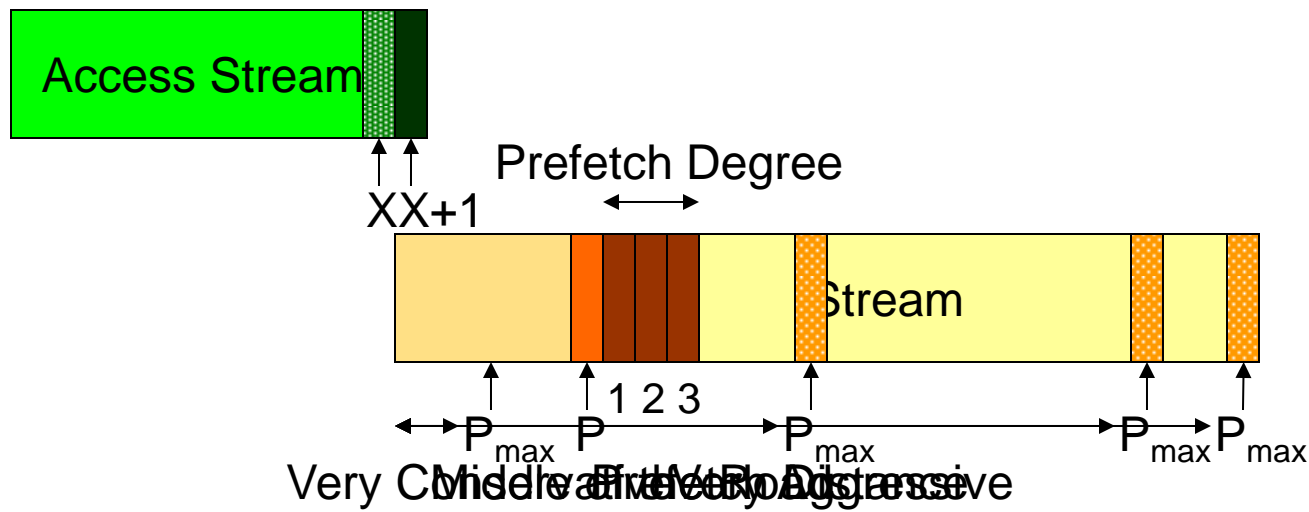
- **Bandwidth consumption**
 - Memory bandwidth consumed with prefetcher / without prefetcher
 - Good news: **Can utilize idle bus bandwidth (if available)**

- **Cache pollution**
 - Extra demand misses due to prefetch placement in cache
 - More difficult to quantify but affects performance

We did not cover the following slides in lecture. They are for your benefit.

Prefetcher Performance (II)

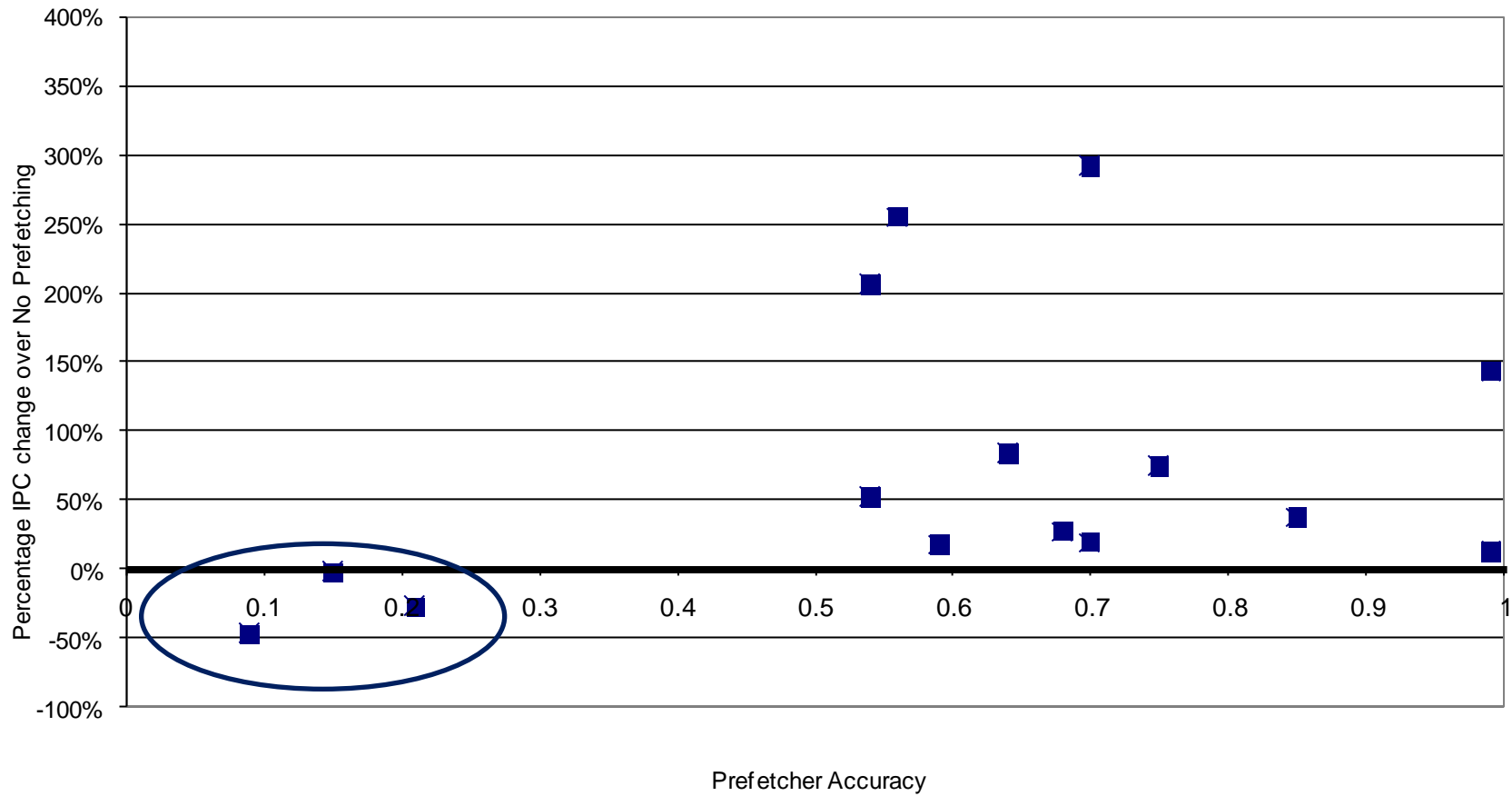
- Prefetcher aggressiveness affects all performance metrics
- Aggressiveness dependent on prefetcher type
- For most hardware prefetchers:
 - **Prefetch distance**: how far ahead of the demand stream
 - **Prefetch degree**: how many prefetches per demand access



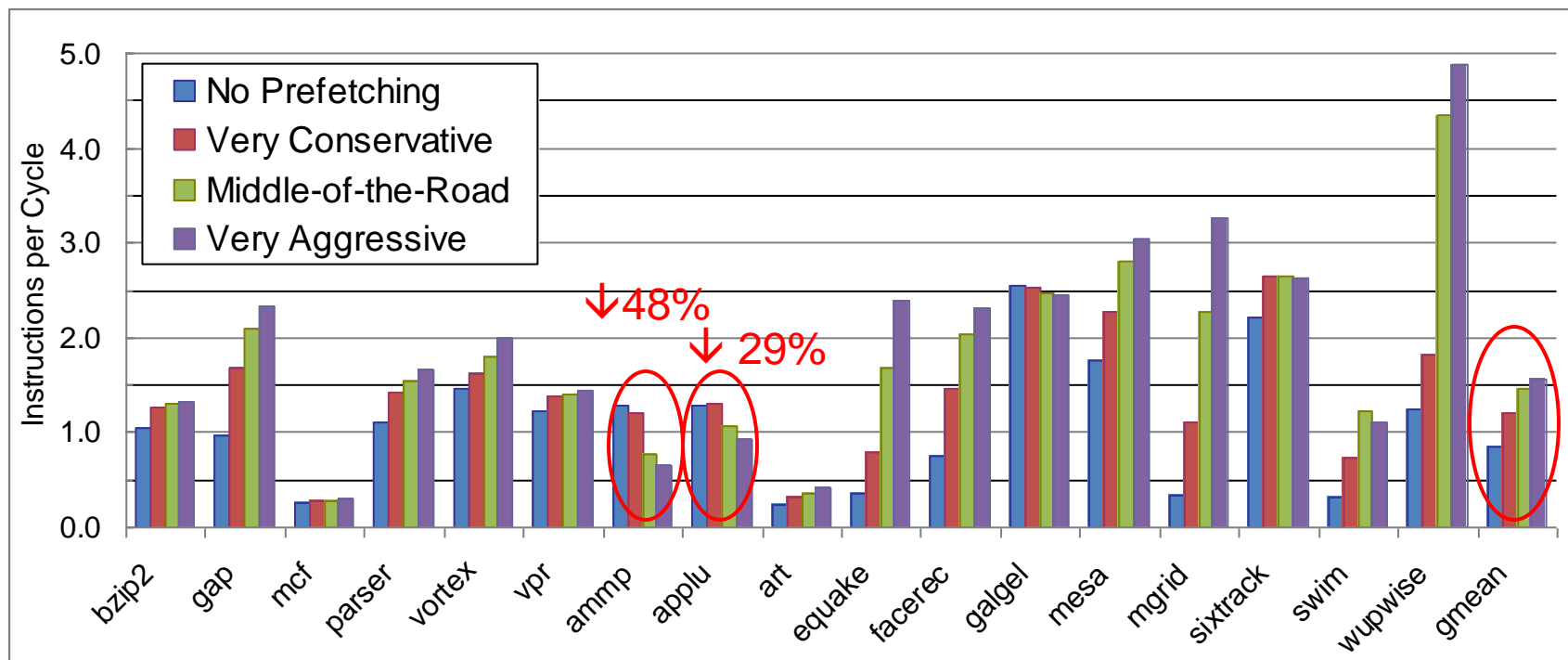
Prefetcher Performance (III)

- How do these metrics interact?
- Very Aggressive
 - Well ahead of the load access stream
 - Hides memory access latency better
 - More speculative
 - + Higher coverage, better timeliness
 - Likely lower accuracy, higher bandwidth and pollution
- Very Conservative
 - Closer to the load access stream
 - Might not hide memory access latency completely
 - Reduces potential for cache pollution and bandwidth contention
 - + Likely higher accuracy, lower bandwidth, less polluting
 - Likely lower coverage and less timely

Prefetcher Performance (IV)



Prefetcher Performance (V)

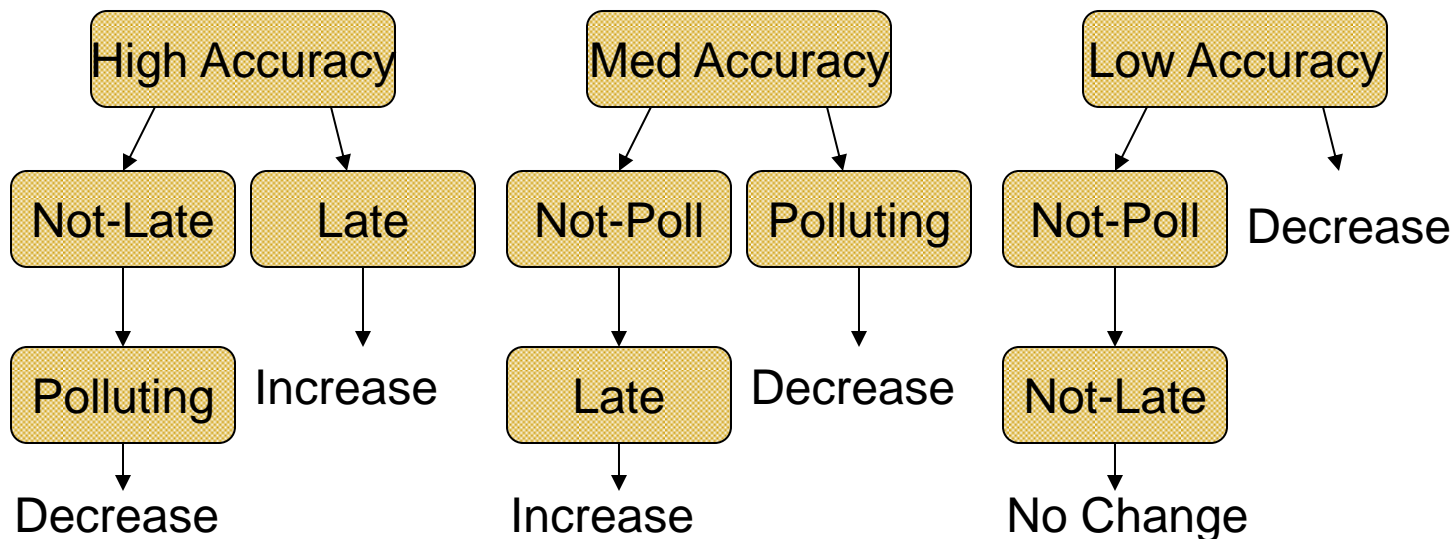


- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

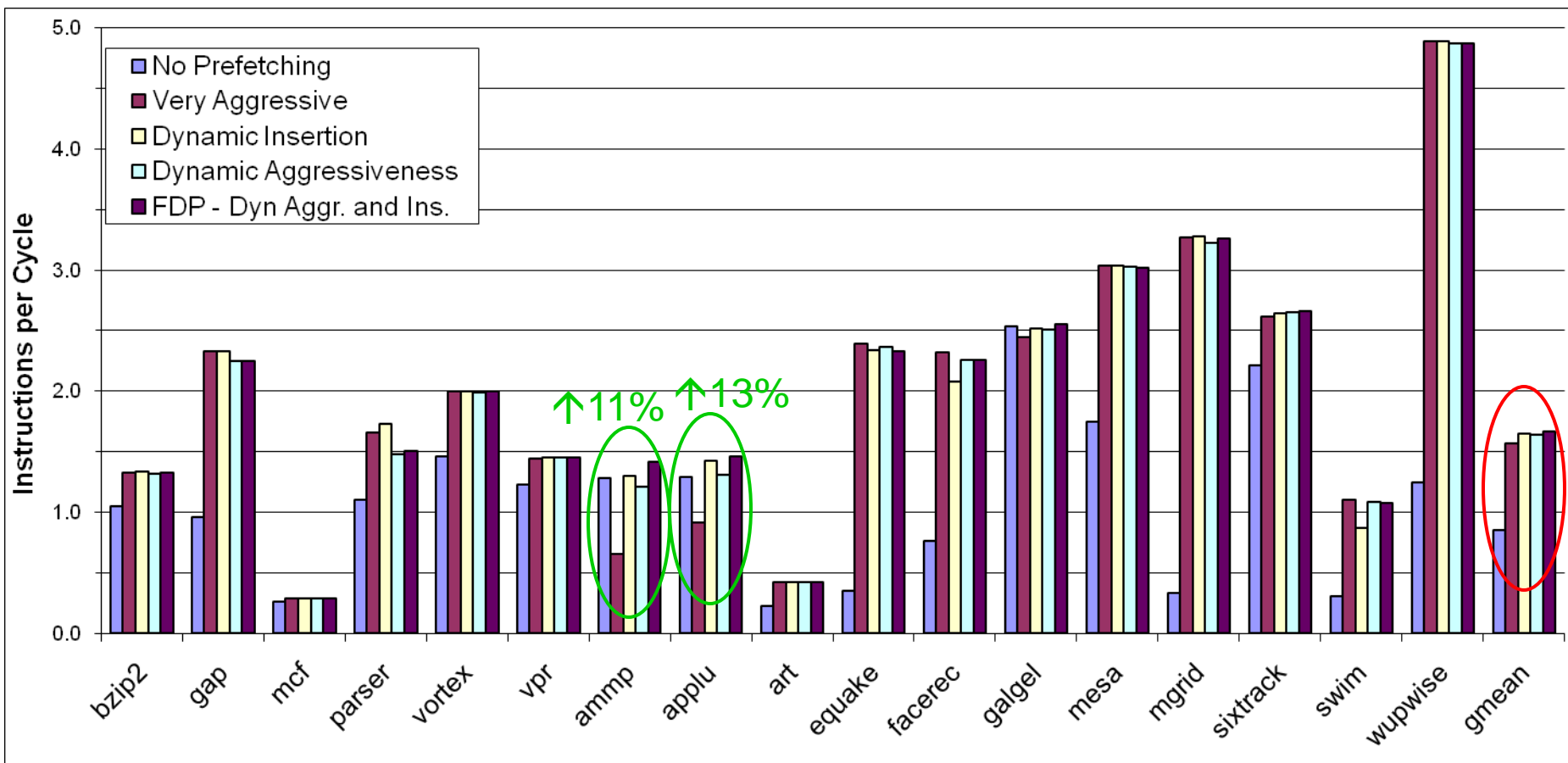
Feedback-Directed Prefetcher Throttling (I)

■ 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 (II)



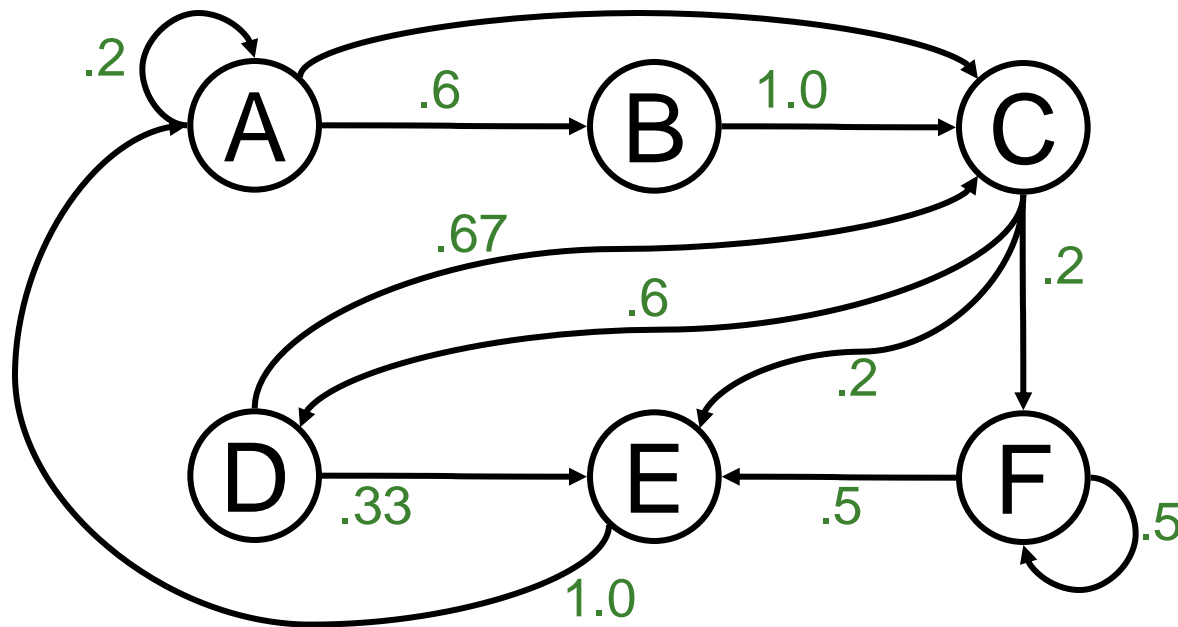
- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

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

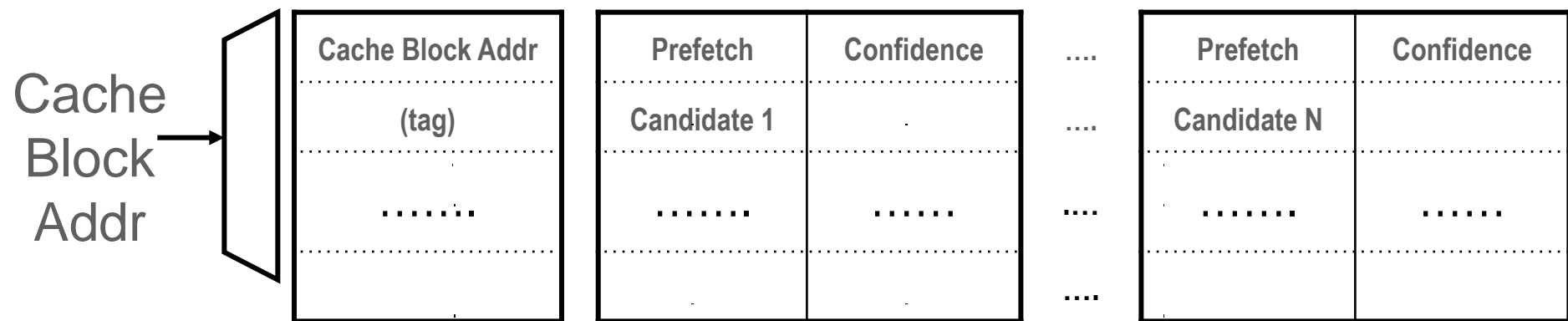
Markov Prefetching (I)

- 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), are some addresses more likely to be referenced next



*Markov
Model*

Markov 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 *accuracy* is generally low so 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.

Markov Prefetching (III)

■ Advantages:

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

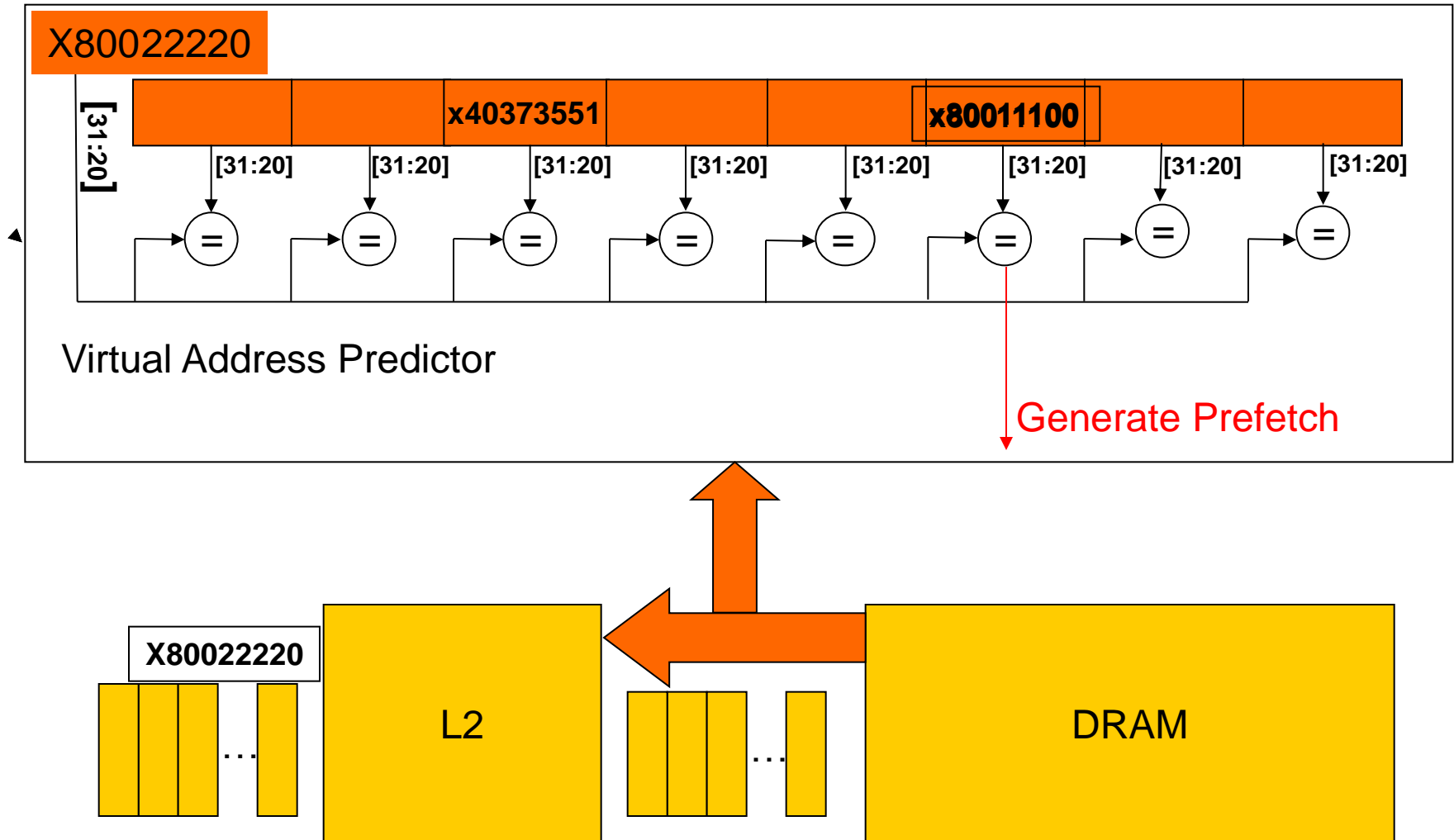
■ Disadvantages:

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

Content Directed Prefetching (I)

- A specialized prefetcher for pointer values
 - Cooksey et al., “A stateless, content-directed data prefetching mechanism,” ASPLOS 2002.
 - Idea: Identify pointers among all values in a fetched cache block and issue prefetch requests for them.
- + No need to memorize/record past addresses!
- + Can eliminate compulsory misses (never-seen pointers)
- Indiscriminately prefetches *all* pointers in a cache block
-
- How to identify pointer addresses:
 - Compare address sized values within cache block with cache block's address → if most-significant few bits match, pointer

Content Directed Prefetching (II)



Making Content Directed Prefetching Efficient

- Hardware does not have enough information on pointers
- Software does (and can profile to get more information)
- Idea:
 - **Compiler** profiles and provides hints as to **which pointer addresses are likely-useful to prefetch.**
 - **Hardware** uses hints **to prefetch only likely-useful pointers.**
- Ebrahimi et al., “**Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems,**” HPCA 2009.

Shortcomings of CDP – An example

```
HashLookup(int Key) {
```

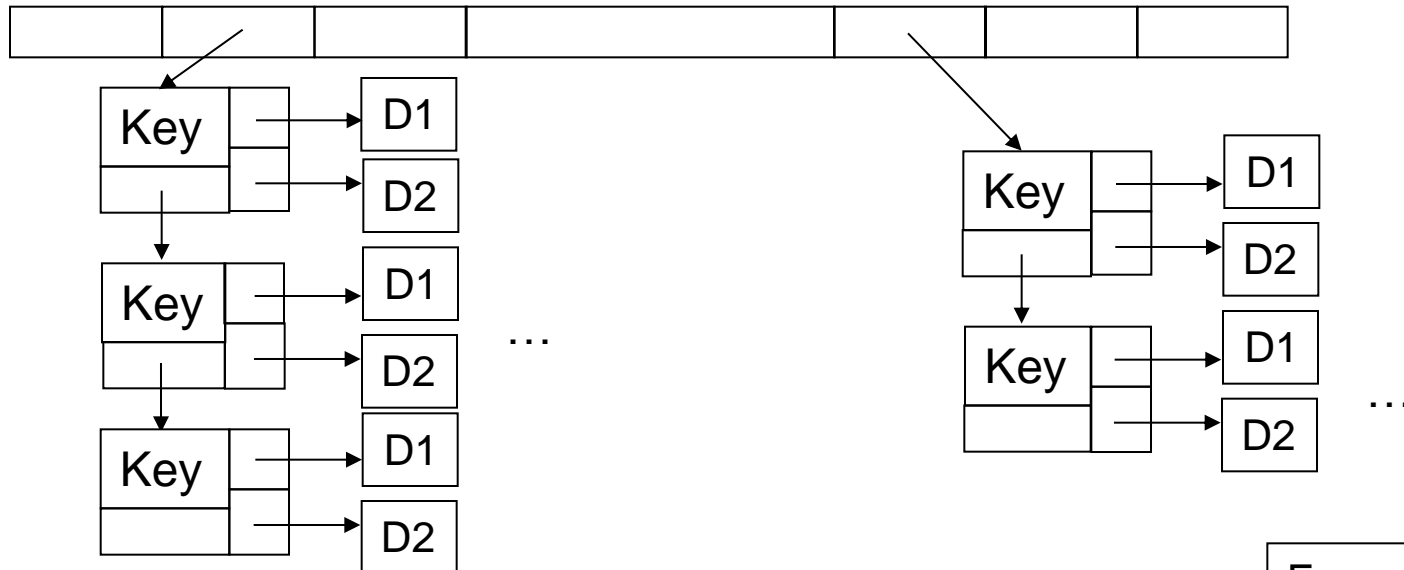
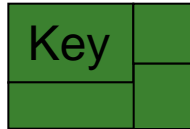
```
...
```

```
for (node = head ; node -> Key != Key; node = node -> Next; ) ;
```

```
if (node) return node->D1;
```

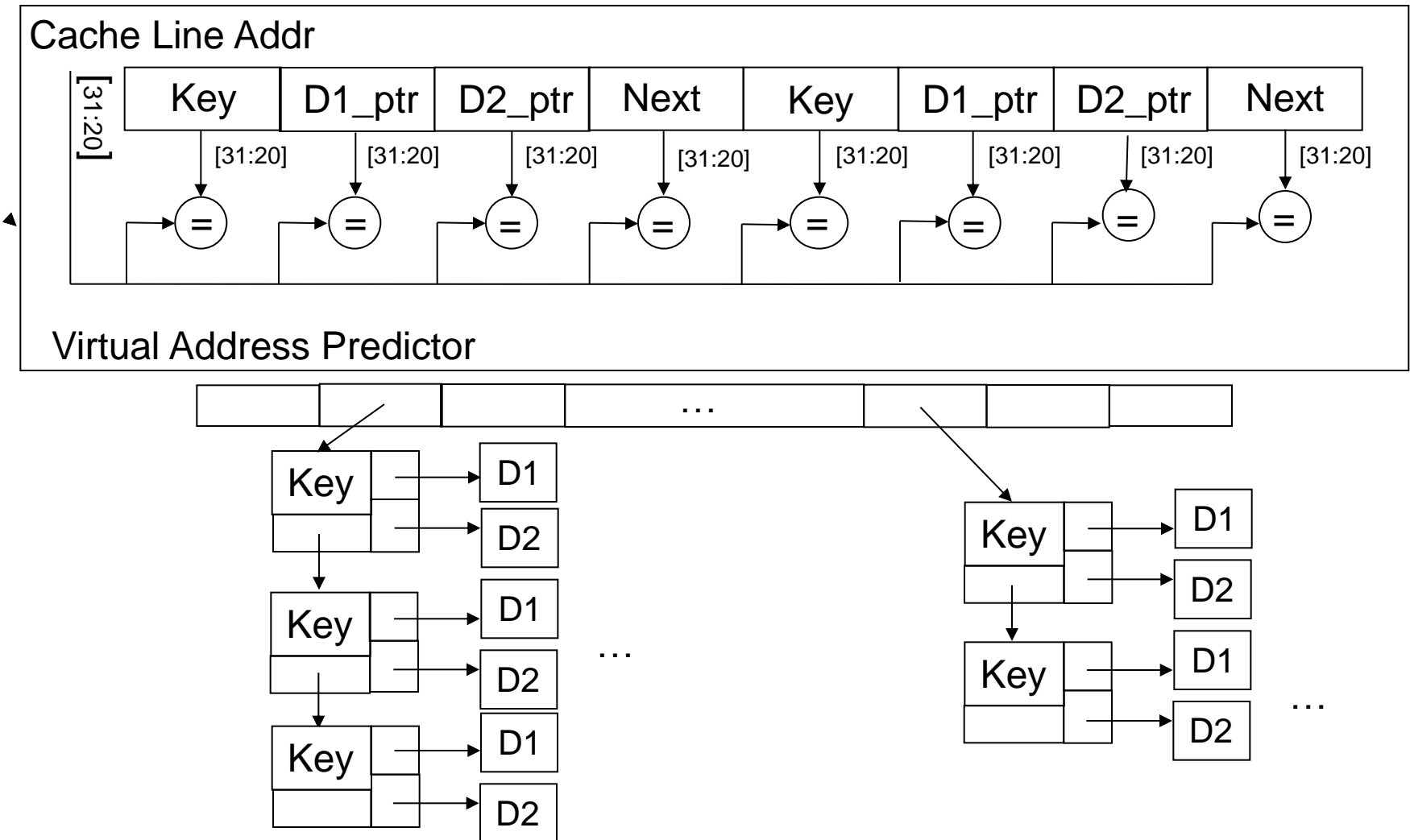
```
}
```

```
Struct node{  
    int Key;  
    int * D1_ptr;  
    int * D2_ptr;  
    node * Next;  
}
```



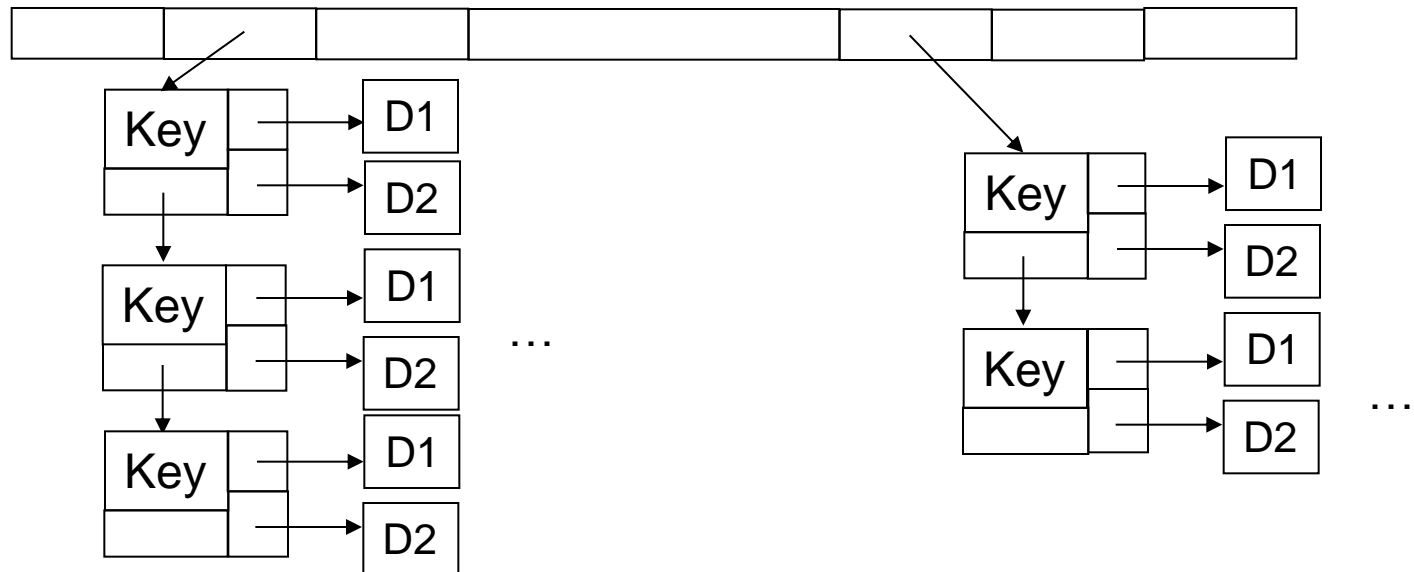
Example from mst

Shortcomings of CDP – An example

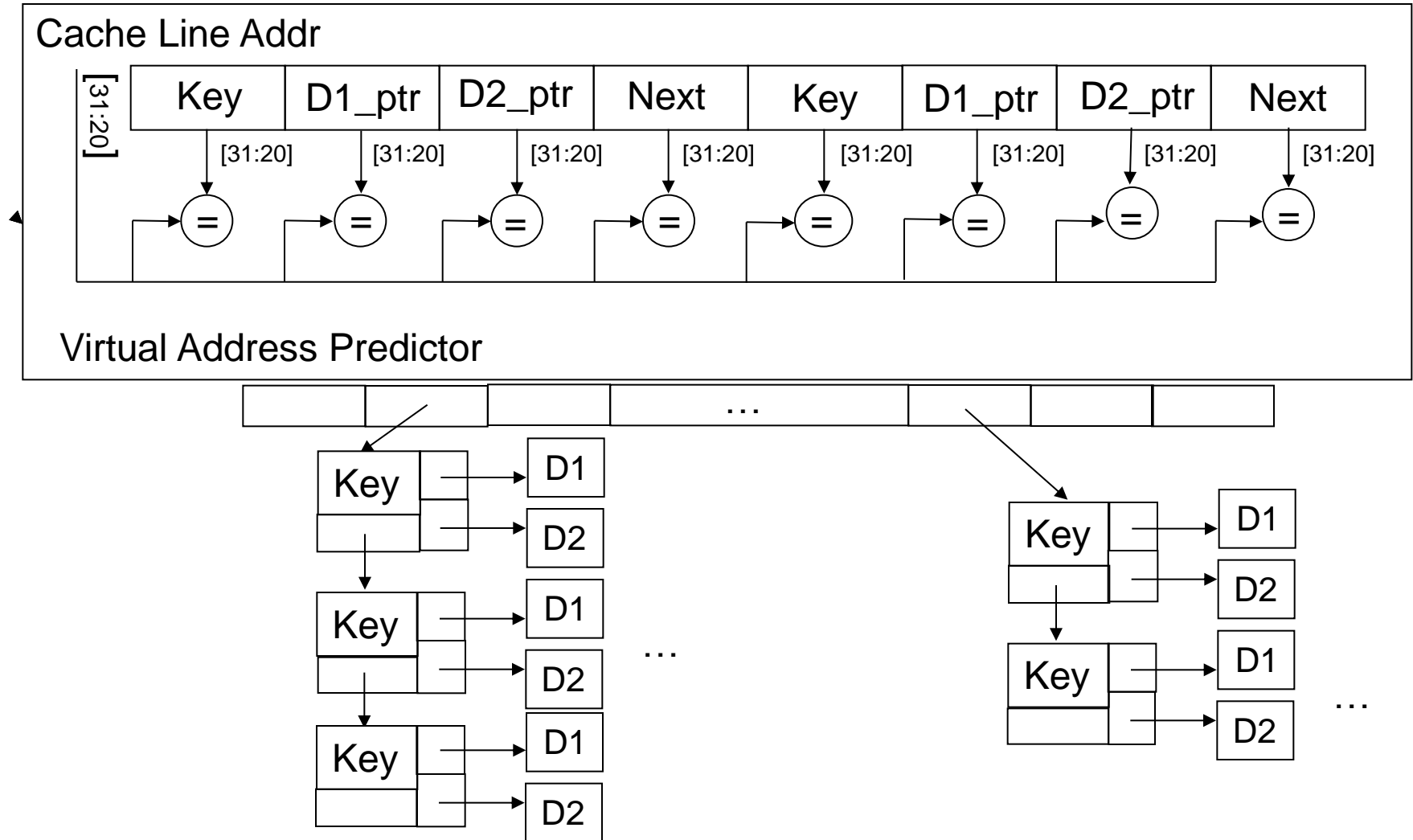


Shortcomings of CDP – An example

```
HashLookup(int Key) {  
    ...  
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;  
    if (node) return node -> D1;  
}
```



Shortcomings of CDP – An example



Execution-based Prefetchers (I)

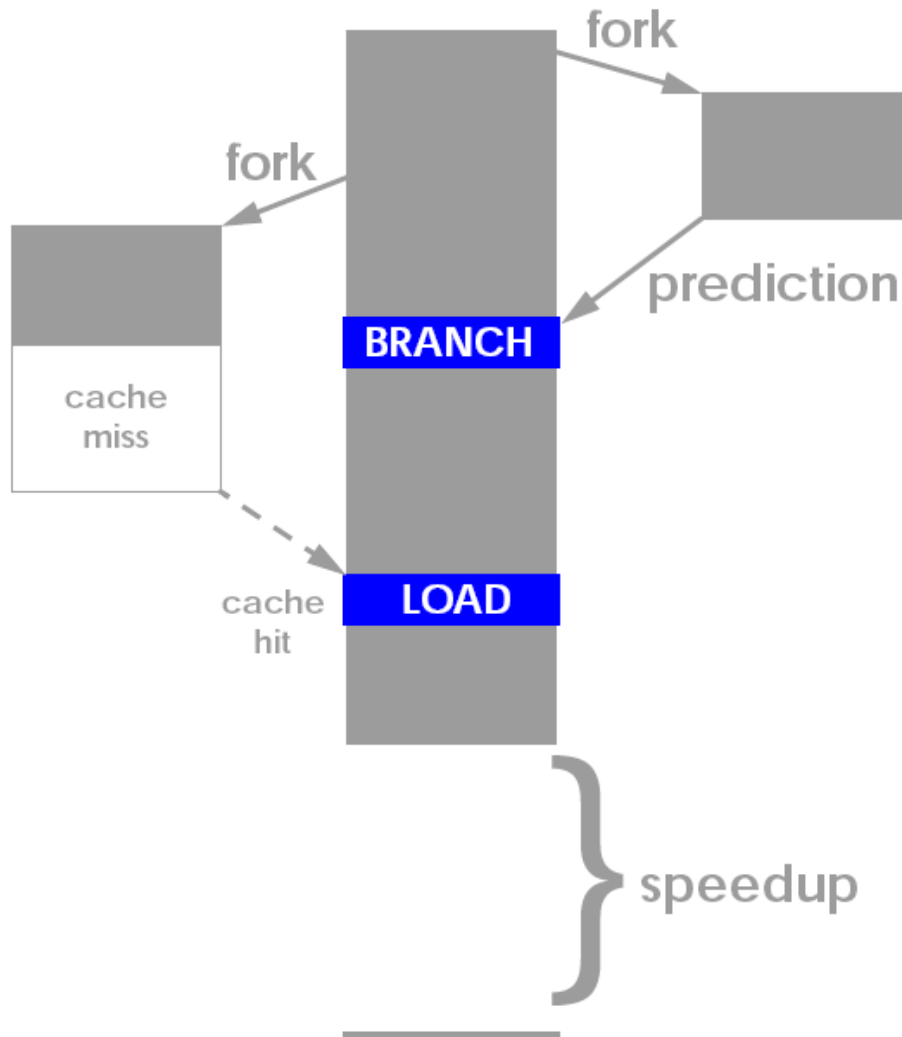
- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
 - Only need to distill pieces that lead to cache misses
- **Speculative thread:** Pre-executed program piece can be considered a “thread”
- Speculative thread can be executed
 - On a separate processor/core
 - On a separate 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)

Thread-Based Pre-Execution



- Dubois and Song, “**Assisted Execution**,” USC Tech Report 1998.
- Chappell et al., “**Simultaneous Subordinate Microthreading (SSMT)**,” ISCA 1999.
- Zilles and Sohi, “**Execution-based Prediction Using Speculative Slices**”, ISCA 2001.

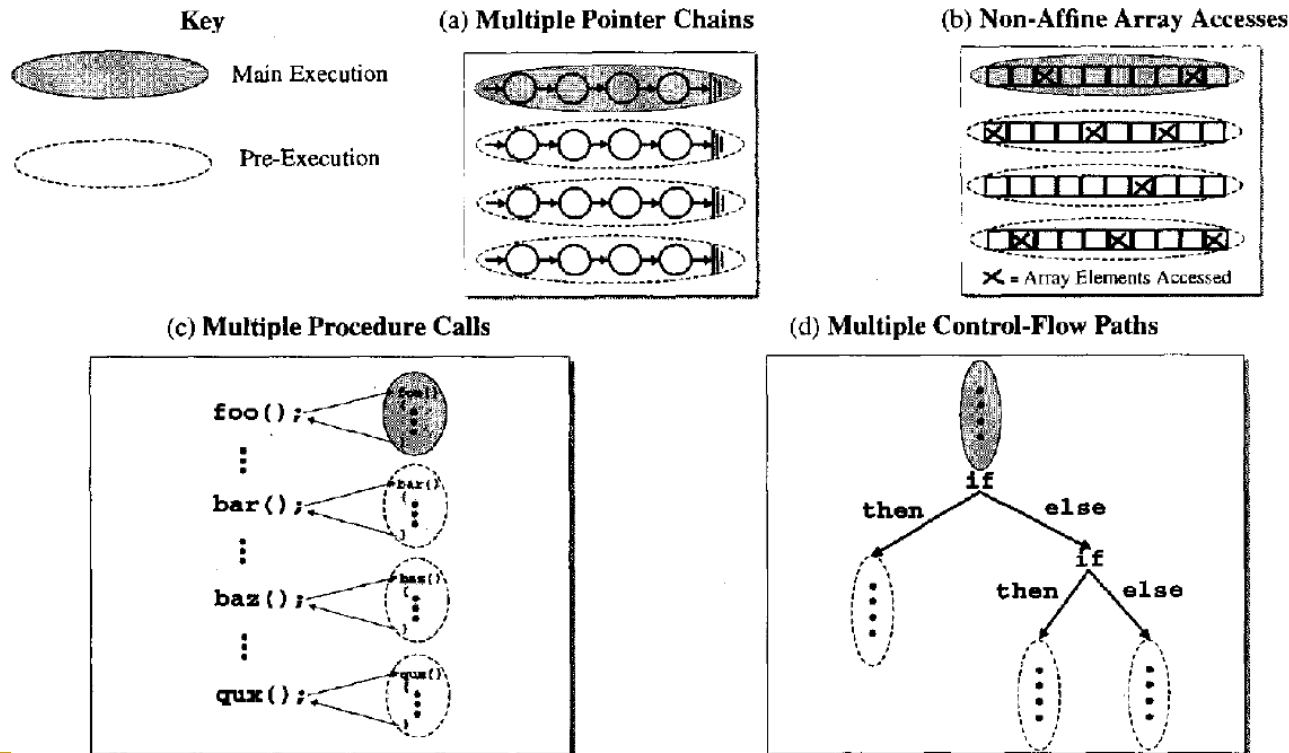
Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?
 1. Separate core (least contention with main thread)
 2. Separate thread context on the same core (more contention)
 3. Same core, same context
 - When the main thread is stalled
- When to spawn the precomputation thread?
 1. Insert spawn instructions well before the “problem” load
 - How far ahead?
 - Too early: prefetch might not be needed
 - Too late: prefetch might not be timely
 2. When the main thread is stalled
- When to terminate the precomputation thread?
 1. With pre-inserted CANCEL instructions
 2. Based on effectiveness/contention feedback

Thread-Based Pre-Execution Issues

■ Read

- Luk, “Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” ISCA 2001.
- Many issues in software-based pre-execution discussed



An Example

(a) Original Code

```
register int i;
register arc_t *arcout;
for( i < trips; ){
    // loop over 'trips' lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    arcin = (arc_t *)first_of_sparse_list
        → tail → mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin → tail;
        ...
        arcin = (arc_t *)tail → mark;
    }
    i++, arcout += 3;
}
```

(b) Code with Pre-Execution

```
register int i;
register arc_t *arcout;
for( i < trips; ){
    // loop over 'trips' lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    // invoke a pre-execution starting
    // at END_FOR
    PreExecute_Start(END_FOR);
    arcin = (arc_t *)first_of_sparse_list
        → tail → mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin → tail;
        ...
        arcin = (arc_t *)tail → mark;
    }
    // terminate this pre-execution after
    // prefetching the entire list
    PreExecute_Stop();
END_FOR:
    // the target address of the pre-
    // execution
    i++, arcout += 3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

The Spec2000 benchmark `mcf` spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer `first_of_sparse_list`, whose value is in fact determined by `arcout`, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where `arcout` gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T , starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T 's registers that hold the values of `i` and `arcout` will be updated. Then `i`'s value is compared against `trips` to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop()**, which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop()**. Notice that any **PreExecute_Start()** instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop()** instructions cannot terminate the main thread either.

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark `mcf`. Loads that incur many cache misses are underlined.

Example ISA Extensions

Thread_ID = PreExecute_Start(Start_PC, Max_Insts):

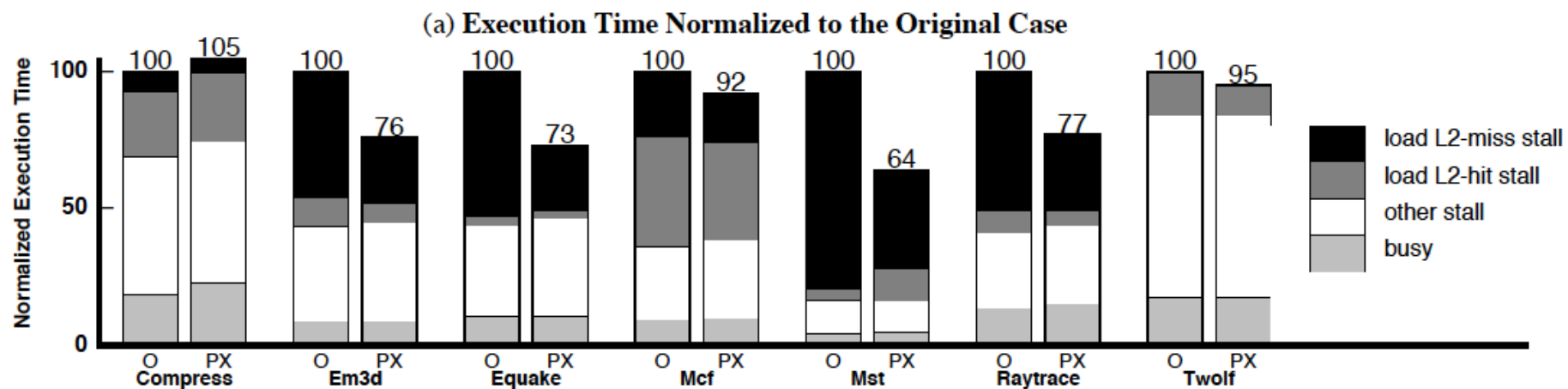
Request for an idle context to start pre-execution at *Start_PC* and stop when *Max_Insts* instructions have been executed; *Thread_ID* holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel(Thread_ID): Terminate the pre-execution thread with *Thread_ID*. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

Results on an SMT Processor



Problem Instructions

- Zilles and Sohi, “Execution-based Prediction Using Speculative Slices”, ISCA 2001.
- Zilles and Sohi, “Understanding the backward slices of performance degrading instructions,” ISCA 2000.

Figure 2. Example problem instructions from heap insertion routine in *vpr*.

```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

void add_to_heap (struct s_heap *hptr) {
    ...
1.  heap[heap_tail] = hptr;
2.  int ifrom = heap_tail;
3.  int ito = ifrom/2;
4.  heap_tail++;
5.  while ((ito >= 1) &&
6.         (heap[ifrom]->cost < heap[ito]->cost))
7.      struct s_heap *temp_ptr = heap[ito];
8.      heap[ito] = heap[ifrom];
9.      heap[ifrom] = temp_ptr;
10.     ifrom = ito;
11.     ito = ifrom/2;
    }
}
```

branch misprediction (points to line 6)

cache miss (points to line 7)

Fork Point for Prefetching Thread

Figure 3. The **node_to_heap** function, which serves as the fork point for the slice that covers **add_to_heap**.

```
void node_to_heap (... , float cost, ...) {  
    struct s_heap *hptr; ← fork point  
    ...  
    hptr = alloc_heap_data();  
    hptr->cost = cost;  
    ...  
    add_to_heap (hptr);  
}
```


Pre-execution Slice Construction

Figure 4. Alpha assembly for the `add_to_heap` function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.

```
node_to_heap:
... /* skips ~40 instructions */
2  lda    s1, 252(gp)    # &heap_tail
2  ldl    t2, 0(s1)      # ifrom = heap_tail
1  ldq    t5, -76(s1)    # &heap[0]
3  cmplt  t2, 0, t4      # see note
4  addl   t2, 0x1, t6    # heap_tail ++
1  s8addq t2, t5, t3      # &heap[heap_tail]
4  stl    t6, 0(s1)      # store heap_tail
1  stq    s0, 0(t3)      # heap[heap_tail]
3  addl   t2, t4, t4      # see note
3  sra    t4, 0x1, t4     # ito = ifrom/2
5  ble    t4, return     # (ito < 1)
loop:
6  s8addq t2, t5, a0      # &heap[ifrom]
6  s8addq t4, t5, t7      # &heap[ito]
11 cmplt  t4, 0, t9       # see note
10 move   t4, t2          # ifrom = ito
6  ldq    a2, 0(a0)       # heap[ifrom]
6  ldq    a4, 0(t7)       # heap[ito]
11 addl   t4, t9, t9       # see note
11 sra    t9, 0x1, t4      # ito = ifrom/2
6  lds    $f0, 4(a2)      # heap[ifrom]->cost
6  lds    $f1, 4(a4)      # heap[ito]->cost
6  cmpltlt $f0,$f1,$f0     # (heap[ifrom]->cost
6  fbeq   $f0, return     # < heap[ito]->cost)
8  stq    a2, 0(t7)       # heap[ito]
9  stq    a4, 0(a0)       # heap[ifrom]
5  bgt    t4, loop        # (ito >= 1)
return:
... /* register restore code & return */
```

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.

Figure 5. Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.

```
slice:
1  ldq    $6, 328(gp)    # &heap
2  ldl    $3, 252(gp)    # ito = heap_tail
slice_loop:
3,11 sra   $3, 0x1, $3    # ito /= 2
6  s8addq $3, $6, $16     # &heap[ito]
6  ldq    $18, 0($16)     # heap[ito]
6  lds    $f1, 4($18)     # heap[ito]->cost
6  cmptle $f1,$f17,$f31   # (heap[ito]->cost
                          # < cost) PRED
                          br    slice_loop

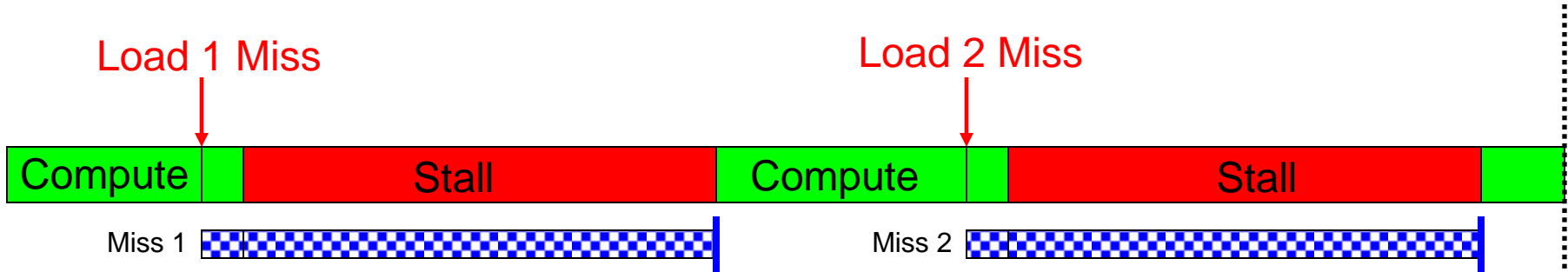
## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```


Runahead Execution (I)

- A simple pre-execution method for prefetching purposes
- When the oldest instruction is a long-latency cache miss:
 - Checkpoint architectural state and enter runahead mode
- In runahead mode:
 - Speculatively pre-execute instructions
 - The purpose of pre-execution is to generate prefetches
 - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
 - Checkpoint is restored and normal execution resumes
- Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

Runahead Execution (Mutlu et al., HPCA 2003)

Small Window:



Runahead:

