

15-740/18-740
Computer Architecture
Lecture 24: Prefetching

Prof. Onur Mutlu
Carnegie Mellon University
Fall 2011, 11/11/11

Announcements

- Milestone II presentations on Monday Nov 14 in class
 - 12 minutes per group
 - 7 minute presentation
 - 5 minute feedback
 - Include your latest results after the submission of milestone
 - Make sure the problem and key ideas are very clear and the presentation flows
 - Submit your slides after lecture to the TAs

- Milestone III – stay tuned
 - Keep working hard on your project

Review Sets 14 and 15

- Due Wednesday, Nov 16
 - Kim et al., “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,” MICRO 2010.

- Due Friday Nov 18
 - Chappell et al., “Simultaneous Subordinate Microthreading (SSMT),” ISCA 1999.

Last Time

- Caching
- Virtual memory
- Interaction between virtual memory and caching

Today and the Next Few Days: Prefetching

- 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
- Issues in multi-core

Readings in Prefetching

■ Required:

- ❑ Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” ISCA 1990.
- ❑ Joseph and Grunwald, “Prefetching using Markov Predictors,” ISCA 1997.

■ Recommended:

- ❑ Mowry et al., “Design and Evaluation of a Compiler Algorithm for Prefetching,” ASPLOS 1992.
- ❑ Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers“, HPCA 2007.
- ❑ Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

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 eliminate all cache misses? Capacity, conflict, coherence?
- Involves predicting which address will be needed in the future
 - ❑ Works if programs have predictable miss address patterns

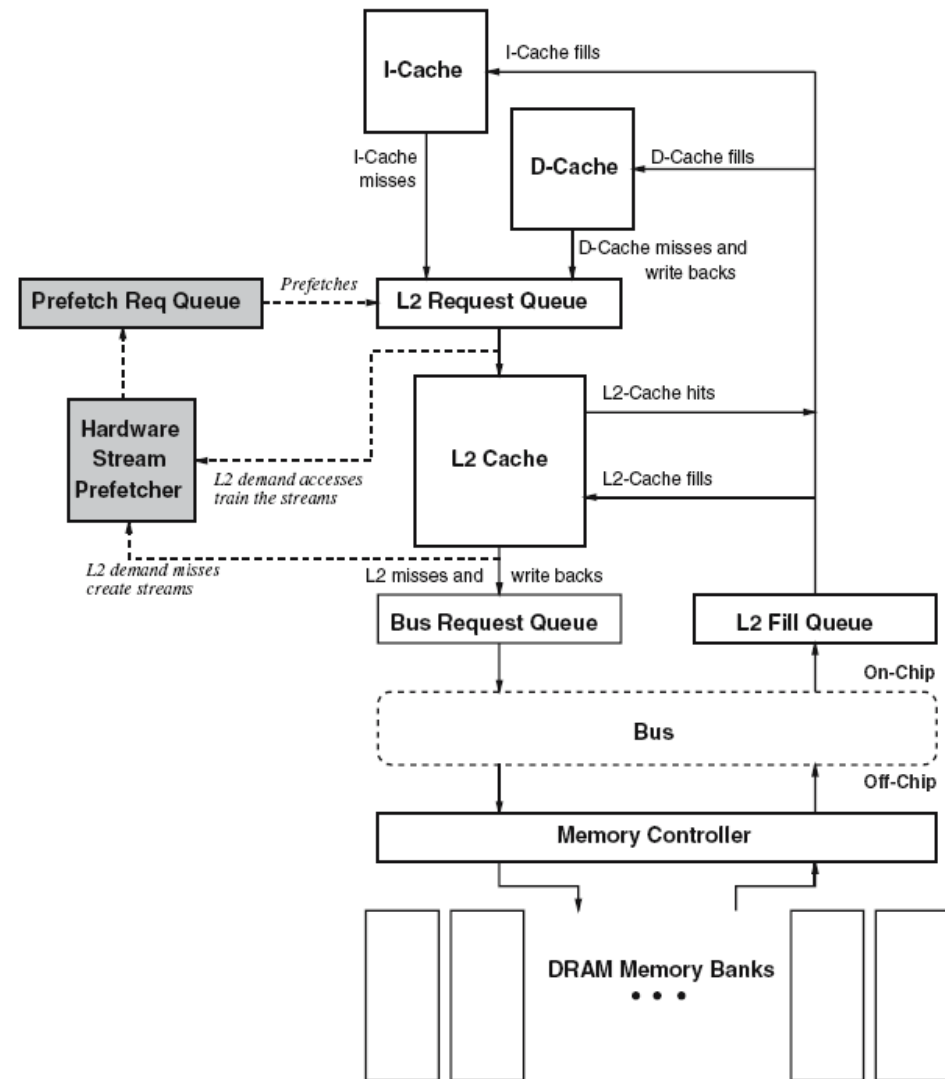
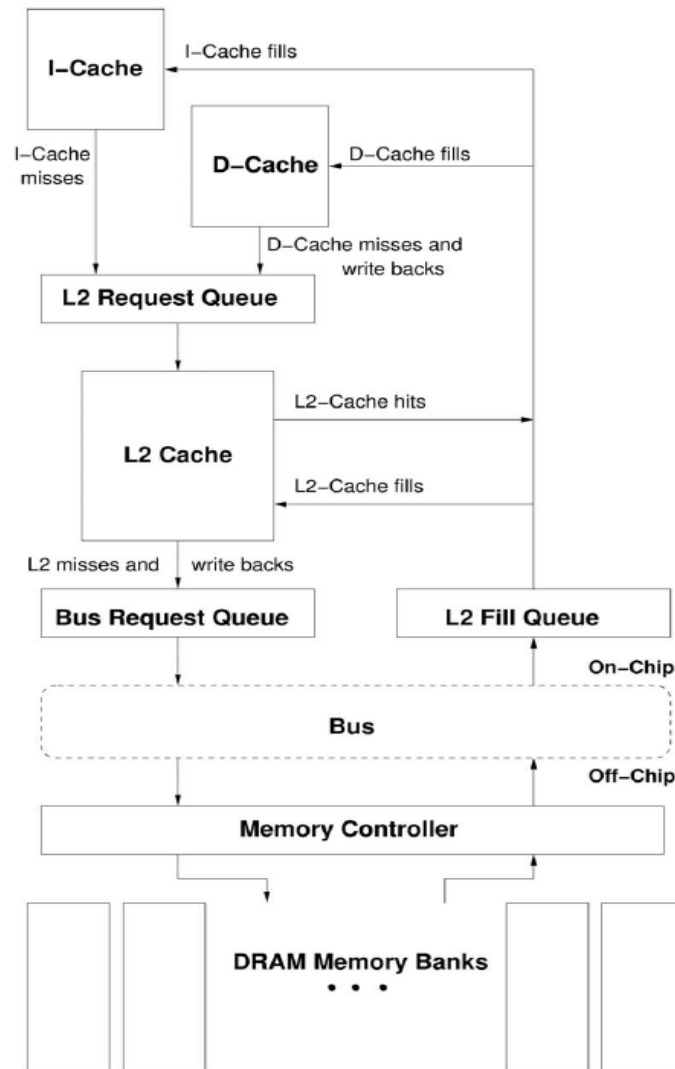
Prefetching and Correctness

- 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 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

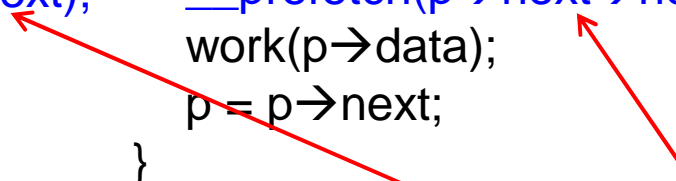
- **Software** prefetching
 - ❑ ISA provides prefetch instructions
 - ❑ Programmer or compiler inserts prefetch instructions (effort)
 - ❑ Usually works well only for “regular access patterns”
- **Hardware** prefetching
 - ❑ Hardware monitors processor accesses
 - ❑ Memorizes or finds patterns/strides
 - ❑ 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.
- Two types: binding vs. non-binding
 - Binding: Prefetch into a register (using a regular load)
 - + No need for a separate “prefetch” instruction
 - Takes up registers. Exceptions?
 - What if another processor modifies the data value before it is used?
 - Non-binding: Prefetch into cache (special instruction?)
 - + No coherence issues since caches are coherent
 - Prefetches treated differently from regular loads

Software Prefetching (II)

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



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

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 (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
 - + No code portability issues (in terms of performance variation between implementations)
 - + Does not waste instruction execution bandwidth
 - More hardware complexity to detect patterns
 - Software can be more efficient in some cases

Next-Line Prefetchers

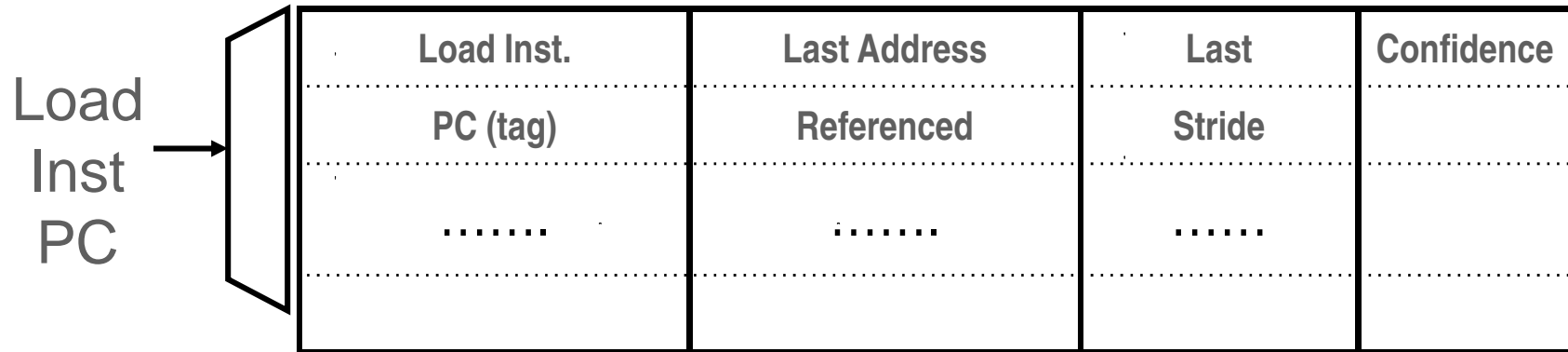
- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
 - Next-line prefetcher (or next sequential prefetcher)
 - Tradeoffs:
 - + Simple to implement. No need for sophisticated pattern detection
 - + Works well for sequential/streaming access patterns (instructions?)
 - Can waste bandwidth with irregular patterns
 - 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

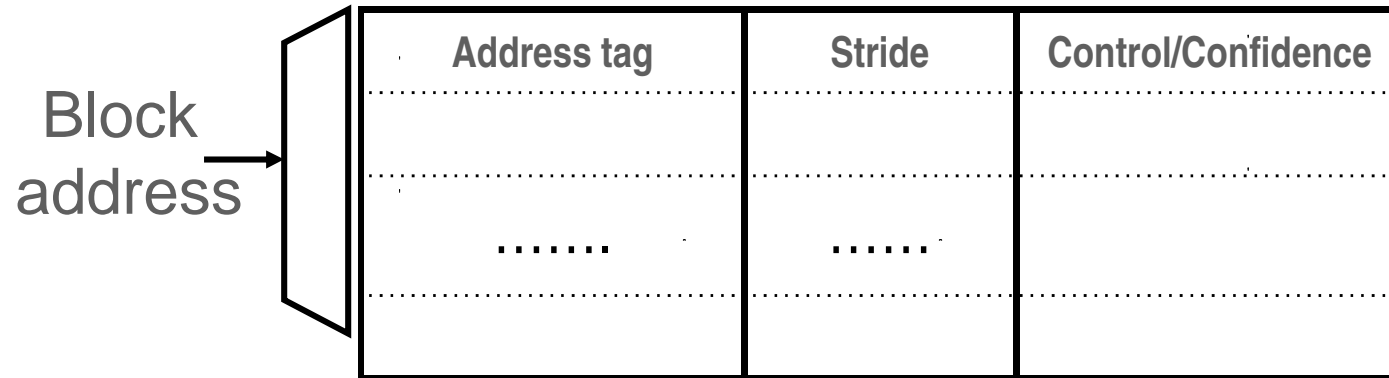
- Instruction based:
 - Baer and Chen, “An effective on-chip preloading scheme to reduce data access penalty,” SC 1991.
 - Idea:
 - Record the distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load
 - Next time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

Instruction Based Stride Prefetching



- What is the problem with this?
 - 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

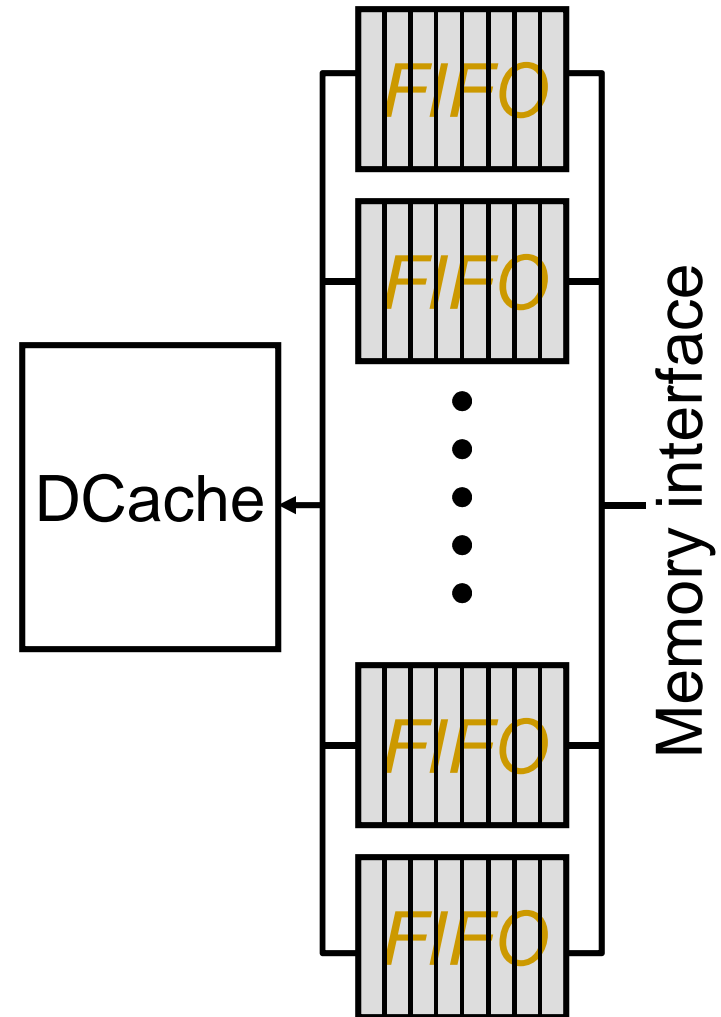
Cache-Block Address Based Stride Prefetching



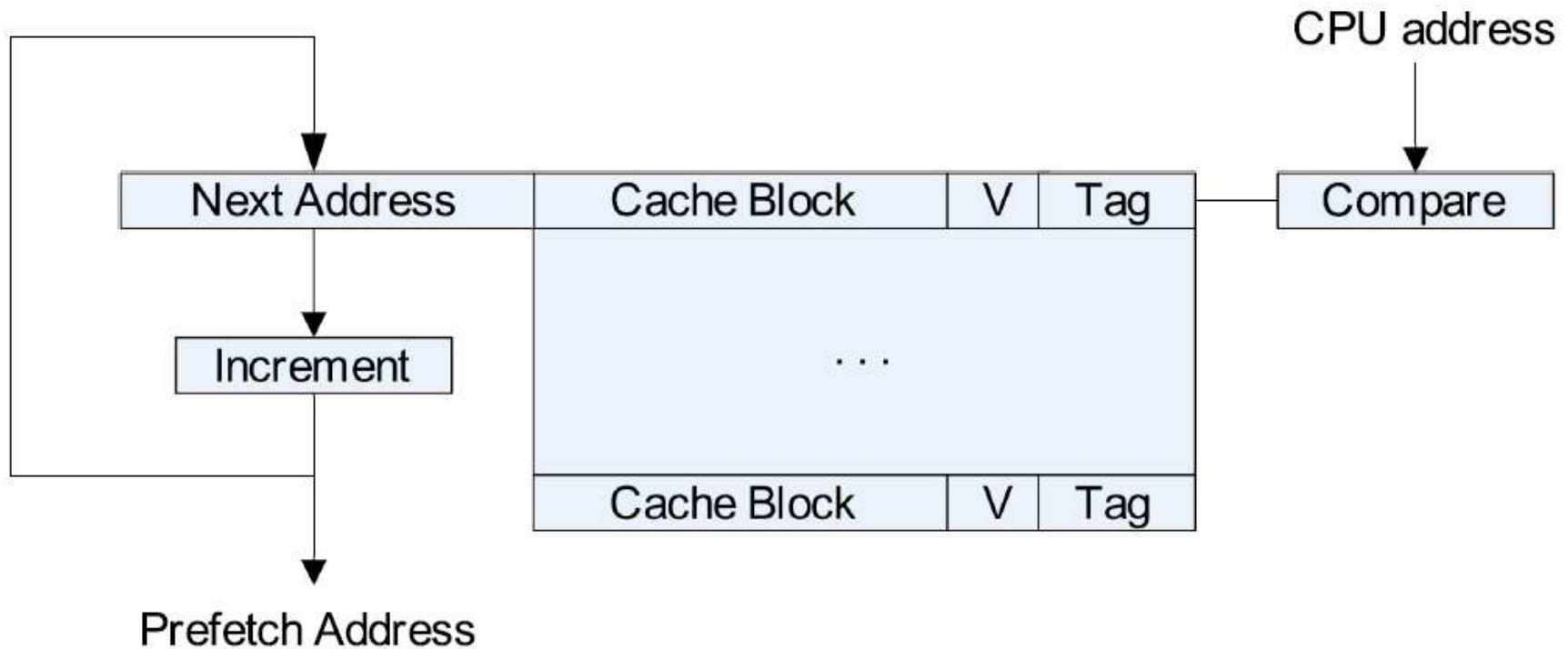
- Can detect
 - $A, A+N, A+2N, A+3N, \dots$
 - **Stream buffers** are a special case of cache block address based stride prefetching where $N = 1$
 - Read the Jouppi paper
 - Stream buffer also has data storage in that paper (no prefetching into cache)

Stream Buffers (Jouppi, ISCA 1990)

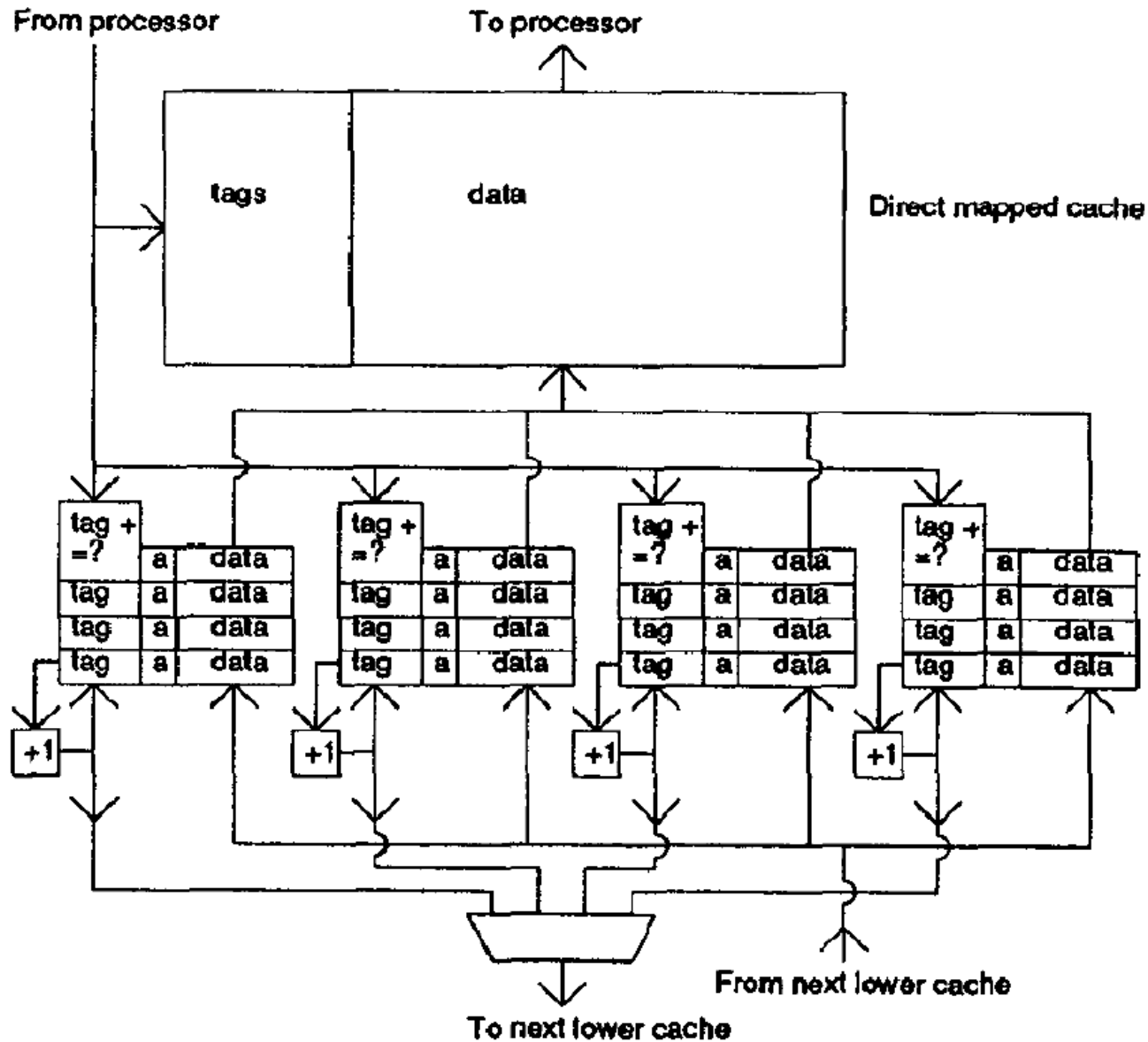
- Each stream buffer holds one stream of sequentially prefetched cache lines
- On a load miss check the head of all stream buffers for an address match
 - if hit, pop the entry from FIFO, update the cache with data
 - if not, allocate a new stream buffer to the new miss address (may have to recycle a 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
- Can incorporate stride prediction mechanisms to support non-unit-stride streams
 - See “Evaluating stream buffers as a secondary cache replacement”, ISCA 1994



Stream Buffer Design



Stream Buffer Design



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

Locality Based Prefetchers

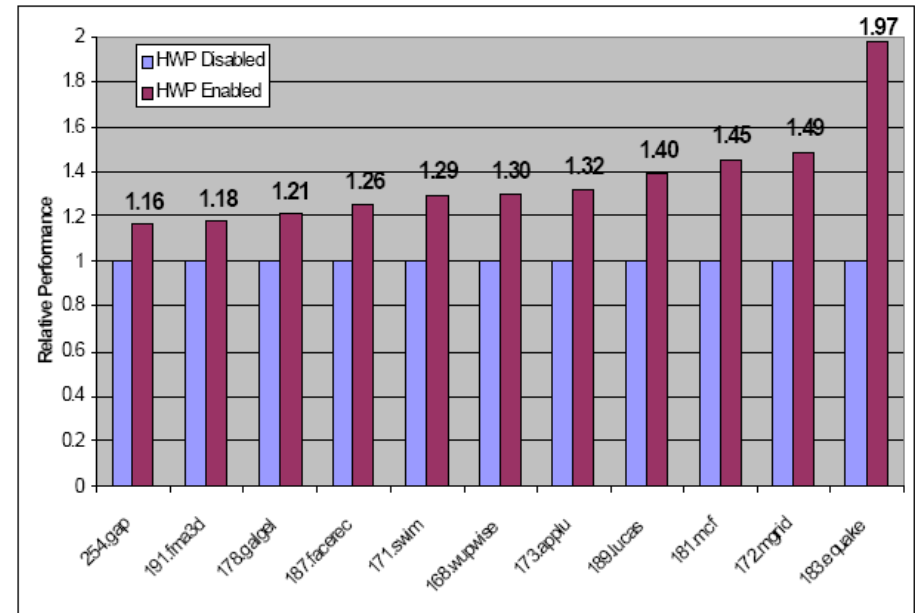
- In many applications access patterns are not perfectly strided
 - Some patterns look random to closeby addresses
 - How do you capture such accesses?
- Locality based prefetching
 - Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers“, HPCA 2007.

Pentium 4 (Like) Prefetcher (Srinath et al., HPCA 2007)

- Multiple tracking entries for a range of addresses
- **Invalid:** The tracking entry is not allocated a stream to keep track of. Initially, all tracking entries are in this state.
- **Allocated:** A demand (i.e. load/store) L2 miss allocates a tracking entry if the demand miss does not find any existing tracking entry for its cache-block address.
- **Training:** The prefetcher trains the direction (ascending or descending) of the stream based on the next two L2 misses that occur +/- 16 cache blocks from the first miss. If the next two accesses in the stream are to ascending (descending) addresses, the direction of the tracking entry is set to 1 (0) and the entry transitions to *Monitor and Request state*.
- **Monitor and Request:** The tracking entry monitors the accesses to a memory region from a *start pointer (address A)* to an *end pointer (address P)*. The *maximum distance* between the start pointer and the end pointer is determined by *Prefetch Distance*, which indicates how far ahead of the demand access stream the prefetcher can send requests. If there is a demand L2 cache access to a cache block in the monitored memory region, the prefetcher requests cache blocks [P+1, ..., P+N] as prefetch requests (assuming the direction of the tracking entry is set to 1). N is called the *Prefetch Degree*. After sending the prefetch requests, the tracking entry starts monitoring the memory region between addresses A+N to P+N (i.e. effectively it moves the tracked memory region by N cache blocks).

Limitations of Locality-Based Prefetchers

- Bandwidth intensive
 - Why?
 - Can be fixed by
 - Stride detection
 - Feedback mechanisms



- Limited to prefetching closeby addresses
 - What about large jumps in addresses accessed?
- However, they work very well in real life
 - Single-core systems
 - Boggs et al., Intel Technology Journal, Feb 2004.

Prefetcher Performance (I)

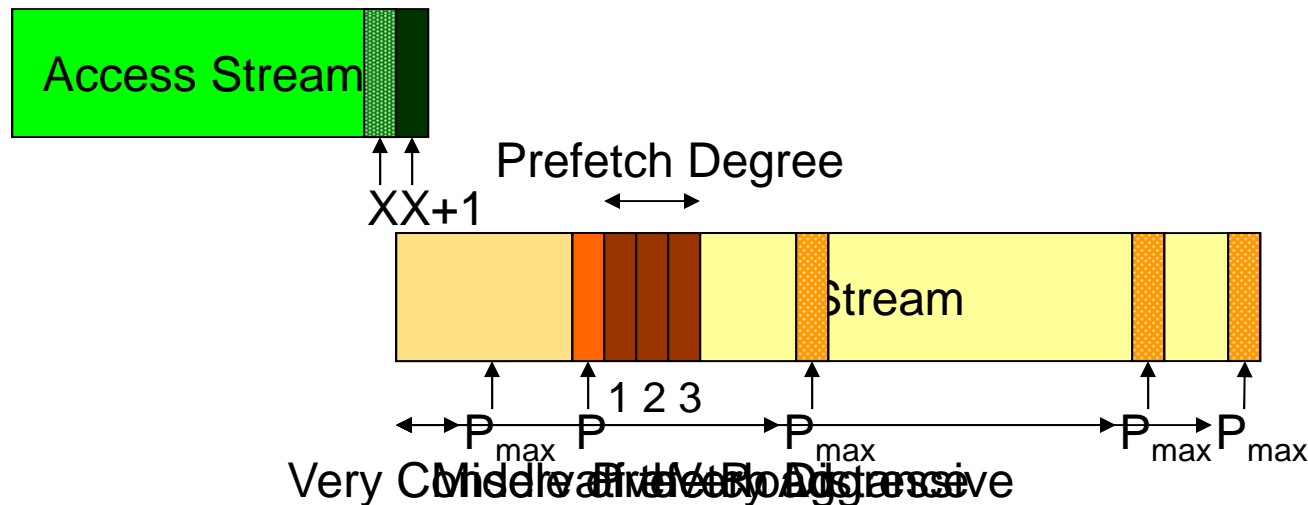
- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetches / used prefetches)

- **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

Prefetcher Performance (II)

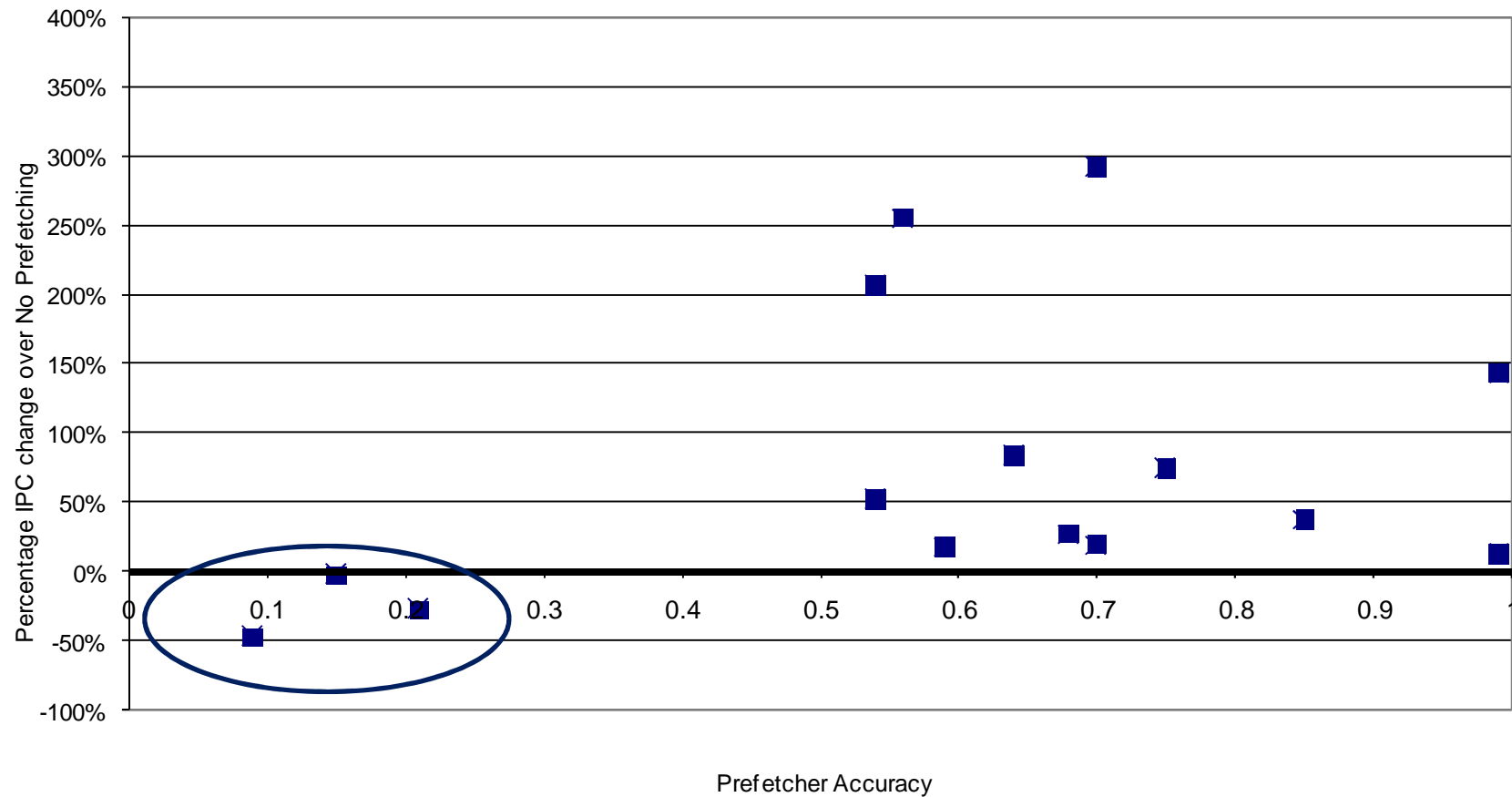
- Prefetcher aggressiveness affects all performance metrics
- Aggressiveness dependent on prefetcher type
- For most hardware prefetchers:
 - **Prefetch distance**: how far ahead of the demand stream
 - **Prefetch degree**: how many prefetches per demand access



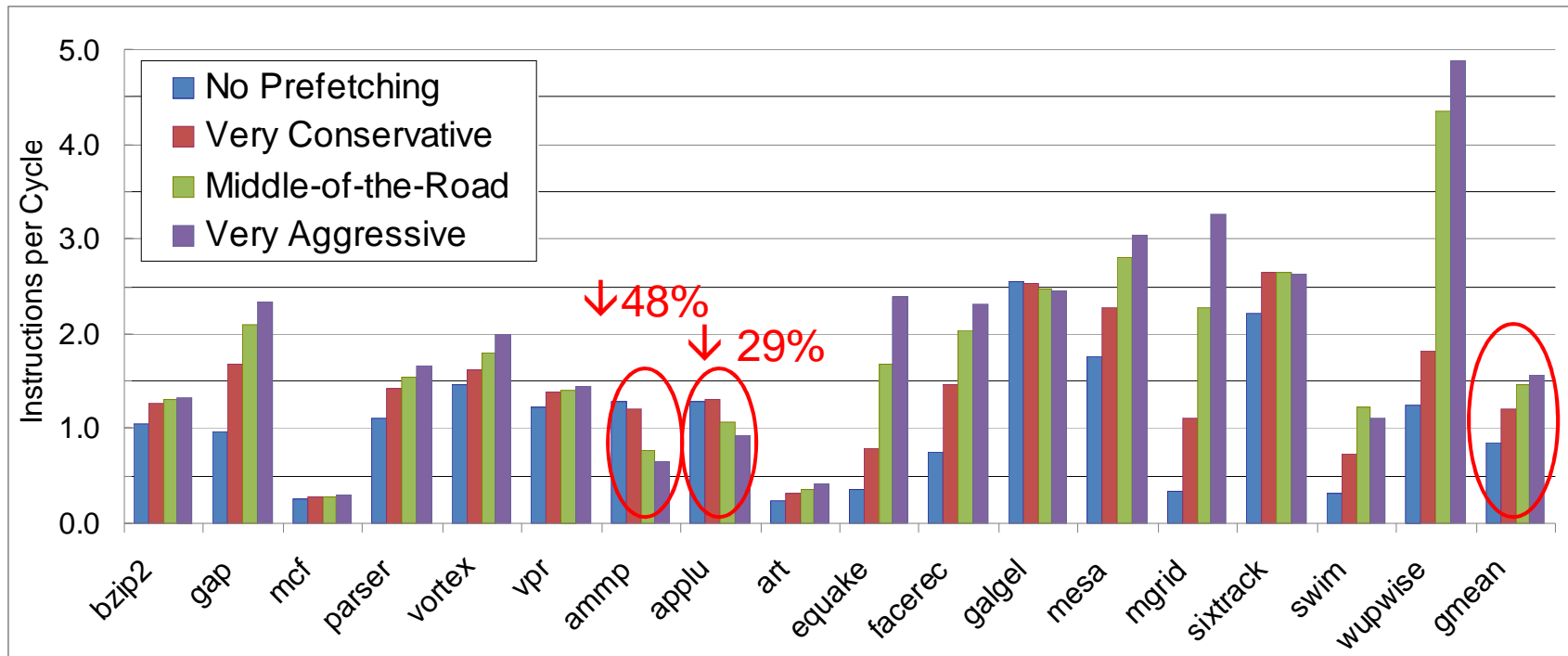
Prefetcher Performance (III)

- 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)

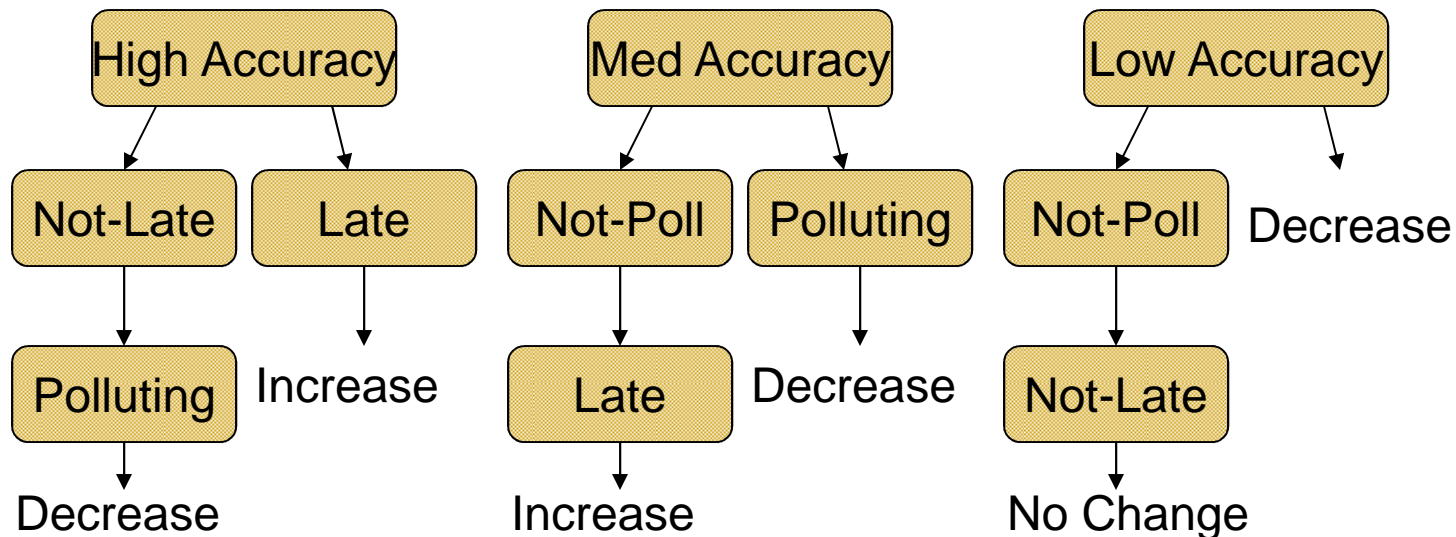


- Srinath et al., “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA 2007.

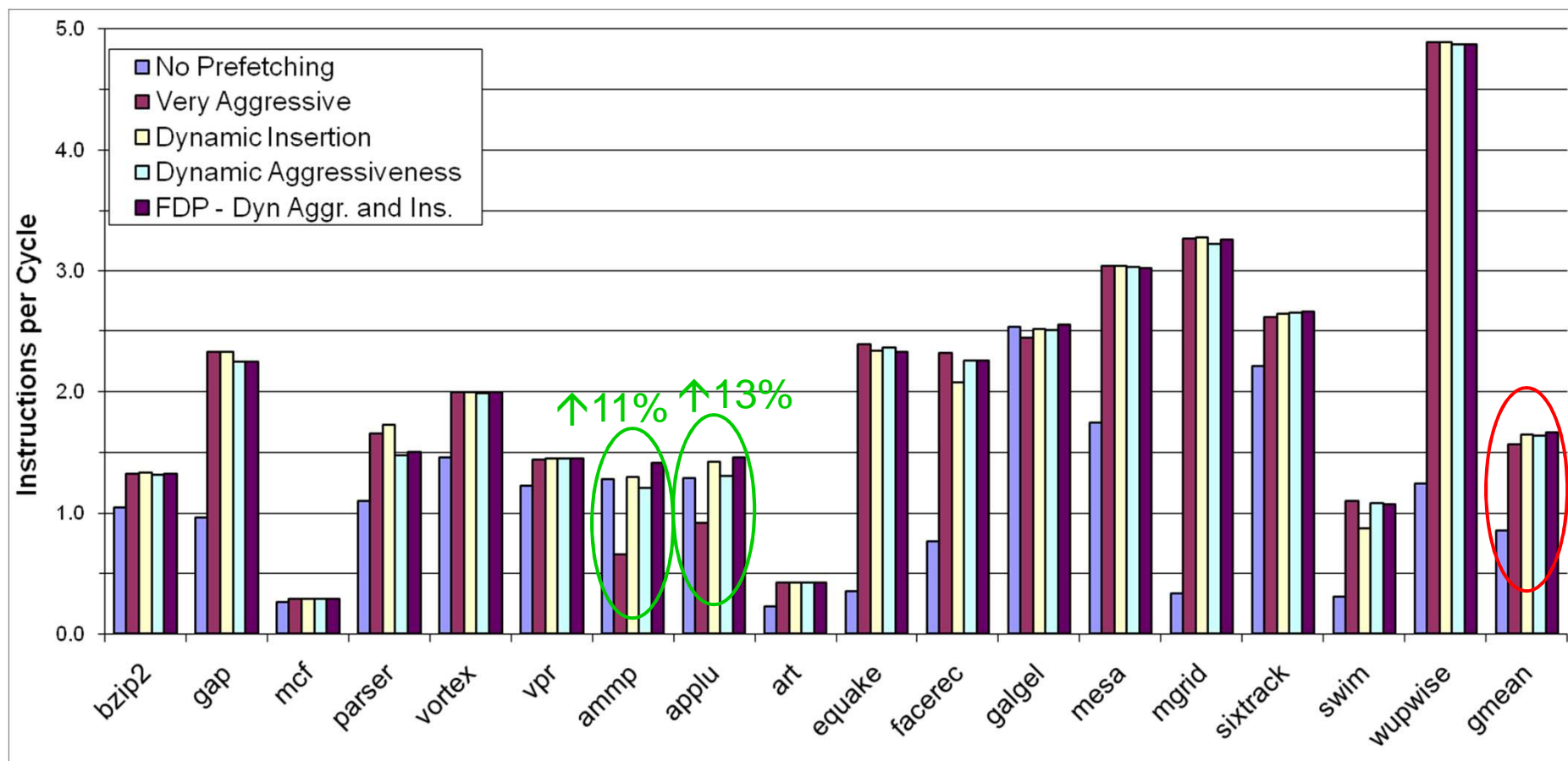
Feedback-Directed Prefetcher Throttling (I)

■ Idea:

- 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.

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

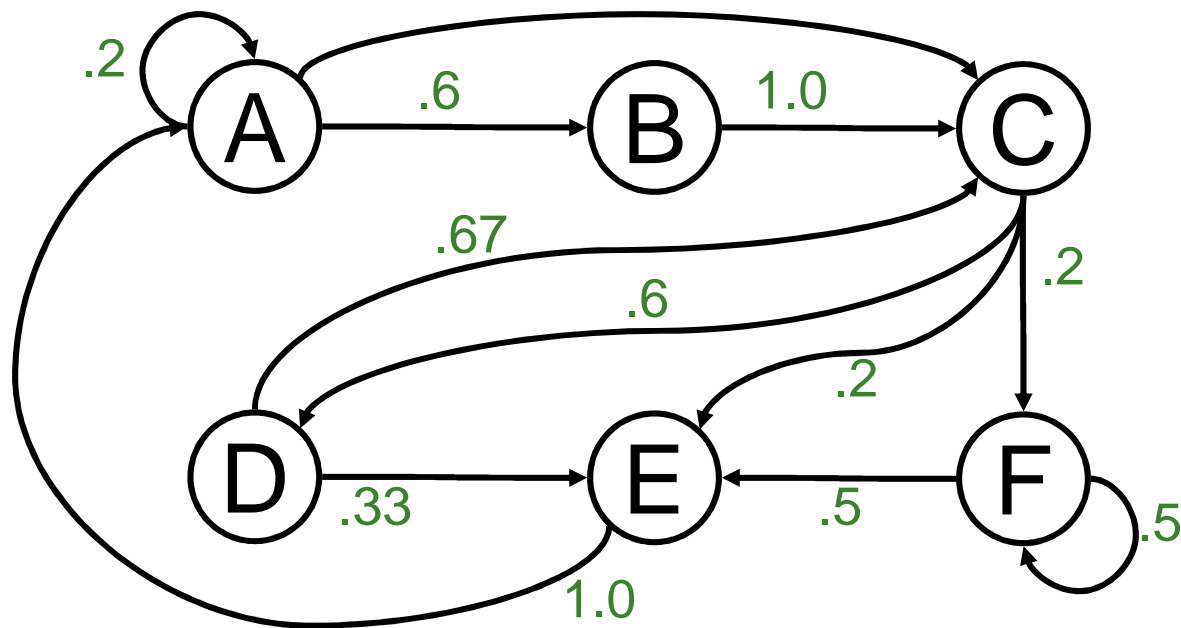
	No. Pref.	Very Cons	Mid	Very Aggr	FDP
IPC	0.85	1.21	1.47	1.57	1.67
BPKI	8.56	9.34	10.60	13.38	10.88

How to Cover More Irregular Access Patterns?

- 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

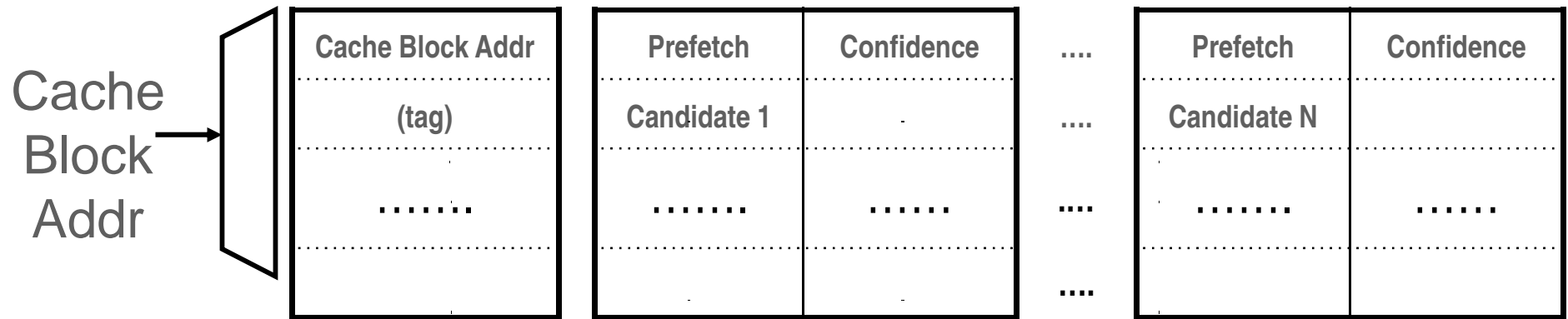
Markov Prefetching (I)

- 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)



- Track the likely next addresses after seeing a particular address
- Prefetch *accuracy* is generally low so prefetch up to N next addresses to increase *coverage*
- Prefetch accuracy can be improved by using longer history
 - Decide which address to prefetch next by looking at the last K load addresses instead of just the current one
 - e.g., index with the XOR of the data addresses from the last K loads
 - Using history of a few loads can increase accuracy dramatically
- 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!)

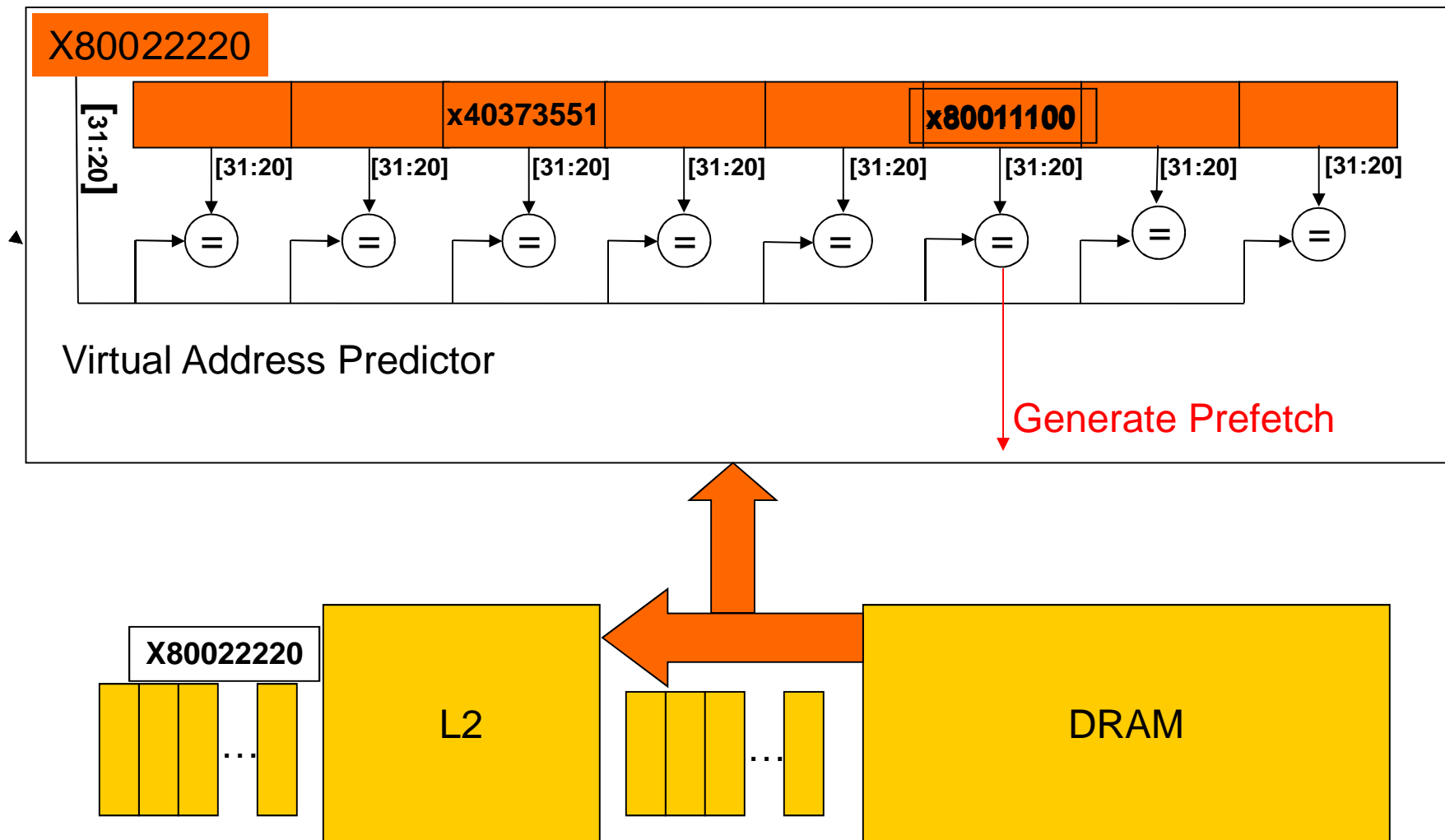
■ Disadvantages:

- ❑ **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) {
```

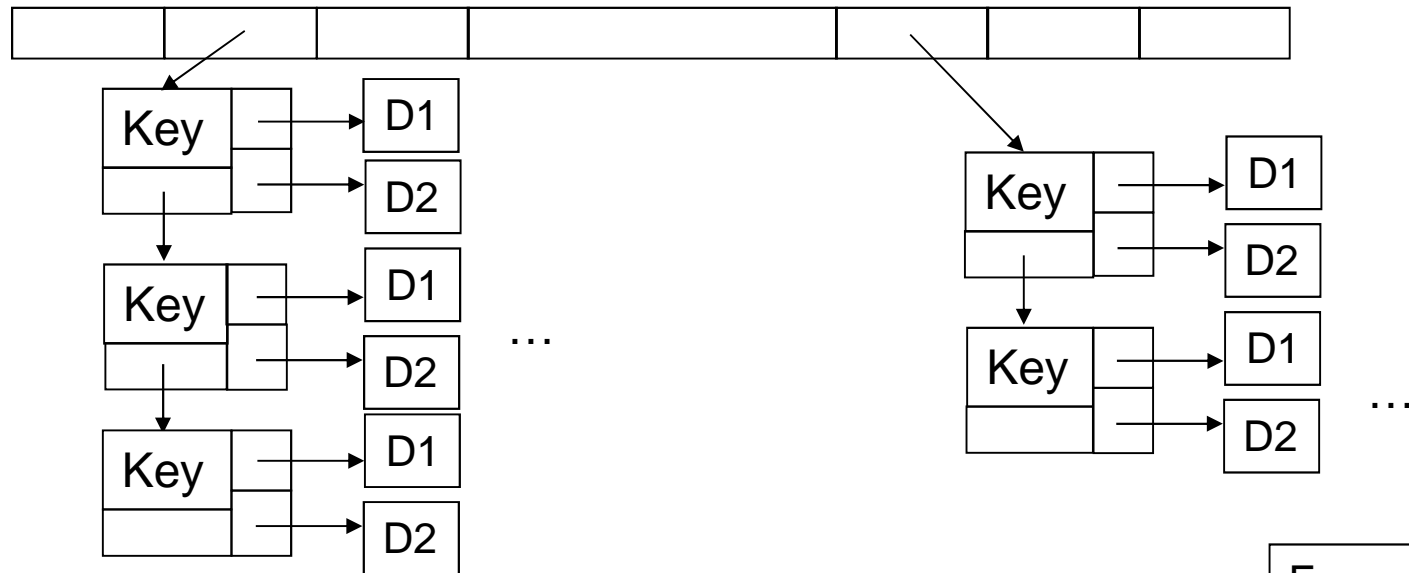
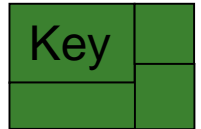
```
...
```

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

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

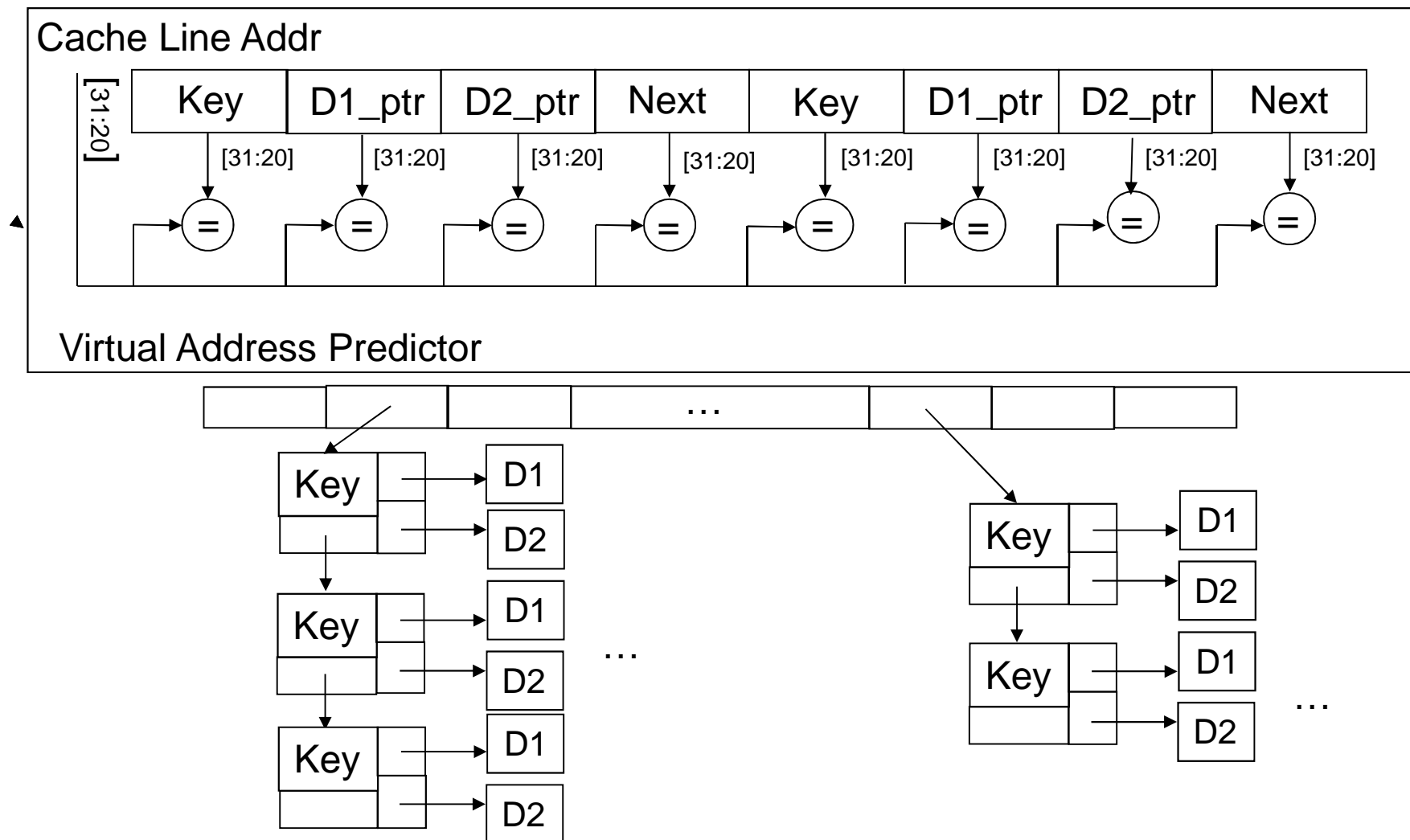
```
}
```

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



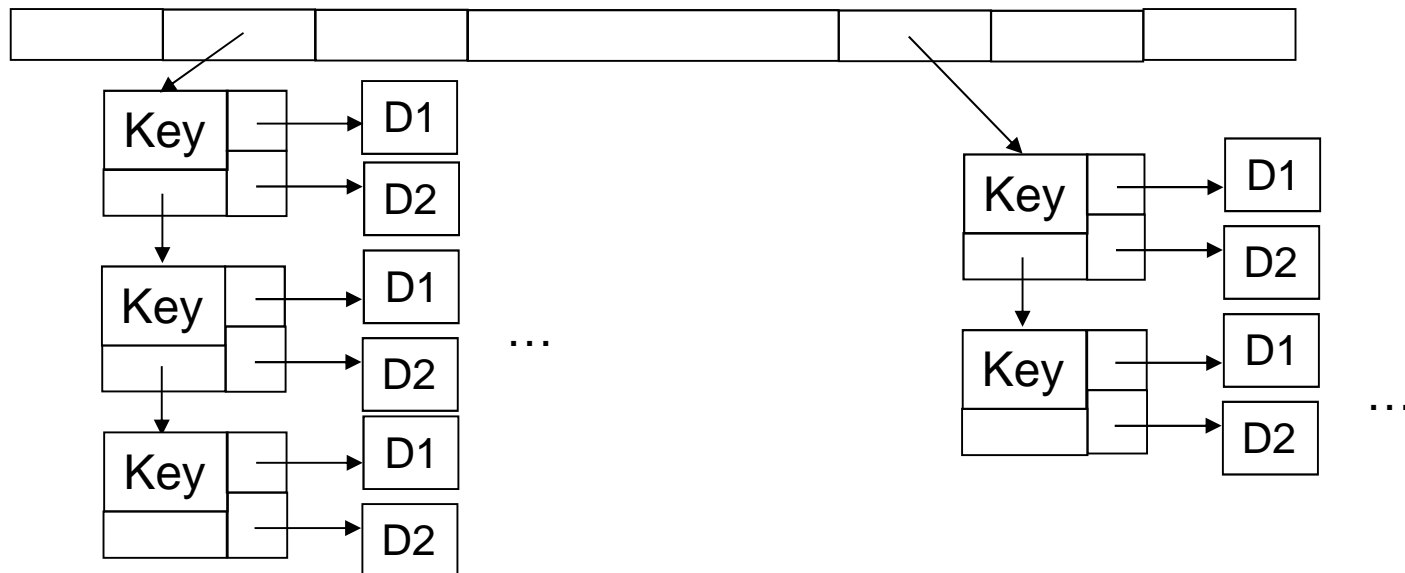
Example from mst

Shortcomings of CDP – An example

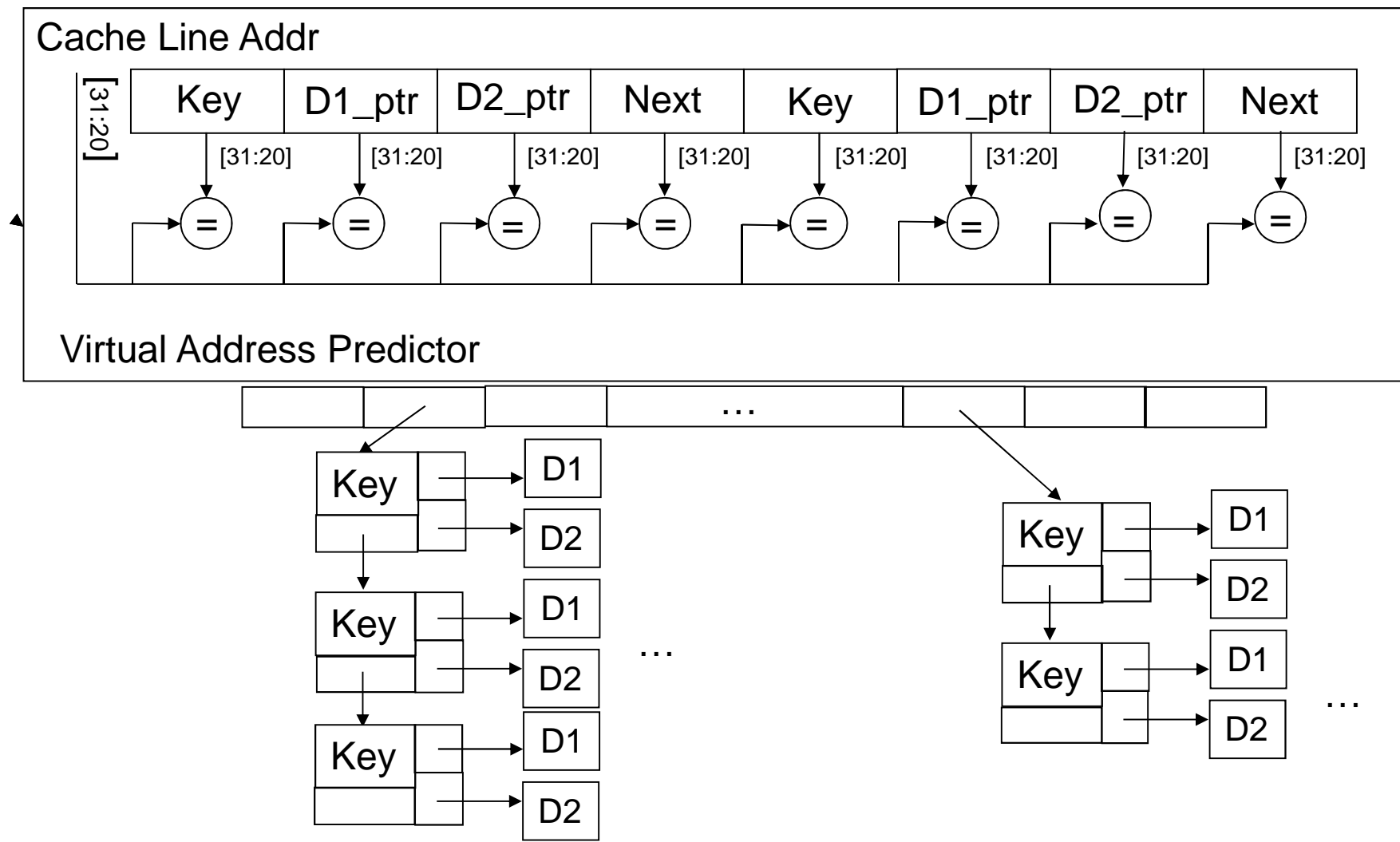


Shortcomings of CDP – An example

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



Shortcomings of CDP – An example



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)

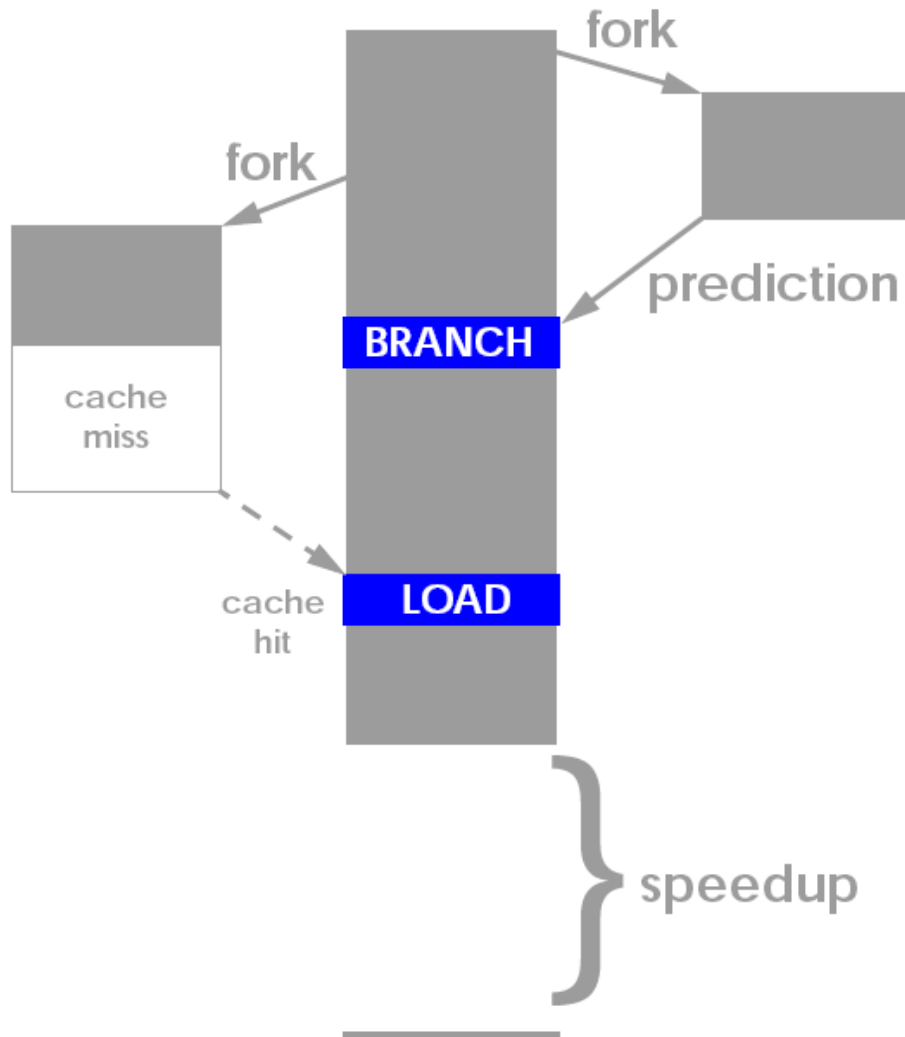
- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
 - Only need to distill pieces that lead to cache misses
- **Speculative thread:** Pre-executed program piece can be considered a “thread”
- Speculative thread can be executed
 - On a separate processor/core
 - On a separate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

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.

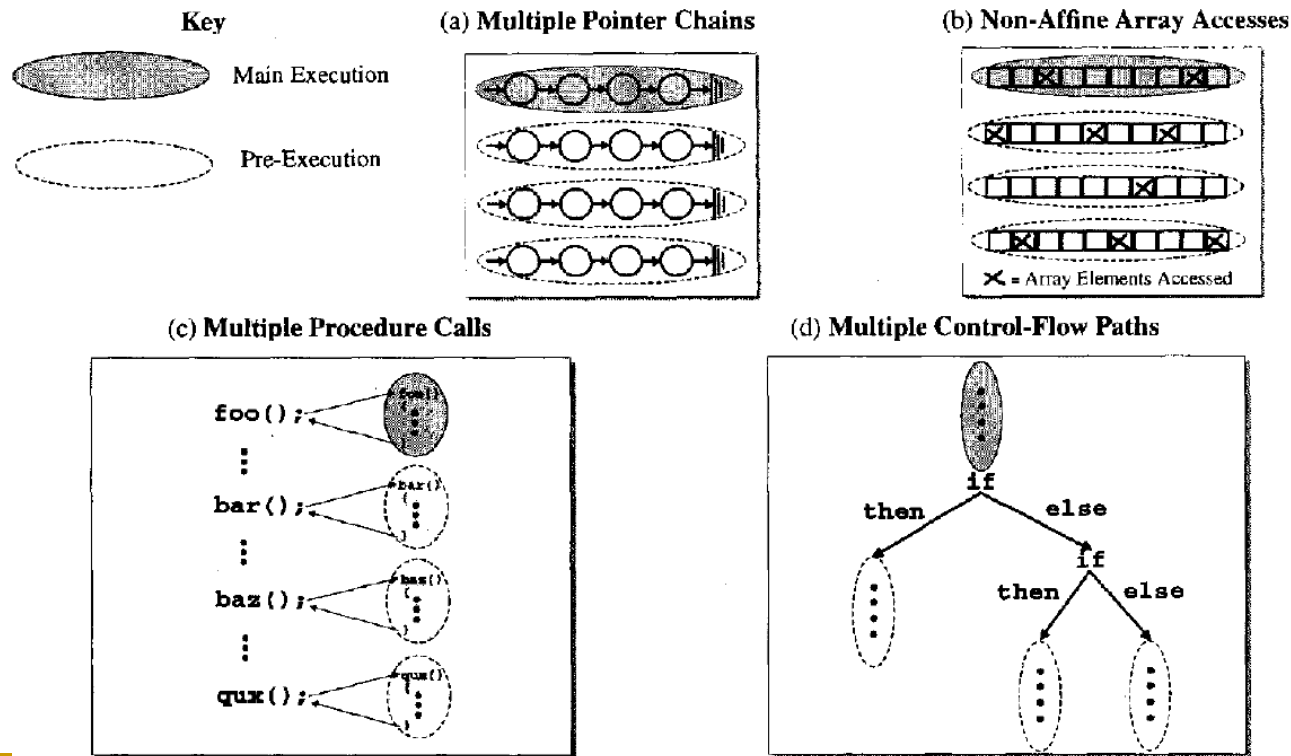
Thread-Based Pre-Execution Issues

- 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();
```

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

Example ISA Extensions

Thread_ID = PreExecute_Start(Start_PC, Max_Insts):

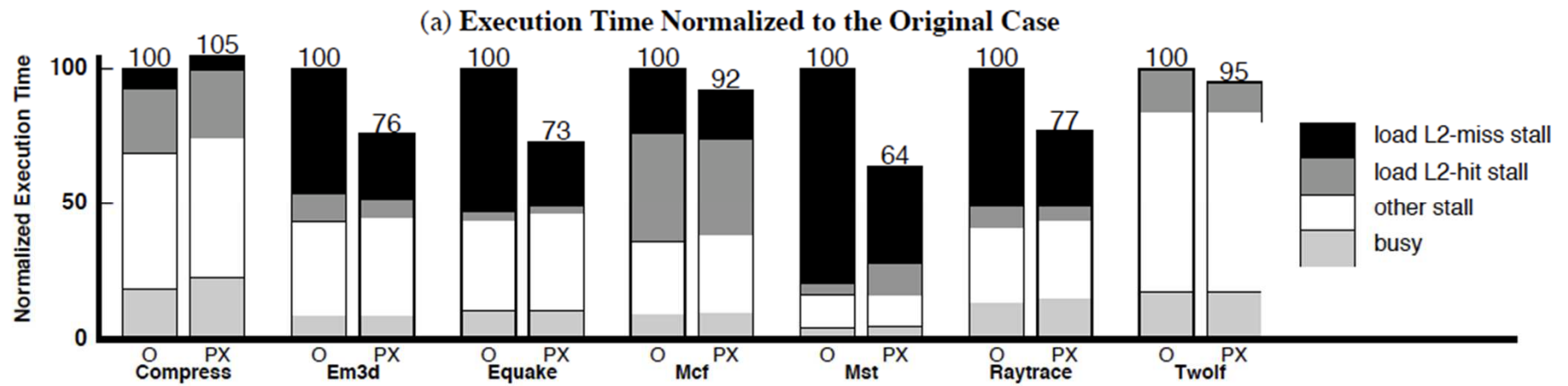
Request for an idle context to start pre-execution at *Start_PC* and stop when *Max_Insts* instructions have been executed; *Thread_ID* holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

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

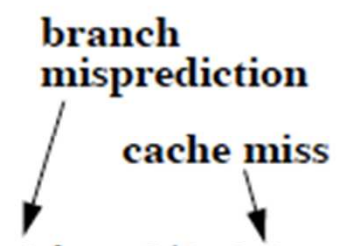
Figure 2. *Example problem instructions from heap insertion routine in **vpr**.*

```
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

cache miss



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

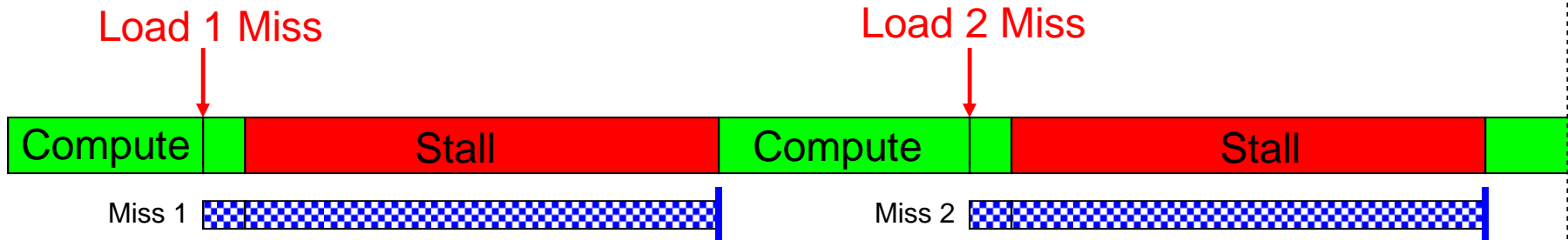
## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```


Runahead Execution (I)

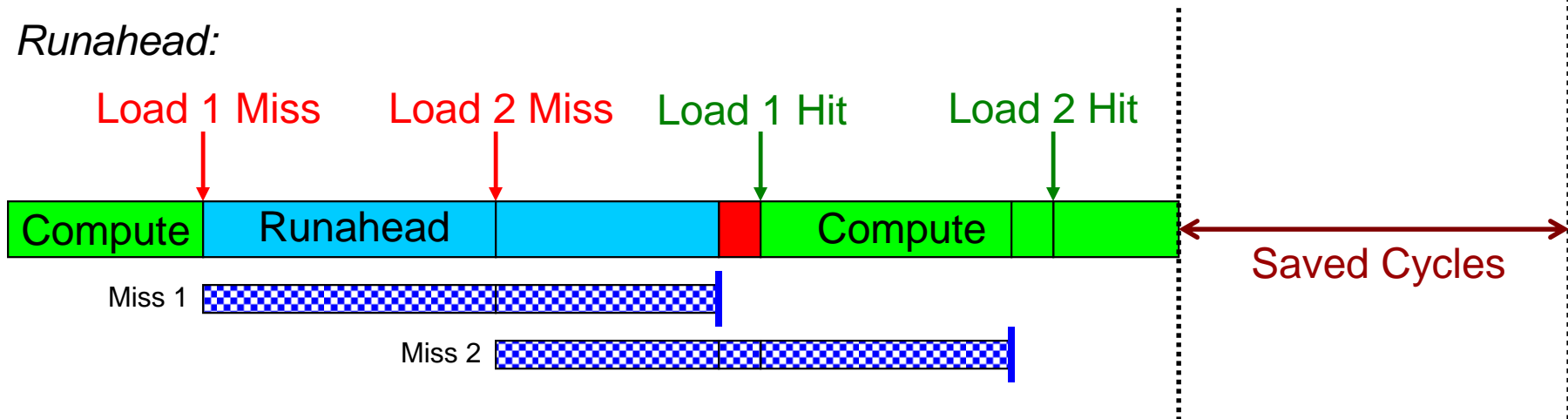
- A simple pre-execution method for prefetching purposes
- When the oldest instruction is a long-latency cache miss:
 - Checkpoint architectural state and enter runahead mode
- In runahead mode:
 - Speculatively pre-execute instructions
 - The purpose of pre-execution is to generate prefetches
 - L2-miss 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:



Runahead Execution (III)

■ Advantages:

- + Very **accurate** prefetches for data/instructions (all cache levels)
 - + Follows the program path
- + No need to construct a pre-execution thread
- + Uses the same thread context as main thread, no waste of context
- + **Simple to implement**, most of the hardware is already built in

■ Disadvantages/Limitations:

- **Extra executed instructions**
- Limited by branch prediction accuracy
- Cannot prefetch dependent cache misses. Solution?
- **Effectiveness limited by available MLP**
- **Prefetch distance limited by memory latency**

■ Implemented in IBM POWER6, Sun “Rock”

Execution-based Prefetchers (III)

- + Can prefetch pretty much **any access pattern**
- + **Can be very low cost** (e.g., runahead execution)
 - + Especially if it uses the same hardware context
 - + Why? The processor is equipped to execute the program anyway
- + **Can be bandwidth-efficient** (e.g., runahead execution)

- Depend on **branch prediction and possibly value prediction accuracy**
 - Mispredicted branches dependent on missing data throw the thread off the correct execution path
- Can be **wasteful**
 - speculatively execute many instructions
 - can occupy a separate thread context

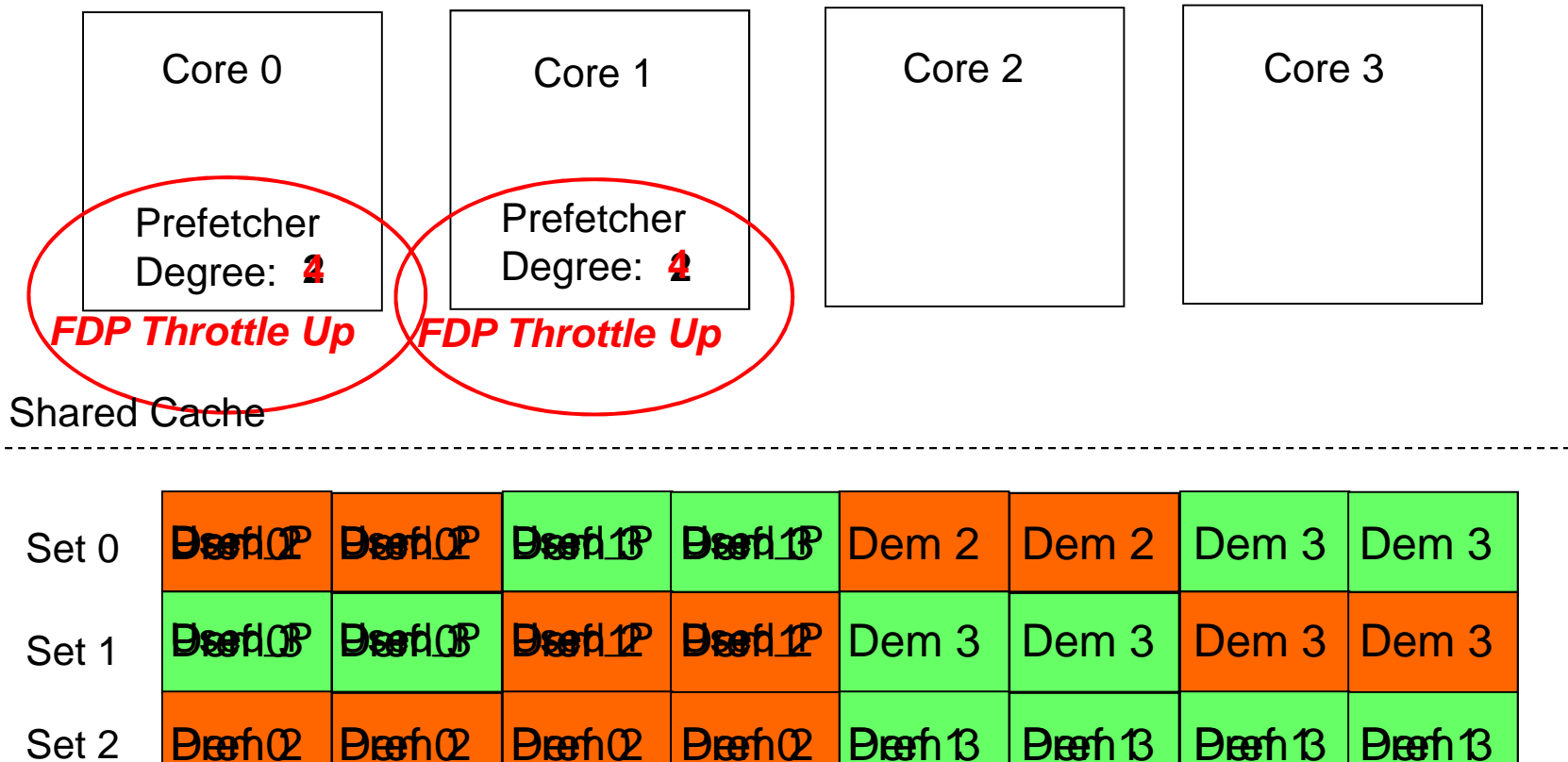
Multi-Core Issues in Prefetching

Prefetching in Multi-Core (I)

- Prefetching shared data
 - Coherence misses

- Prefetch efficiency a lot more important
 - Bus bandwidth more precious
 - Prefetchers on different cores can deny service to each other and to demand requests
 - DRAM contention
 - Bus contention
 - Cache conflicts
 - Need to **coordinate the actions of independent prefetchers** for best system performance
 - Each prefetcher has different accuracy, coverage, timeliness

Shortcoming of Local Prefetcher Throttling



Local-only prefetcher control techniques
have no mechanism to detect inter-core interference

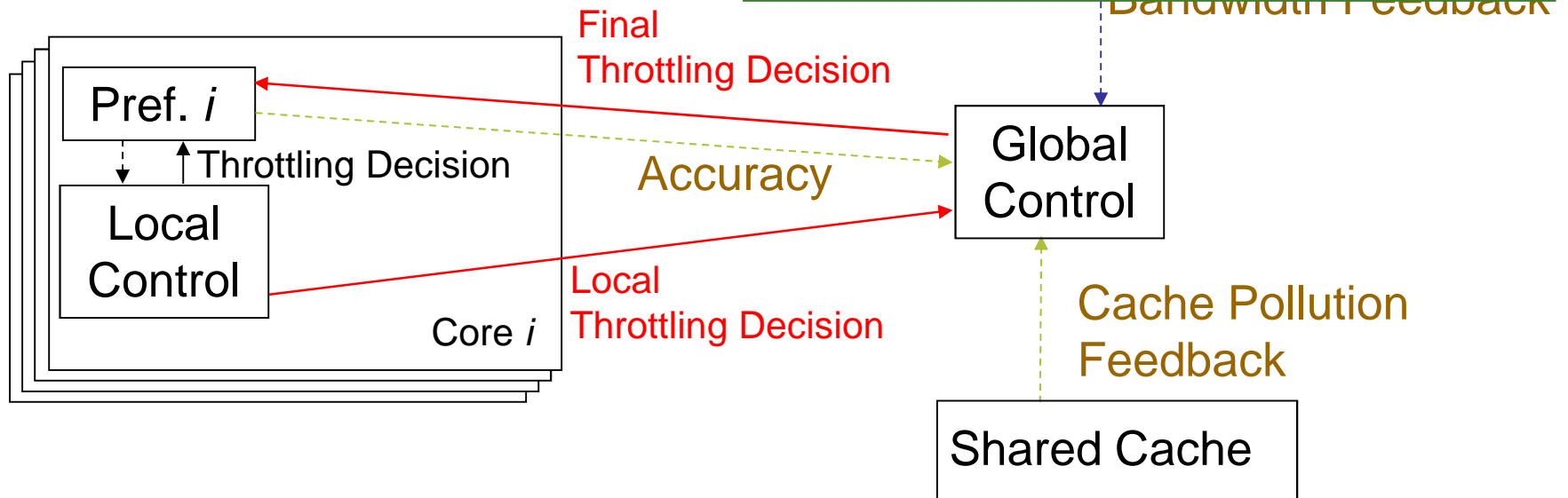
Prefetching in Multi-Core (II)

- Ideas for coordinating different prefetchers' actions
 - Utility-based prioritization
 - Prioritize prefetchers that provide the best marginal utility on system performance
 - Cost-benefit analysis
 - Compute cost-benefit of each prefetcher to drive prioritization
 - Heuristic based methods
 - Global controller overrides local controller's throttling decision based on interference and accuracy of prefetchers
 - Ebrahimi et al., “Coordinated Management of Multiple Prefetchers in Multi-Core Systems,” MICRO 2009.

Hierarchical Prefetcher Throttling

Global Control's goal: **Maximize the prefetching performance of core i independently** or **overrides the decisions made by local control to improve overall system performance**

Global control's goal: Keep track of and control **prefetcher-caused inter-core interference** in shared memory system



Hierarchical Prefetcher Throttling Example

- High accuracy
- High pollution
- High bandwidth consumed while other cores need bandwidth

