

15-740/18-740  
Computer Architecture  
Lecture 16: Prefetching Wrap-up

Prof. Onur Mutlu  
Carnegie Mellon University

# Announcements

---

- Exam solutions online
- Pick up your exams
  
- Feedback forms

# Feedback Survey Results (I)

---

- How fast is the pace of the course so far?
  - Good: 29
  - Fast: 13
  - Slow: 2
- How fast is the pace of lectures?
  - Good: 33
  - Fast: 6
  - Slow: 5
- How easy is the course material?
  - Right level: 33
  - Hard: 11
  - Easy: 0

# Feedback Survey Results (II)

---

- How heavy is the course workload?
  - Right amount: 13
  - High: 29
  - Low: 1

# Last Time

---

- Hardware Prefetching
  - Next-line
  - Stride
    - Instruction based
    - Cache block address based
  - Stream buffers
  - Locality based prefetchers
  - Prefetcher performance: Accuracy, coverage, timeliness
  - Prefetcher aggressiveness
  - Feedback directed prefetcher throttling

# How to Cover More Irregular Access Patterns?

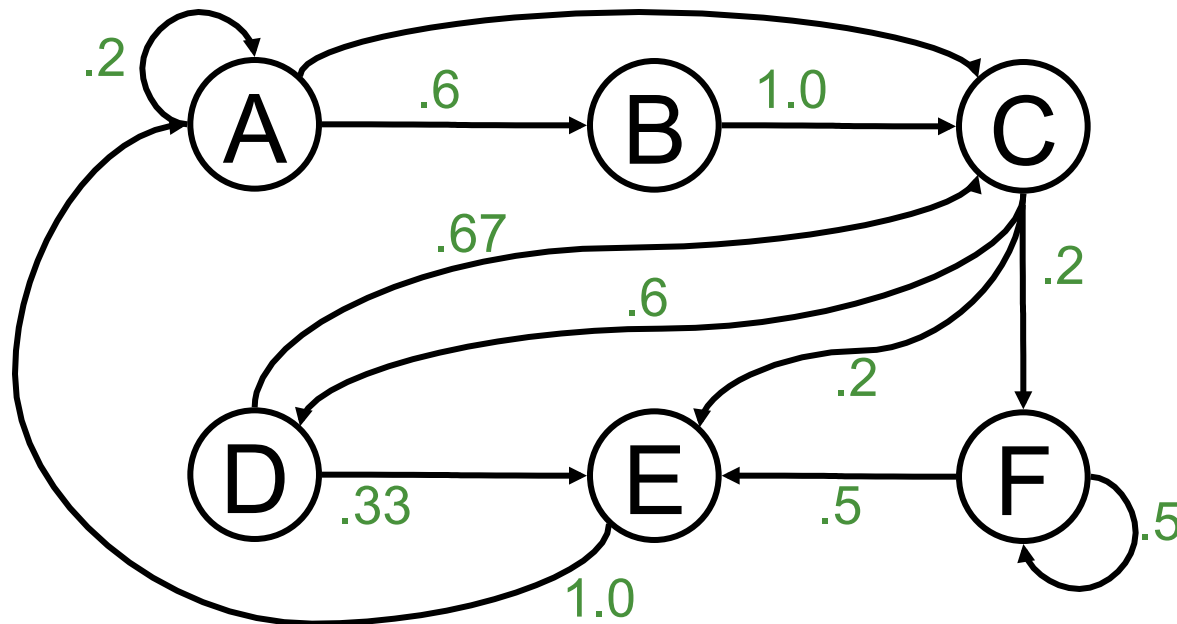
---

- 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)

---

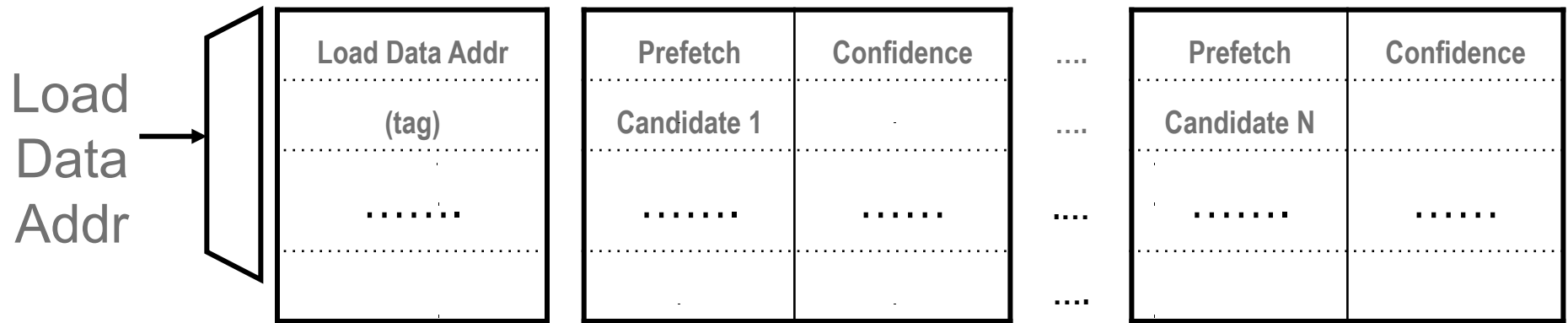
- Consider the following history of load addresses  
A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C
- After referencing a particular address (say A or E), are some addresses more likely to be referenced next



*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)

---

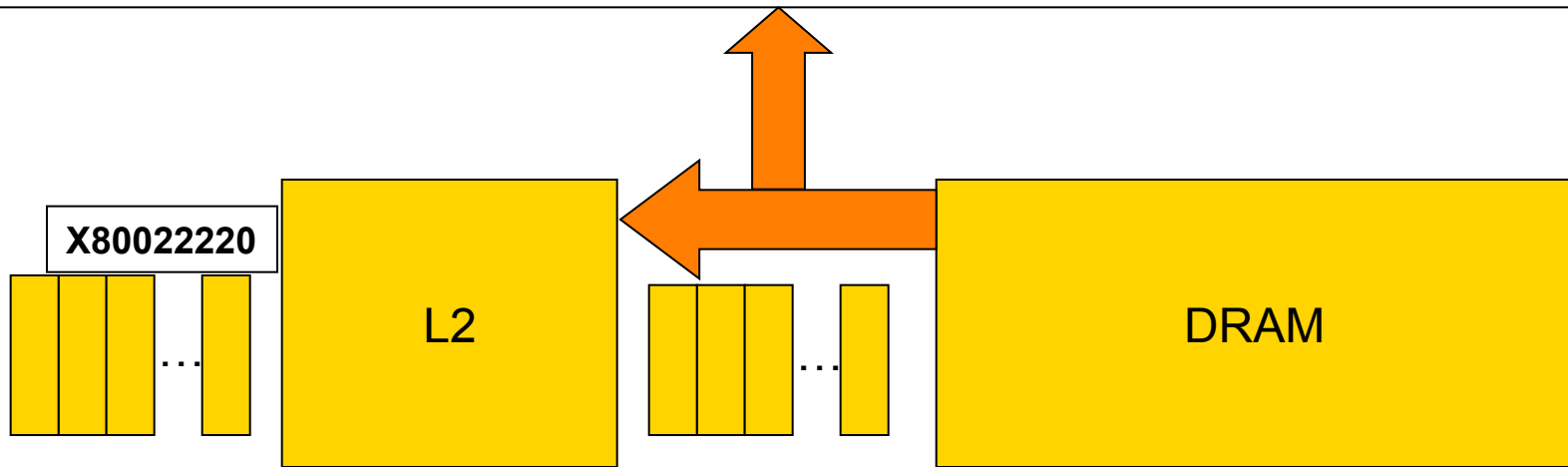
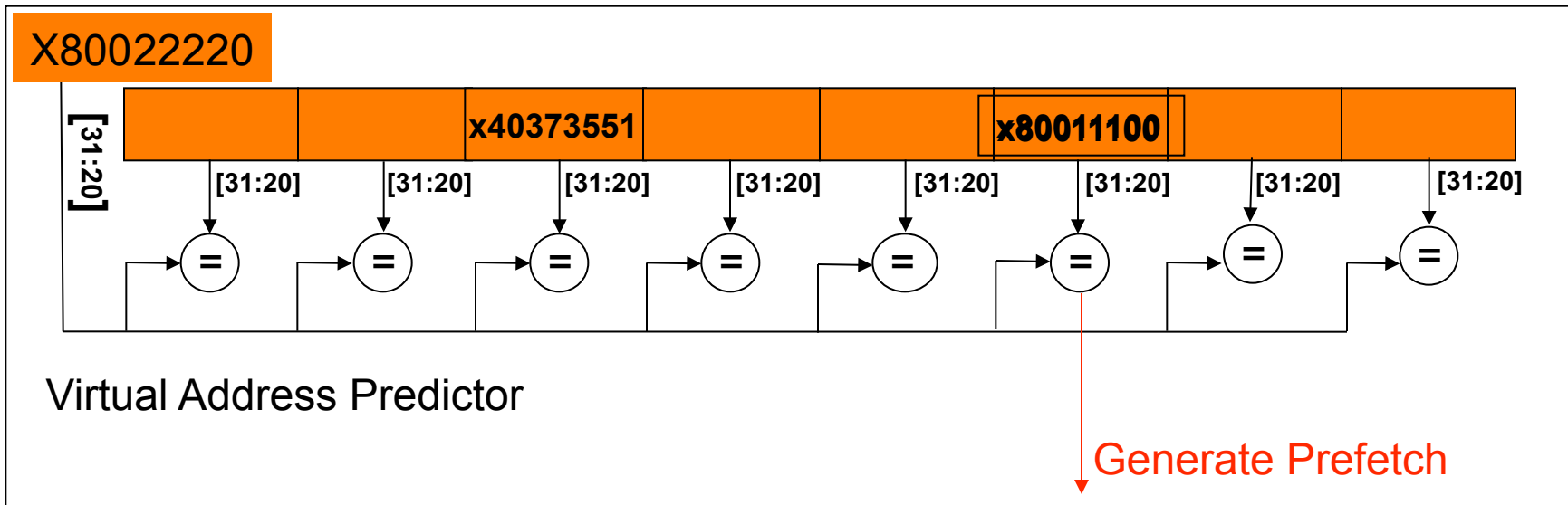
- Advantages:
  - Can cover **arbitrary access patterns**
    - Linked data structures
    - Streaming patterns (though not so efficiently!)
- Disadvantages:
  - **Correlation table** needs to be very large for high coverage
    - Recording every miss address and its subsequent miss addresses is infeasible
  - **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

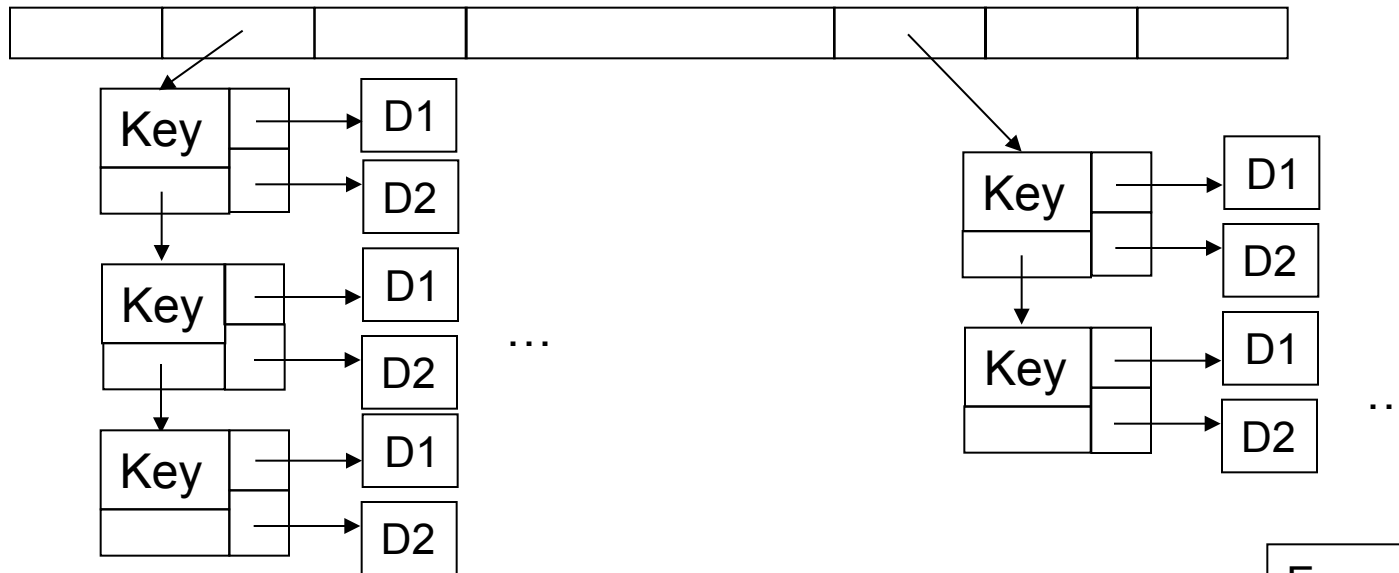
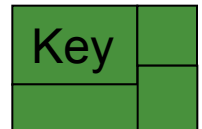
---

- 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

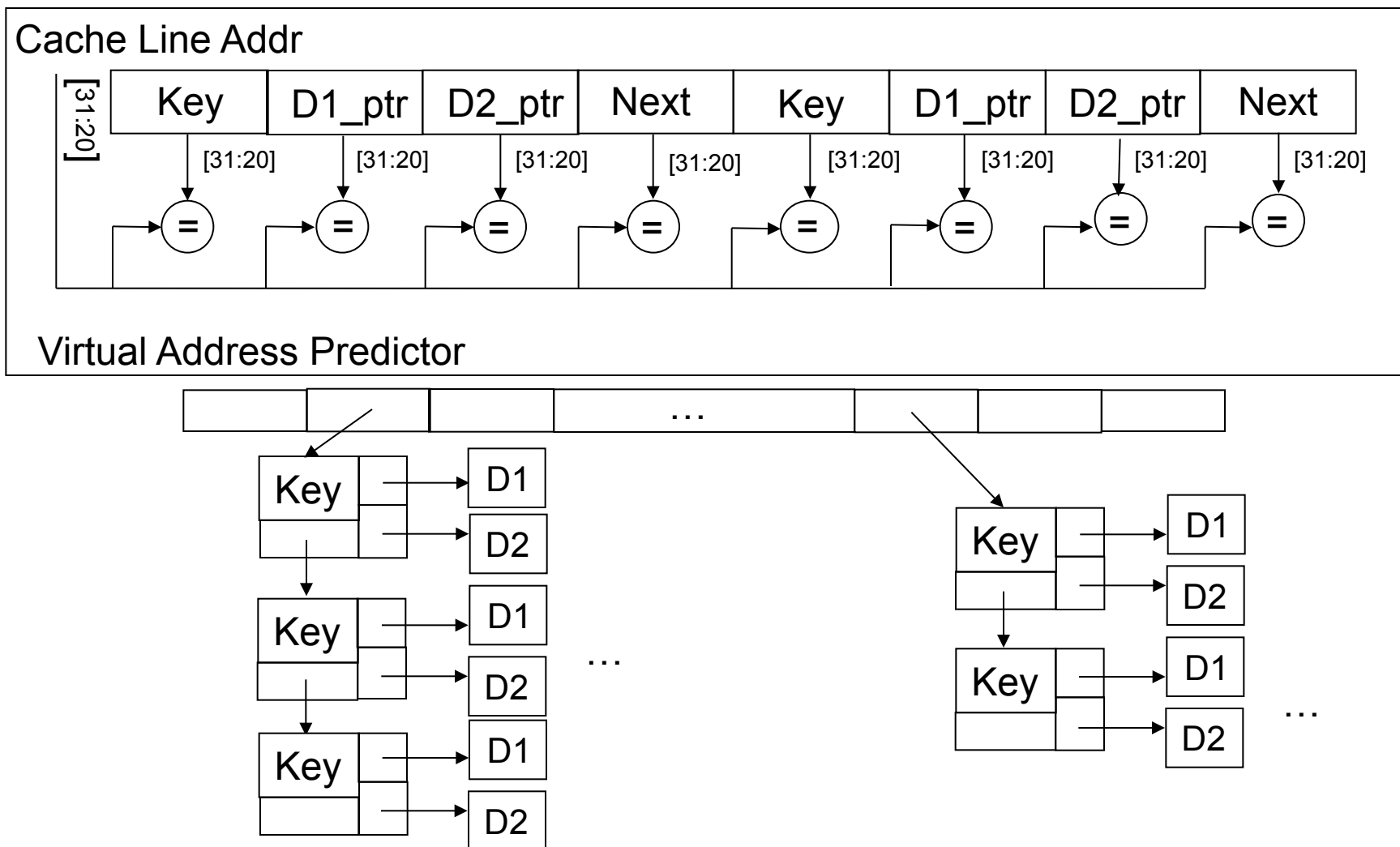
```
HashLookup(int Key) {
    ...
    for (node = head ; node -> Key != Key; node = node -> Next; ) ;
    if (node) return node->D1;
}
```

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



Example from mst

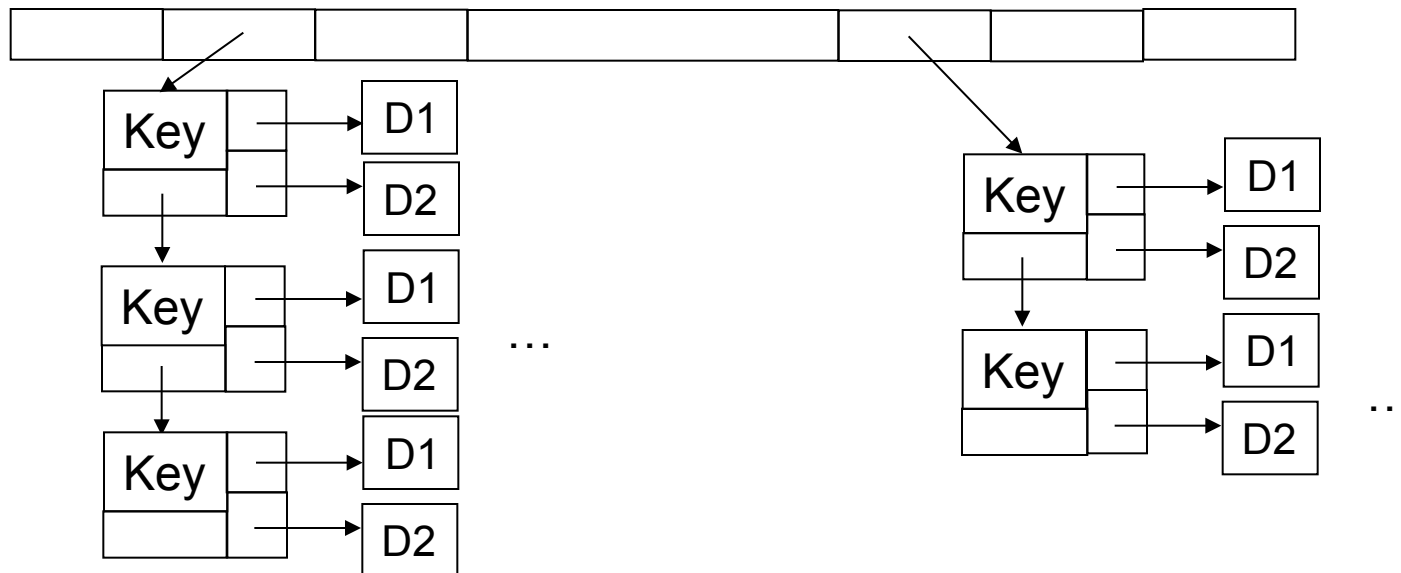
# Shortcomings of CDP – An example



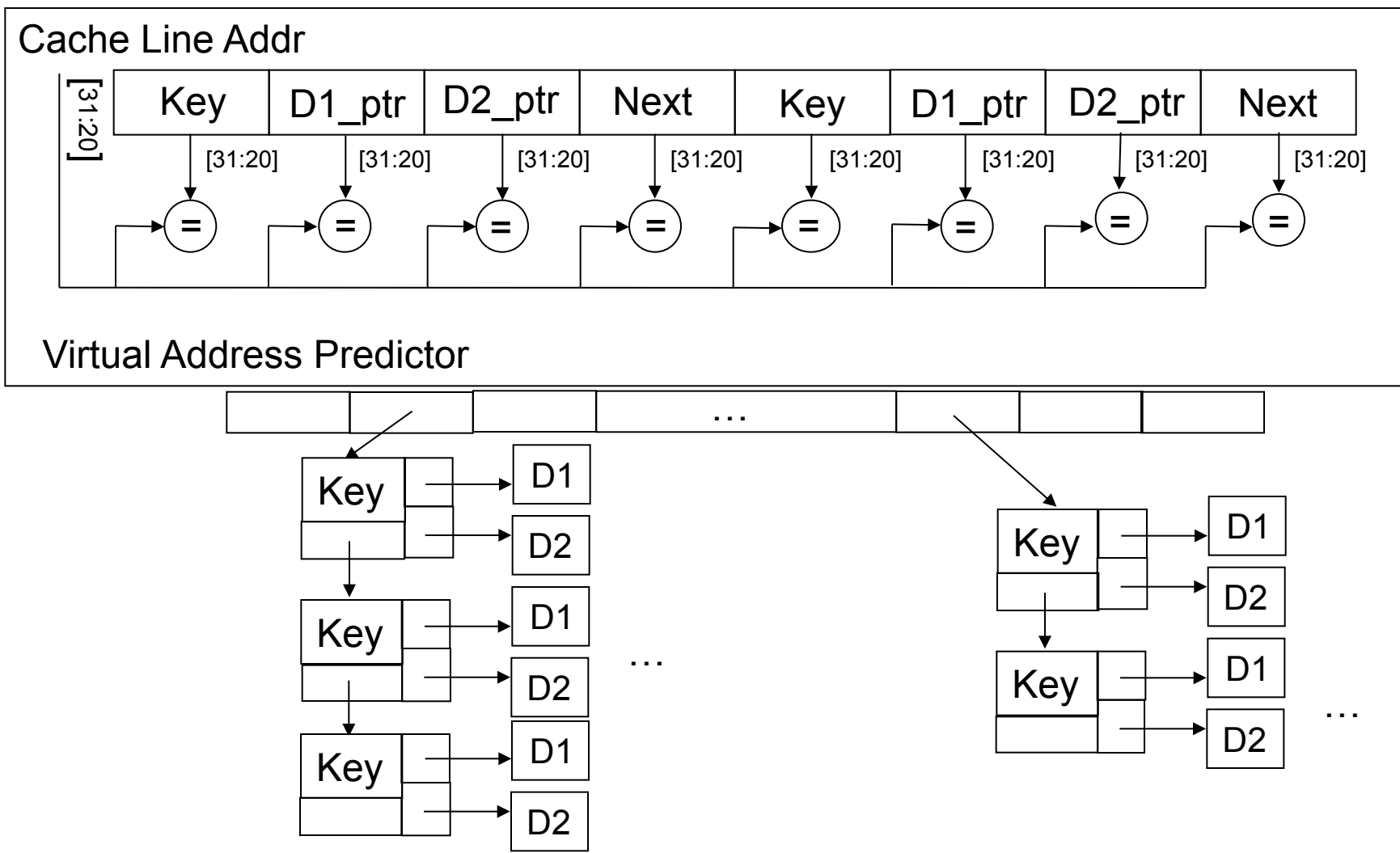
# Shortcomings of CDP – An example

---

```
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)

---

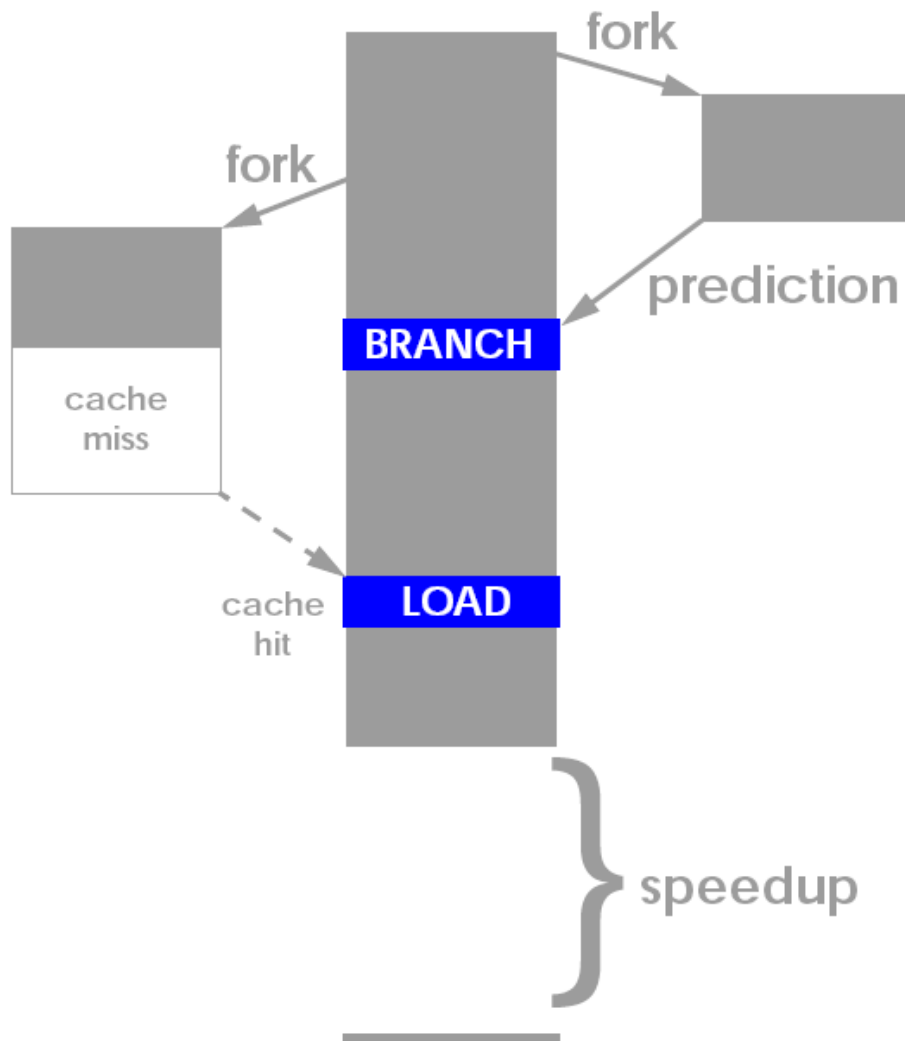
- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses
- **Speculative thread:** Pre-executed program piece can be considered a "thread"
- Speculative thread can be executed
  - On a separate processor/core
  - On a separate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

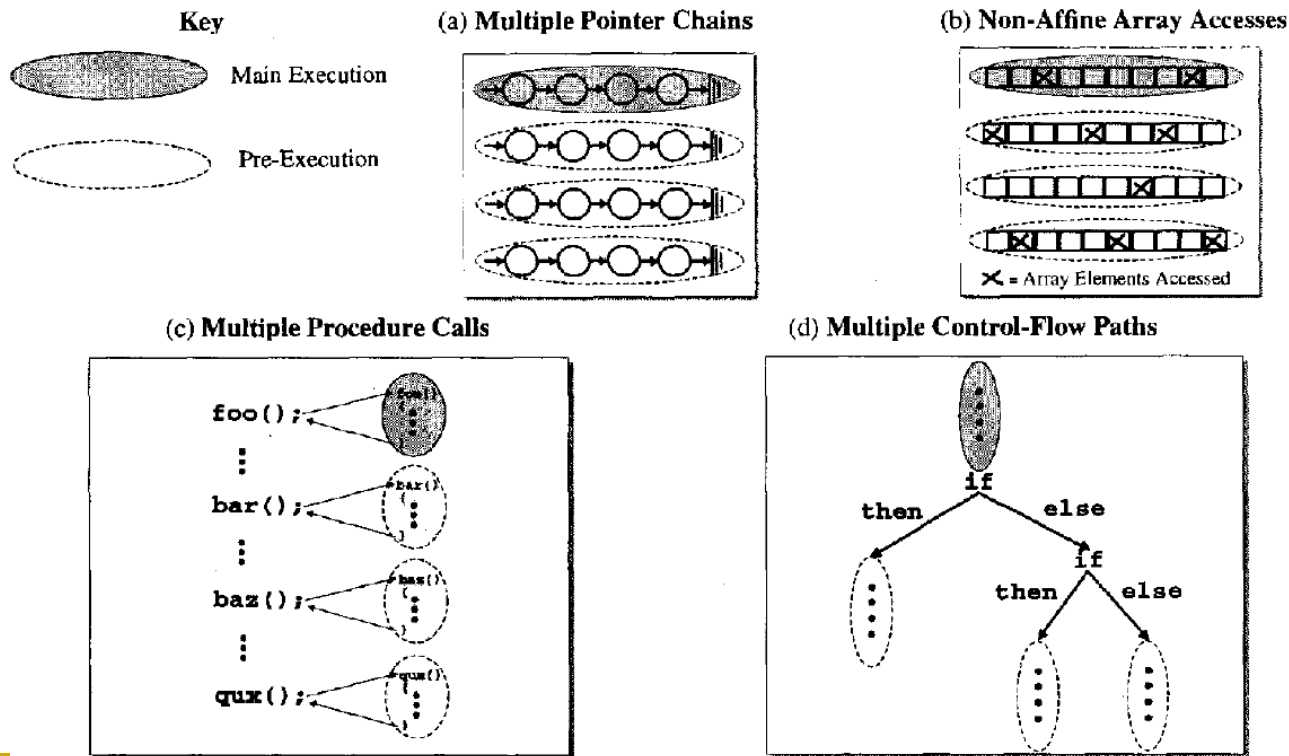
# Thread-Based Pre-Execution Issues

---

- **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.

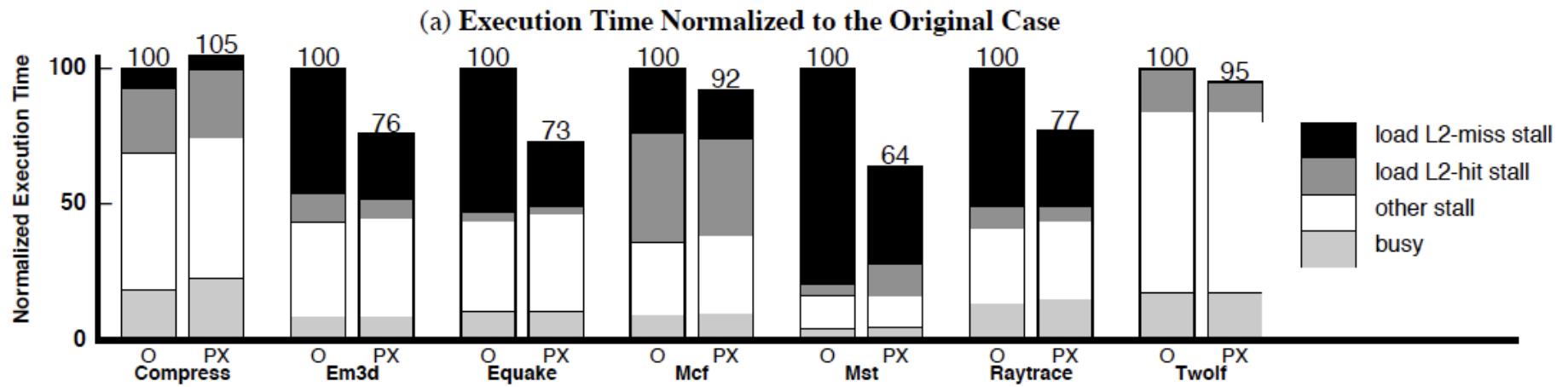
***PreExecute\_Stop():*** The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

***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

---

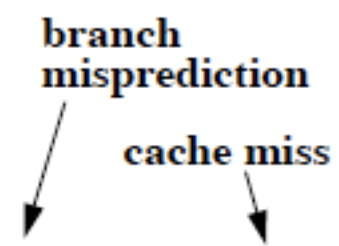
**Figure 2.** Example problem instructions from heap insertion routine in *vpr*.

```
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  cmplt  $f0,$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  cmplt  $f1,$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
```

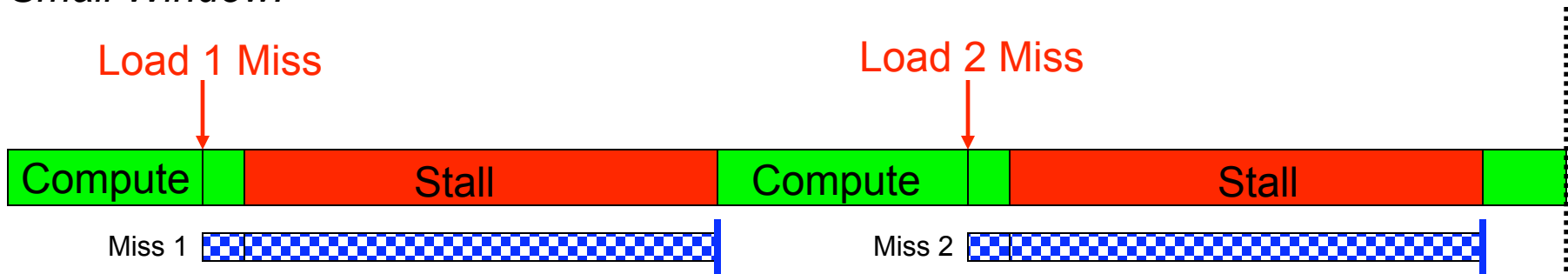
# Runahead Execution (I)

---

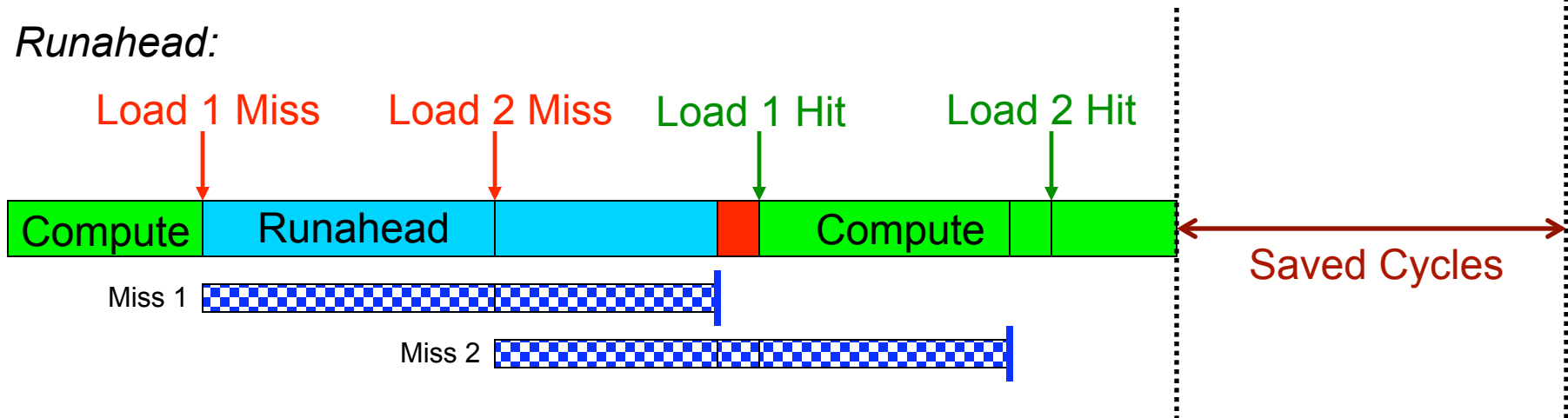
- A simple pre-execution method for prefetching purposes
- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - **Speculatively pre-execute instructions**
  - **The purpose of pre-execution is to generate prefetches**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