18-447
Computer Architecture
Lecture 27: Prefetching

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 4/9/2013
Announcements

- Graded homework and labs
  - You can find grade distributions on the website

- Lab 6: Memory Hierarchy
  - Due April 20

- HW 6: Due today!

- HW 7: Will be out soon.
  - Please do the homework to prepare for Midterm II

- Midterm II: April 23 – start preparing now
  - Similar in format and spirit to Midterm I. Solve past midterms.
Suggestions for Midterm II

- Solve past midterms (and finals) on your own...
  - And, check your solutions vs. the online solutions
  - Questions will be similar in spirit


- Do Homework 7

- Study and internalize the lecture material well.

- Do the readings that are required.
Lab 6: Memory Hierarchy

- Due Wednesday (April 20)
- Cycle-level modeling of L2 cache and DRAM-based main memory

- Extra credit: Prefetching
  - Design your own hardware prefetcher to improve system performance
Last Lecture

- Start Memory Latency Tolerance

- Runahead Execution and Enhancements
  - Efficient Runahead Execution
  - Address-Value Delta Prediction
Today

- Basics of Prefetching
- Advanced Prefetching
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 prefetched / sent prefetcheds

- **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 $\rightarrow$ cache pollution
  - In a separate **prefetch buffer**
    + Demand data protected from prefetches $\rightarrow$ 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
- Prefetch instructions prefetch data into caches
- Compiler or programmer can insert such instructions into the program
X86 PREFETCH Instruction

**PREFETCHh—Prefetch Data Into Caches**

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

**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
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

```c
for (i=0; i<N; i++) {
    __prefetch(a[i+8]);
    __prefetch(b[i+8]);
    sum += a[i]*b[i];
}
```

```c
while (p) {
    __prefetch(p->next);
    work(p->data);
    p = p->next;
}
```

```c
while (p) {
    __prefetch(p->next->next->next);
    work(p->data);
    p = p->next;
}
```

Which one is better?
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:
  - 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 \textit{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 (last address + N*stride)
    - Generate multiple prefetches
Cache-Block Address Based Stride Prefetching

- Can detect
  - A, A+N, A+2N, A+3N, ...
  - **Stream buffers** are a special case of cache block address based stride prefetching where N = 1
- 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 prefetchedes / sent prefetchedes)
- **Coverage** (prefetched misses / all misses)
- **Timeliness** (on-time prefetchedes / used prefetchedes)

**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)

Percentage IPC change over No Prefetching

Prefetcher Accuracy
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

<table>
<thead>
<tr>
<th>High Accuracy</th>
<th>Med Accuracy</th>
<th>Low Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not-Late</td>
<td>Late</td>
<td>Not-Poll</td>
</tr>
<tr>
<td>Polluting</td>
<td>Not-Poll</td>
<td>Not-Poll</td>
</tr>
<tr>
<td>Decrease</td>
<td>Decrease</td>
<td>Decrease</td>
</tr>
<tr>
<td>Increase</td>
<td>Late</td>
<td>Polluting</td>
</tr>
<tr>
<td></td>
<td>Decrease</td>
<td>Not-Late</td>
</tr>
<tr>
<td></td>
<td>Increase</td>
<td>No Change</td>
</tr>
</tbody>
</table>

- Increase: throttle up
- Decrease: throttle down
- No Change: do nothing
- Late: prefetches are late
- Not-Late: prefetches are not late
- Polluting: prefetches are polluting
- Not-Poll: prefetches are not polluting
- Decrease: decrease aggressiveness
- Increase: increase aggressiveness
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 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: \( (A, B) \rightarrow (C) \)
  - \( (A, B) \) correlated with C
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
- **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)

Virtual Address Predictor

Generate Prefetch
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.

Efficient CDP – An Example

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

Example from mst
Efficient CDP – An Example

Cache Line Addr

Virtual Address Predictor
HashLookup(int Key) {
    ...
    for (node = head; node -> Key != Key; node = node -> Next; ) {
        if (node) return node -> D1;
    }
}
Efficient CDP – An Example

Cache Line Addr

Virtual Address Predictor
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

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**
  - Many issues in software-based pre-execution discussed

![Diagram of Thread-Based Pre-Execution Issues](image)
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;
}
...
// 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();

(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.
**Example ISA Extensions**

\[
Thread\_ID = \textbf{PreExecute\_Start} (Start\_PC, Max\_Insts):
\]

Request for an idle context to start pre-execution at \textit{Start\_PC} and stop when \textit{Max\_Insts} instructions have been executed; \textit{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.

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

\textbf{PreExecute\_Cancel}(Thread\_ID): Terminate the pre-execution thread with \textit{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

(a) Execution Time Normalized to the Original Case

- Compress
- Em3d
- Equake
- Mcf
- Mst
- Raytrace
- Twolf

Normalized Execution Time

- Load L2-miss stall
- Load L2-hit stall
- Other stall
- Busy
Problem Instructions


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

```c
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) &&
     (heap[ifrom]->cost < heap[ito]->cost))
6.     struct s_heap *temp_ptr = heap[ito];
7.     heap[ito] = heap[ifrom];
8.     heap[ifrom] = temp_ptr;
9.     ifrom = ito;
10.    ito = ifrom/2;
11. }
```
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`.

```c
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 ld1 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
3 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]
10 cmplt t4, 0, t9    # see note
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 cmptlt $f0,$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 ld1 $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
```
Review: Runahead Execution

- 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

Review: Runahead Execution (Mutlu et al., HPCA 2003)

**Small Window:**

```
Compute  | Stall | Compute  | Stall
         |       |          |
Miss 1   |       | Miss 1   |
         |       | Miss 2   |
```

**Runahead:**

```
Compute  | Runahead | Compute
         |          |
Miss 1   |          | Miss 1   |
         |          | Miss 2   |
```

---

**Saved Cycles**