Recall: Outline of Prefetching Lecture(s)

- 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 (if we get to it)**
More on Content Directed Prefetching

- Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems"
  Best paper session. One of the three papers nominated for the Best Paper Award by the Program Committee.

Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems

Eiman Ebrahimi† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
  The University of Texas at Austin
  {ebrahimi, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
  Carnegie Mellon University
  onur@cmu.edu
Recall: 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 of the main thread
    - Performs only address generation computation, branch prediction, value prediction (to predict “unknown” values)
  - Purely speculative so there is no need for recovery of main program if the speculative thread is incorrect
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 (recall throttling)
Thread-Based Pre-Execution Issues

- What, when, where, how
  - 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
    \rightarrow tail \rightarrow mark;
// traverse the list starting with
// the first node just assigned
while (arcin) {
    tail = arcin \rightarrow tail;
    ...
    arcin = (arc_t *)tail \rightarrow 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
    \rightarrow tail \rightarrow mark;
// traverse the list starting with
// the first node just assigned
while (arcin) {
    tail = arcin \rightarrow tail;
    ...
    arcin = (arc_t *)tail \rightarrow 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 a Multithreaded Processor

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;
}
```
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 Thread 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.

```assembly
node_to_heap:
... /* skips ~40 instructions */
  2 distra 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
  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, 0(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 # < heap[ito]->cost)
  6 fbeq $f0, return # # (ito >= 1)
  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.

```assembly	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
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:

Runahead:

Saved Cycles
Benefits of Runahead Execution

Instead of stalling during an L2 cache miss:

- Pre-executed loads and stores independent of L2-miss instructions generate very accurate data prefetches:
  - For both regular and irregular access patterns

- Instructions on the predicted program path are prefetched into the instruction/trace cache and L2.

- Hardware prefetcher and branch predictor tables are trained using future access information.
Runahead Execution Mechanism

- Entry into runahead mode
  - Checkpoint architectural register state

- Instruction processing in runahead mode

- Exit from runahead mode
  - Restore architectural register state from checkpoint
Runahead mode processing is the same as normal instruction processing, EXCEPT:

- It is purely speculative: Architectural (software-visible) register/memory state is NOT updated in runahead mode.

- L2-miss dependent instructions are identified and treated specially.
  - They are quickly removed from the instruction window.
  - Their results are not trusted.
L2-Miss Dependent Instructions

- Two types of results produced: INV and VALID
- INV = Dependent on an L2 miss
- INV results are marked using INV bits in the register file and store buffer.
- INV values are not used for prefetching/branch resolution.
Removal of Instructions from Window

- Oldest instruction is examined for **pseudo-retirement**
  - An INV instruction is removed from window immediately.
  - A VALID instruction is removed when it completes execution.

- **Pseudo-retired instructions free their allocated resources.**
  - This allows the processing of later instructions.

- **Pseudo-retired stores communicate their data to dependent loads.**
Store/Load Handling in Runahead Mode

- A pseudo-retired store writes its data and INV status to a dedicated memory, called runahead cache.

- Purpose: Data communication through memory in runahead mode.

- A dependent load reads its data from the runahead cache.

- Does not need to be always correct → Size of runahead cache is very small.
Branch Handling in Runahead Mode

- **INV branches cannot be resolved.**
  - A mispredicted INV branch causes the processor to stay on the wrong program path until the end of runahead execution.

- **VALID branches are resolved and initiate recovery if mispredicted.**
A Runahead Processor Diagram

Mutlu+, “Runahead Execution,”
HPCA 2003.
Runahead Execution Pros and Cons

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

- **Disadvantages/Limitations:**
  -- **Extra executed instructions**
  -- Limited by branch prediction accuracy
  -- Cannot prefetch dependent cache misses
  -- Effectiveness limited by available “memory-level parallelism” (MLP)
  -- Prefetch distance (how far ahead to prefetch) limited by memory latency

- **Implemented in IBM POWER6, Sun “Rock”**
Performance of Runahead Execution

- No prefetcher, no runahead
- Only prefetcher (baseline)
- Only runahead
- Prefetcher + runahead

Bars represent Micro-operations Per Cycle for different workloads and configurations:

- S95: 12%
- FP00: 35%
- INT00: 15%
- WEB: 22%
- MM: 12%
- PROD: 16%
- SERV: 52%
- WS: 22%
- AVG
Runahead on In-order vs. Out-of-order

- In-order baseline
- In-order + runahead
- Out-of-order baseline
- Out-of-order + runahead

Micro-operations Per Cycle

<table>
<thead>
<tr>
<th>Application</th>
<th>Micro-operations Per Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>S95</td>
<td>1.1</td>
</tr>
<tr>
<td>FP00</td>
<td>0.75</td>
</tr>
<tr>
<td>INT00</td>
<td>0.65</td>
</tr>
<tr>
<td>WEB</td>
<td>0.55</td>
</tr>
<tr>
<td>MM</td>
<td>0.5</td>
</tr>
<tr>
<td>PROD</td>
<td>0.62</td>
</tr>
<tr>
<td>SERV</td>
<td>0.7</td>
</tr>
<tr>
<td>WS</td>
<td>0.75</td>
</tr>
<tr>
<td>AVG</td>
<td>0.74</td>
</tr>
</tbody>
</table>
More on Runahead Execution (Short)


Runahead Execution: An Effective Alternative to Large Instruction Windows
High-Performance Throughput Computing

Throughput computing, achieved through multithreading and multicore technology, can lead to performance improvements that are 10 to 30× those of conventional processors and systems. However, such systems should also offer good single-thread performance. Here, the authors show that hardware scouting increases the performance of an already robust core by up to 40 percent for commercial benchmarks.

More on Runahead in SUN ROCK

Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun’s ROCK Processor

Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay
Sun Microsystems, Inc.
4180 Network Circle, Mailstop SCA18-211
Santa Clara, CA 95054, USA
{shailender.chaudhry, robert.cypher, magnus.ekman, martin.karlsson, anders.landin, sherman.yip, haakan.zeffer, marc.tremblay}@sun.com

Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor

Harold W. Cain        Priya Nagpurkar
IBM T.J. Watson Research Center
Yorktown Heights, NY
{tcain, pnagpurkar}@us.ibm.com

Cain+, “Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor,” ISPASS 2010
Runahead Execution in IBM POWER6

Abstract

After many years of prefetching research, most commercially available systems support only two types of prefetching: software-directed prefetching and hardware-based prefetchers using simple sequential or stride-based prefetching algorithms. More sophisticated prefetching proposals, despite promises of improved performance, have not been adopted by industry. In this paper, we explore the efficacy of both hardware and software prefetching in the context of an IBM POWER6 commercial server. Using a variety of applications that have been compiled with an aggressively optimizing compiler to use software prefetching when appropriate, we perform the first study of a new runahead prefetching feature adopted by the POWER6 design, evaluating it in isolation and in conjunction with a conventional hardware-based sequential stream prefetcher and compiler-inserted software prefetching.

We find that the POWER6 implementation of runahead prefetching is quite effective on many of the memory intensive applications studied; in isolation it improves performance as much as 36% and on average 10%. However, it outperforms the hardware-based stream prefetcher on only two of the benchmarks studied, and in those by a small margin. When used in conjunction with the conventional prefetching mechanisms, the runahead feature adds an additional 6% on average, and 39% in the best case (GemsFDTD).
Runahead Enhancements
Readings

- **Required**

- **Recommended**
Limitations of the Baseline Runahead Mechanism

- **Energy Inefficiency**
  - A large number of instructions are speculatively executed
  - Efficient Runahead Execution [ISCA’ 05, IEEE Micro Top Picks’ 06]

- **Ineffectiveness for pointer-intensive applications**
  - Runahead cannot parallelize dependent L2 cache misses
  - Address-Value Delta (AVD) Prediction [MICRO’ 05]

- **Irresolvable branch mispredictions in runahead mode**
  - Cannot recover from a mispredicted L2-miss dependent branch
  - Wrong Path Events [MICRO’ 04]
The Efficiency Problem

- % Increase in IPC
- % Increase in Executed Instructions

- % Increase in IPC: 235%
- % Increase in Executed Instructions: 22%
Causes of Inefficiency

- Short runahead periods
- Overlapping runahead periods
- Useless runahead periods

Short Runahead Periods

- Processor can initiate runahead mode due to an already in-flight L2 miss generated by
  - the prefetcher, wrong-path, or a previous runahead period

- Short periods
  - are less likely to generate useful L2 misses
  - have high overhead due to the flush penalty at runahead exit
Overlapping Runahead Periods

- Two runahead periods that execute the same instructions

- Second period is inefficient
Useless Runahead Periods

- Periods that do not result in prefetches for normal mode

- They exist due to the lack of memory-level parallelism

- Mechanism to eliminate useless periods:
  - Predict if a period will generate useful L2 misses
  - Estimate a period to be useful if it generated an L2 miss that cannot be captured by the instruction window

- Useless period predictors are trained based on this estimation
Overall Impact on Executed Instructions

- Increase in Executed Instructions
  - baseline runahead
  - all techniques

- Overall Impact on Executed Instructions
  - AVG: 26.5%
  - 6.2%
Overall Impact on IPC

- Increase in IPC for each benchmark
  - baseline runahead
  - all techniques

- AVG increase: 22.6%
- AVG overall: 22.1%
More on Efficient Runahead Execution

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
  "Techniques for Efficient Processing in Runahead Execution Engines"
  One of the 13 computer architecture papers of 2005 selected as Top Picks by IEEE Micro.

Techniques for Efficient Processing in Runahead Execution Engines

Onur Mutlu  Hyesoon Kim  Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu
More on Efficient Runahead Execution


Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance
Limitations of the Baseline Runahead Mechanism

- **Energy Inefficiency**
  - A large number of instructions are speculatively executed
  - Efficient Runahead Execution [ISCA’ 05, IEEE Micro Top Picks’ 06]

- **Ineffectiveness for pointer-intensive applications**
  - Runahead cannot parallelize dependent L2 cache misses
  - Address-Value Delta (AVD) Prediction [MICRO’ 05]

- **Irresolvable branch mispredictions in runahead mode**
  - Cannot recover from a mispredicted L2-miss dependent branch
  - Wrong Path Events [MICRO’ 04]
Runahead execution cannot parallelize dependent misses
- wasted opportunity to improve performance
- wasted energy (useless pre-execution)

Runahead performance would improve by 25% if this limitation were ideally overcome
Parallelizing Dependent Cache Misses

- **Idea:** Enable the parallelization of dependent L2 cache misses in runahead mode with a low-cost mechanism.

- **How:** Predict the values of L2-miss **address (pointer) loads**
  - **Address load:** loads an address into its destination register, which is later used to calculate the address of another load.
  - as opposed to **data load**

- **Read:**
Parallelizing Dependent Cache Misses

Cannot Compute Its Address!

Value Predicted

Can Compute Its Address

Saved Speculative Instructions

Saved Cycles
More on AVD Prediction


One of the five papers nominated for the Best Paper Award by the Program Committee.

Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns

Onur Mutlu    Hyesoon Kim    Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu
More on AVD Prediction (II)


Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses

Onur Mutlu, Member, IEEE, Hyesoon Kim, Student Member, IEEE, and Yale N. Patt, Fellow, IEEE
Even More on Runahead Execution

- Lecture video from Fall 2017
  - [https://www.youtube.com/watch?v=Kj3relihGF4](https://www.youtube.com/watch?v=Kj3relihGF4)

- Onur Mutlu, "Efficient Runahead Execution Processors"

  Nominated for the ACM Doctoral Dissertation Award by the University of Texas at Austin.
Runahead as an Execution-Based Prefetcher
Runahead as an Execution-based Prefetcher

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

- Idea of Runahead: Pre-execute the main program solely for prefetching data

- Advantages and disadvantages of runahead vs. other execution-based prefetchers?

- Can you make runahead even better by pruning the program portion executed in runahead mode?
Taking Advantage of Pure Speculation

- Runahead mode is purely speculative

- The goal is to find and generate cache misses that would otherwise stall execution later on

- How do we achieve this goal most efficiently and with the highest benefit?

- Idea: Find and execute only those instructions that will lead to cache misses (that cannot already be captured by the instruction window)

- How?
Execution-based Prefetchers: Pros and Cons

+ 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

-- Complexity in deciding when and what to pre-execute
Multi-Core Issues in Prefetching
Prefetching in Multi-Core (I)

- **Prefetching shared data**
  - Coherence misses

- **Prefetch efficiency is a lot more important**
  - Bus bandwidth more precious
  - Cache space more valuable

- **One cores’ prefetches interfere with other cores’ requests**
  - Cache conflicts
  - Bus contention
  - DRAM bank and row buffer contention
Two key issues

- How to prioritize prefetches vs. demands (of different cores)
- How to control the aggressiveness of multiple prefetchers to achieve high overall performance

Need to coordinate the actions of independent prefetchers for best system performance

- Each prefetcher has different accuracy, coverage, timeliness
Some Examples

- **Controlling prefetcher aggressiveness**
  - Feedback directed prefetching [HPCA’07]
  - Coordinated control of multiple prefetchers [MICRO’09]

- **How to prioritize prefetches vs. demands from cores**
  - Prefetch-aware memory controllers and shared resource management [MICRO’08, ISCA’11]

- **Bandwidth efficient prefetching of linked data structures**
  - Through hardware/software cooperation (software hints) [HPCA’09]
More on Feedback Directed Prefetching


One of the five papers nominated for the Best Paper Award by the Program Committee.

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

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

†Microsoft ssri@microsoft.com §Microsoft Research onur@microsoft.com ‡Department of Electrical and Computer Engineering
The University of Texas at Austin
{santhosh, hyesoon, patt}@ece.utexas.edu
On Bandwidth-Efficient Prefetching

Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt,
"Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems"

Best paper session. One of the three papers nominated for the Best Paper Award by the Program Committee.

Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems

Eiman Ebrahimi† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu
More on Coordinated Prefetcher Control


Coordinated Control of Multiple Prefetchers in Multi-Core Systems

Eiman Ebrahimi† Onur Mutlu§ Chang Joo Lee† Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu
More on Prefetching in Multi-Core (I)


Prefetch-Aware DRAM Controllers

Chang Joo Lee† Onur Mutlu§ Veynu Narasiman† Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
cjlee, narasima, patt}@ece.utexas.edu

§Microsoft Research and Carnegie Mellon University
onur@{microsoft.com,cmu.edu}
More on Prefetching in Multi-Core (II)

- Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching"

---

Improving Memory Bank-Level Parallelism in the Presence of Prefetching

Chang Joo Lee† Veynu Narasiman† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
  The University of Texas at Austin
  {cjlee, narasima, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
  Carnegie Mellon University
  onur@cmu.edu
More on Prefetching in Multi-Core (III)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, "Prefetch-Aware Shared Resource Management for Multi-Core Systems"
  Proceedings of the 38th International Symposium on Computer Architecture (ISCA), San Jose, CA, June 2011. Slides (pptx)

Prefetch-Aware Shared-Resource Management for Multi-Core Systems

Eiman Ebrahimi†  Chang Joo Lee‡‡  Onur Mutlu§  Yale N. Patt†

†HPS Research Group  ‡Intel Corporation  §Carnegie Mellon University
The University of Texas at Austin  chang.joo.lee@intel.com  onur@cmu.edu

{ebrahimi, patt}@hps.utexas.edu
More on Prefetching in Multi-Core (IV)

- Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry,

"Mitigating Prefetcher-Caused Pollution using Informed Caching Policies for Prefetched Blocks"

*ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 11, No. 4, January 2015.

Presented at the 10th HiPEAC Conference, Amsterdam, Netherlands, January 2015.

[Slides (pptx) (pdf)]

[Source Code]

Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks

VIVEK SESHADRI, SAMIHAN YEDKAR, HONGYI XIN, and ONUR MUTLU, Carnegie Mellon University

PHILLIP B. GIBBONS and MICHAEL A. KOZUCH, Intel Pittsburgh

TODD C. MOWRY, Carnegie Mellon University
Prefetching in GPUs

- Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das,

"Orchestrated Scheduling and Prefetching for GPGPUs"
Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. Slides (pptx) Slides (pdf)

Orchestrated Scheduling and Prefetching for GPGPUs

Adwait Jog† Onur Kayiran† Asit K. Mishra§ Mahmut T. Kandemir†
Onur Mutlu‡ Ravishankar Iyer§ Chita R. Das†

†The Pennsylvania State University
University Park, PA 16802
{adwait, onur, kandemir, das}@cse.psu.edu
‡Carnegie Mellon University
Pittsburgh, PA 15213
onur@cmu.edu
§Intel Labs
Hillsboro, OR 97124
{asit.k.mishra, ravishankar.iyer}@intel.com
More on Multi-Core Issues in Prefetching
Prefetching in Multi-Core (I)

- Prefetching shared data
  - Coherence misses

- Prefetch efficiency is a lot more important
  - Bus bandwidth more precious
  - Cache space more valuable

- One cores’ prefetches interfere with other cores’ requests
  - Cache conflicts
  - Bus contention
  - DRAM bank and row buffer contention
Prefetching in Multi-Core (II)

- Two key issues
  - How to prioritize prefetches vs. demands (of different cores)
  - How to control the aggressiveness of multiple prefetchers to achieve high overall performance

- Need to *coordinate the actions of independent prefetchers* for best system performance
  - Each prefetcher has different accuracy, coverage, timeliness
Some Ideas

- **Controlling prefetcher aggressiveness**
  - Feedback directed prefetching [HPCA’07]
  - Coordinated control of multiple prefetchers [MICRO’09]

- **How to prioritize prefetches vs. demands from cores**
  - Prefetch-aware memory controllers and shared resource management [MICRO’08, ISCA’11]

- **Bandwidth efficient prefetching of linked data structures**
  - Through hardware/software cooperation (software hints) [HPCA’09]
Motivation

- Aggressive prefetching improves memory latency tolerance of many applications when they run alone.

- Prefetching for concurrently-executing applications on a CMP can lead to:
  - Significant system performance degradation and bandwidth waste.

- Problem:
  Prefetcher-caused inter-core interference
  - Prefetches of one application contend with prefetches and demands of other applications.
Potential Performance

System performance improvement of *ideally* removing all prefetcher-caused inter-core interference in shared resources

Exact workload combinations can be found in [Ebrahimi et al., MICRO 2009]
High Interference caused by Accurate Prefetchers

In a multi-core system, accurate prefetchers can cause significant interference with concurrently-executing applications.
Shortcoming of Local Prefetcher Throttling

Local-only prefetcher control techniques have no mechanism to detect inter-core interference.
Shortcoming of Local-Only Prefetcher Control

4-core workload example: lbm_06 + swim_00 + crafty_00 + bzip2_00

Our Approach: Use both *global* and per-core feedback to determine each prefetcher’s aggressiveness
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
Hierarchical Prefetcher Throttling

Local Control: goals are to:
- Maximize the prefetching performance of core \( i \) independently
- Override the local control to improve overall system performance

Global control’s goal: Keep track of and control prefetter-caused inter-core interference in shared memory system

- Pref. \( i \)
- Local Control
- Core \( i \)

- Global Control
- Memory Controller
- Bandwidth Feedback
- Cache Pollution Feedback

Accuray

Final Throttling Decision

Local Throttling Decision
Hierarchical Prefetcher Throttling Example

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

Memory Controller

High BW (i)

High BWNO (i)

Global Control

High Acc (i)

Local Throttling Decision

Core i

Pref. i

Local Control

High Pol (i)

Pol. Filter i

Shared Cache
HPAC Control Policies

<table>
<thead>
<tr>
<th>Pol ((i))</th>
<th>Acc ((i))</th>
<th>BW ((i))</th>
<th>BWNO ((i))</th>
<th>Interference Class</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causing Low Pollution</td>
<td>Inaccurate</td>
<td>Low BW Consumption</td>
<td>Others’ low BW need</td>
<td>Severe interference</td>
<td>throttle down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High BW Consumption</td>
<td>Others’ high BW need</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Highly Accurate</td>
<td>Low BW Consumption</td>
<td>Others’ low BW need</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inaccurate</td>
<td>Low BW Consumption</td>
<td>Others’ low BW need</td>
<td>Severe interference</td>
<td>throttle down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High BW Consumption</td>
<td>Others’ high BW need</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Causing High Pollution</td>
<td>Inaccurate</td>
<td>Low BW Consumption</td>
<td>Others’ low BW need</td>
<td>Severe interference</td>
<td>throttle down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>High BW Consumption</td>
<td>Others’ high BW need</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Highly Accurate</td>
<td>Low BW Consumption</td>
<td>Others’ high BW need</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inaccurate</td>
<td>High BW Consumption</td>
<td>Others’ high BW need</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low BW Consumption</td>
<td>Others’ low BW need</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HPAC Evaluation

- No Throttling
- Feedback-Directed Prefetching (FDP)
- Hierarchical Prefetcher Aggressiveness Control (HPAC)

Normalized System Performance

Normalized System Unfairness

Normalized to system with no prefetching

9%

15%
More on Coordinated Prefetcher Control

- Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-Core Systems"

Coordinated Control of Multiple Prefetchers in Multi-Core Systems

Eiman Ebrahimi† Onur Mutlu§ Chang Joo Lee† Yale N. Patt†

†Department of Electrical and Computer Engineering The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM) Carnegie Mellon University
onur@cmu.edu
More on Prefetching in Multi-Core (I)

- Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt, "Prefetch-Aware DRAM Controllers"


Prefetch-Aware DRAM Controllers

Chang Joo Lee†  Onur Mutlu§  Veynu Narasiman†  Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{cjlee, narasima, patt}@ece.utexas.edu

§Microsoft Research and Carnegie Mellon University
onur@{microsoft.com,cmu.edu}
Problems of Prefetch Handling

- How to schedule *prefetches vs demands*?
  - Demand-first: Always prioritizes demands over prefetch requests
  - Demand-prefetch-equal: Always treats them the same

Neither of these perform best

Neither take into account both:
1. Non-uniform access latency of DRAM systems
2. Usefulness of prefetches
When Prefetches are Useful

Prefetch Row A: X
Demand Row B: Y
Prefetch Row A: Z

DRAM
Controller
Row Buffer
Row B

Demand-first

2 row-conflicts, 1 row-hit

DRAM
Processor
Miss Y
Miss X
Miss Z

Processor needs Y, X, and Z
When Prefetches are Useful

Prefetch Row A: X
Dem Row B: Y
Pref Row A: Z

DRAM Controller
Row Buffer

DRAM
Processor

Demand-first
2 row-conflicts, 1 row-hit

Demand-pref-equal outperforms demand-first

Demand-pref-equal
2 row-hits, 1 row-conflict

Stall
Execution

Processor needs Y, X, and Z

Miss Y
Hit X Hit Z

Saved Cycles
When Prefetches are Useless

Demand-first outperforms demand-pref-equal

Processor needs ONLY Y
Demand-first vs. Demand-pref-equal policy

Stream prefetcher enabled

Goal 1: Adaptively schedule prefetches based on prefetch usefulness
Goal 2: Eliminate useless prefetches

Useless prefetches: Off-chip bandwidth, Queue resources, Cache Pollution

IPC normalized to no prefetching
More on Prefetching in Multi-Core (II)

- Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt, "Improving Memory Bank-Level Parallelism in the Presence of Prefetching"

Improving Memory Bank-Level Parallelism in the Presence of Prefetching

Chang Joo Lee†  Veynu Narasiman†  Onur Mutlu§  Yale N. Patt†

†Department of Electrical and Computer Engineering
  The University of Texas at Austin
  {cjlee, narasima, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
  Carnegie Mellon University
  onur@cmu.edu
More on Prefetching in Multi-Core (III)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, "Prefetch-Aware Shared Resource Management for Multi-Core Systems"
  Proceedings of the 38th International Symposium on Computer Architecture (ISCA), San Jose, CA, June 2011. Slides (pptx)

Prefetch-Aware Shared-Resource Management for Multi-Core Systems

Eiman Ebrahimi†  Chang Joo Lee††  Onur Mutlu§  Yale N. Patt†

†HPS Research Group
The University of Texas at Austin
{ebrahimi, patt}@hps.utexas.edu

†Intel Corporation
chang.joo.lee@intel.com

§Carnegie Mellon University
onur@cmu.edu
More on Prefetching in Multi-Core (IV)

- Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
"Mitigating Prefetcher-Caused Pollution using Informed Caching Policies for Prefetched Blocks"
*ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 11, No. 4, January 2015.
Presented at the 10th HiPEAC Conference, Amsterdam, Netherlands, January 2015.
[Slides (pptx) (pdf)]
[Source Code]

Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks

VIVEK SESHADRI, SAMIHAN YEDKAR, HONGYI XIN, and ONUR MUTLU, Carnegie Mellon University
PHILLIP B. GIBBONS and MICHAEL A. KOZUCH, Intel Pittsburgh
TODD C. MOWRY, Carnegie Mellon University
Problem: Existing caching policies for prefetched blocks result in significant cache pollution.
Prefetch Usage Experiment

Classify prefetched blocks into three categories

1. Blocks that are unused
2. Blocks that are used exactly once before evicted from cache
3. Blocks that are used more than once before evicted from cache
Many applications have a significant fraction of inaccurate prefetches.

95% of the useful prefetched blocks are used only once!

Typically, large data structures benefit repeatedly from prefetching. Blocks of such data structures are unlikely to be used more than once!
Shortcoming of Traditional Promotion Policy

This is a **bad** policy. The block is unlikely to be reused in the cache.

This problem exists with state-of-the-art replacement policies (e.g., DRRIP, DIP)
Demotion of Prefetched Block

Ensures that the block is evicted from the cache quickly after it is used!

Only requires the cache to distinguish between prefetched blocks and demand-fetched blocks.
Cache Insertion Policy for Prefetched Blocks

- Good (Accurate prefetch)
- Bad (Inaccurate prefetch)

Prefetch Miss: Insertion Policy?

MRU       Cache Set       LRU

Good (Inaccurate prefetch)
Bad (accurate prefetch)
Predicting Usefulness of Prefetch

Predict Usefulness of Prefetch

Accurate

Inaccurate

Prefetch Miss

MRU

LRU

Cache Set

Fraction of Useful Prefetches

Informed Caching Policies for Prefetched Blocks
Prefetching in GPUs

- Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das,

"Orchestrated Scheduling and Prefetching for GPGPUs"
Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. Slides (pptx) Slides (pdf)

Orchestrated Scheduling and Prefetching for GPGPUs

Adwait Jog†  Onur Kayiran†  Asit K. Mishra§  Mahmut T. Kandemir†
Onur Mutlu‡  Ravishankar Iyer§  Chita R. Das†

†The Pennsylvania State University University Park, PA 16802
‡Carnegie Mellon University Pittsburgh, PA 15213
§Intel Labs Hillsboro, OR 97124
{adwait, onur, kandemir, das}@cse.psu.edu onur@cmu.edu {asit.k.mishra, ravishankar.iyer}@intel.com