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


*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;
    }
    ...
    // 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;
}
```

(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 \( Start\_PC \) 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.

\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 \( Thread\_ID \). This instruction has effect only if it is executed by the main thread.

\textbf{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.

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;
    }
```
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 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  lda  s1, 252(gp)  # &heap_tail
  2  ldl  t2, 0(sl1)  # ifrom = heap_tail
  1  ldq  t5, -76(sl1)  # &heap[0]
  3  cmplt t2, 0, t4  # see note
  4  addl t2, 0x1, t6  # heap_tail +1
  1  s8addq t2, t5, t3  # &heap_tail[t3]
  4  stl  t6, 0(sl1)  # store heap_tail
  3  stq  s0, 0(t3)  # &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  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.

```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  cmptle $f1,$f17,$f31 # (heap[ito]->cost
  6  cmptle $f1,$f17,$f31 # < cost) PRED
  6  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:**

Load 1 Miss

<table>
<thead>
<tr>
<th>Compute</th>
<th>Stall</th>
<th>Compute</th>
<th>Stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss 1</td>
<td></td>
<td>Miss 2</td>
<td></td>
</tr>
</tbody>
</table>

**Runahead:**

Load 1 Miss  Load 2 Miss  Load 1 Hit  Load 2 Hit

<table>
<thead>
<tr>
<th>Compute</th>
<th>Runahead</th>
<th>Compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss 1</td>
<td></td>
<td>Miss 2</td>
</tr>
</tbody>
</table>

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
Instruction Processing in Runahead Mode

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.
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

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

Micro-operations Per Cycle

- S95
- FP00
- INT00
- WEB
- MM
- PROD
- SERV
- WS
- AVG

Percentages:
- 12%
- 35%
- 15%
- 22%
- 12%
- 22%
- 22%
- 16%
- 52%
- 13%
Runahead on In-order vs. Out-of-order

[Bar chart showing micro-operations per cycle for different benchmarks and configurations.]

- **S95**
  - In-order baseline: 15%
  - In-order + runahead: 10%
  - Out-of-order baseline: 73%
  - Out-of-order + runahead: 23%

- **FP00**
  - In-order baseline: 73%
  - In-order + runahead: 16%

- **INT00**
  - In-order baseline: 20%
  - In-order + runahead: 22%

- **WEB**
  - In-order baseline: 17%
  - In-order + runahead: 13%

- **MM**
  - In-order baseline: 14%
  - In-order + runahead: 12%

- **PROD**
  - In-order baseline: 28%
  - In-order + runahead: 15%
  - Out-of-order baseline: 50%
  - Out-of-order + runahead: 47%

- **SERV**
  - In-order baseline: 39%
  - In-order + runahead: 20%

- **WS**
  - In-order baseline: 39%

- **AVG**
More on Runahead Execution

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"
  One of the 15 computer architecture papers of 2003 selected as Top Picks by IEEE Micro.

Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com
More on Runahead Execution (Short)

Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.

![Graph showing performance improvement with increasing L2 cache size](image-url)
More on Runahead in Sun ROCK

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.

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

- bzip2
- crafty
- gap
- gcc
- gzip
- mcf
- parser
- perlbench
- twolf
- vortex
- vpr
- ammp
- applu
- apsi
- art
- equake
- facerec
- fma3d
- galgel
- lucas
- mesa
- mgrid
- sixtrack
- swim
- wupwise

% Increase in IPC

% Increase in Executed Instructions

- AVG: 22%
- AVG: 27%
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

AVG

235%

26.5%

6.2%
Overall Impact on IPC

The chart shows the overall impact on Instruction Per Cycle (IPC) for various benchmarks across two categories:

- **baseline runahead**
- **all techniques**

### Specific Observations

- **bzip2**: 0% increase
- **crafty**: 116% increase
- **eon**: 22.6% increase
- **gap**: 22.1% increase
- **gcc**: 0% increase
- **gzip**: 0% increase
- **mcf**: 0% increase
- **parser**: 0% increase
- **perlbmk**: 0% increase
- **twolf**: 0% increase
- **vortex**: 0% increase
- **vpr**: 0% increase
- **ammp**: 0% increase
- **applu**: 0% increase
- **apsi**: 0% increase
- **art**: 116% increase
- **equake**: 0% increase
- **facerec**: 0% increase
- **fma3d**: 0% increase
- **galgel**: 0% increase
- **lucas**: 0% increase
- **mesa**: 0% increase
- **mgrid**: 0% increase
- **sixtrack**: 0% increase
- **swim**: 0% increase
- **wupwise**: 0% increase

**Average Increase in IPC**

- **Baseline Runahead**: 22.6%
- **All Techniques**: 22.1%
More on Efficient Runahead Execution


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

Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance"

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]
The Problem: Dependent Cache Misses

Runahead: **Load 2 is dependent on Load 1**

Cannot Compute Its Address!

- **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!**

- Load 1 Miss
- Load 2 INV
- Load 1 Hit
- Load 2 Miss

**Value Predicted**

- Load 1 Miss
- Load 2 Miss
- Load 1 Hit
- Load 2 Hit

**Can Compute Its Address**

- Compute
- Runahead
- Saved Speculative Instructions
- Saved Cycles
More on AVD Prediction

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
  "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns"
  Proceedings of the 38th International Symposium on Microarchitecture (MICRO), pages 233-244, Barcelona, Spain, November 2005. Slides (ppt) Slides (pdf)
  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
Even More on Runahead Execution

- Lecture video from Fall 2017
  - 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
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 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


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


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)

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

VIVEK SESHADRI, SAMIHAH 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 {asit.k.mishra, ravishankar.iyer}@intel.com

§Intel Labs
Hillsboro, OR 97124
More on Runahead Execution
Readings on Runahead Execution

- **Required**

- **Optional**
Remember: Latency Tolerance

- An out-of-order execution processor tolerates latency of multi-cycle operations by executing independent instructions concurrently
  - It does so by buffering instructions in reservation stations and reorder buffer
  - Instruction window: Hardware resources needed to buffer all decoded but not yet retired/committed instructions

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
Stalls due to Long-Latency Instructions

- When a long-latency instruction is not complete, it blocks instruction retirement.
  - Because we need to maintain precise exceptions

- Incoming instructions fill the instruction window (reorder buffer, reservation stations).

- Once the window is full, processor cannot place new instructions into the window.
  - This is called a full-window stall.

- A full-window stall prevents the processor from making progress in the execution of the program.
Full-window Stall Example

8-entry instruction window:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD R1 ← mem[R5]</td>
<td></td>
</tr>
<tr>
<td>BEQ R1, R0, target</td>
<td>Independent of the L2 miss, executed out of program order, but cannot be retired.</td>
</tr>
<tr>
<td>ADD R2 ← R2, 8</td>
<td></td>
</tr>
<tr>
<td>LOAD R3 ← mem[R2]</td>
<td></td>
</tr>
<tr>
<td>MUL R4 ← R4, R3</td>
<td></td>
</tr>
<tr>
<td>ADD R4 ← R4, R5</td>
<td></td>
</tr>
<tr>
<td>STOR mem[R2] ← R4</td>
<td></td>
</tr>
<tr>
<td>ADD R2 ← R2, 64</td>
<td></td>
</tr>
<tr>
<td>LOAD R3 ← mem[R2]</td>
<td>Younger instructions cannot be executed because there is no space in the instruction window.</td>
</tr>
</tbody>
</table>

L2 Miss! Takes 100s of cycles.

The processor stalls until the L2 Miss is serviced.

- Long-latency cache misses are responsible for most full-window stalls.
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
The Memory Latency Problem

- Problem: Memory latency is long

- And, it is not very easy to reduce it...
  - We examined many methods for reducing DRAM latency
    - ...
  - See Lecture 10: Low-Latency Memory
    - https://www.youtube.com/watch?v=vQd1YgOH1Mw

- And, even if we reduce memory latency, it is still long
  - Remember the fundamental capacity-latency tradeoff
  - Contention for memory increases latencies
How Do We Tolerate Stalls Due to Memory?

- Two major approaches
  - Reduce/eliminate stalls
  - Tolerate the effect of a stall when it happens

- Four fundamental techniques to achieve these
  - Caching
  - Prefetching
  - Multithreading
  - Out-of-order execution

- Many techniques have been developed to make these four fundamental techniques more effective in tolerating memory latency
Memory Latency Tolerance Techniques

- **Caching** [initially by Bloom+, 1962 and later 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 today)
Runahead Execution
Small Windows: Full-window Stalls

8-entry instruction window:

- LOAD R1 ← mem[R5]
- BEQ R1, R0, target
- ADD R2 ← R2, 8
- LOAD R3 ← mem[R2]
- MUL R4 ← R4, R3
- ADD R4 ← R4, R5
- STOR mem[R2] ← R4
- ADD R2 ← R2, 64

L2 Miss! Takes 100s of cycles.

- Independent of the L2 miss, executed out of program order, but cannot be retired.
- Younger instructions cannot be executed because there is no space in the instruction window.

The processor stalls until the L2 Miss is serviced.

- Long-latency cache misses are responsible for most full-window stalls.
Impact of Long-Latency Cache Misses

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
Impact of Long-Latency Cache Misses

- Non-stall (compute) time
- Full-window stall time

Normalized Execution Time

128-entry window

2048-entry window

L2 Misses

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

- Out-of-order execution requires large instruction windows to tolerate today’s main memory latencies.

- As main memory latency increases, instruction window size should also increase to fully tolerate the memory latency.

- Building a large instruction window is a challenging task if we would like to achieve
  - Low power/energy consumption (tag matching logic, ld/st buffers)
  - Short cycle time (access, wakeup/select latencies)
  - Low design and verification complexity
Efficient Scaling of Instruction Window Size

- One of the major research issues in out of order execution
- How to achieve the benefits of a large window with a small one (or in a simpler way)?
- How do we efficiently tolerate memory latency with the machinery of out-of-order execution (and a small instruction window)?
Memory Level Parallelism (MLP)

- Idea: Find and service multiple cache misses in parallel so that the processor stalls only once for all misses

- Enables latency tolerance: overlaps latency of different misses

- How to generate multiple misses?
  - Out-of-order execution, multithreading, prefetching, runahead
Runahead Execution (I)

- A technique to obtain the memory-level parallelism benefits of a large instruction window

- 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

Perfect Caches:

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.**
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.
- **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**

**Micro-operations Per Cycle**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>S95</th>
<th>FP00</th>
<th>INT00</th>
<th>WEB</th>
<th>MM</th>
<th>PROD</th>
<th>SERV</th>
<th>WS</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetcher, no runahead</td>
<td>12%</td>
<td>35%</td>
<td>13%</td>
<td>22%</td>
<td>12%</td>
<td>22%</td>
<td>16%</td>
<td>52%</td>
<td>22%</td>
</tr>
<tr>
<td>Only prefetcher (baseline)</td>
<td>15%</td>
<td>35%</td>
<td>13%</td>
<td>22%</td>
<td>12%</td>
<td>22%</td>
<td>16%</td>
<td>52%</td>
<td>22%</td>
</tr>
<tr>
<td>Only runahead</td>
<td>22%</td>
<td>12%</td>
<td>15%</td>
<td>22%</td>
<td>12%</td>
<td>22%</td>
<td>16%</td>
<td>52%</td>
<td>22%</td>
</tr>
<tr>
<td>Prefetcher + runahead</td>
<td>22%</td>
<td>12%</td>
<td>15%</td>
<td>22%</td>
<td>12%</td>
<td>22%</td>
<td>16%</td>
<td>52%</td>
<td>22%</td>
</tr>
</tbody>
</table>
Runahead Execution vs. Large Windows

- 128-entry window (baseline)
- 128-entry window with Runahead
- 256-entry window
- 384-entry window
- 512-entry window

Micro-operations Per Cycle

<table>
<thead>
<tr>
<th>Category</th>
<th>128-entry window (baseline)</th>
<th>128-entry window with Runahead</th>
<th>256-entry window</th>
<th>384-entry window</th>
<th>512-entry window</th>
</tr>
</thead>
<tbody>
<tr>
<td>S95</td>
<td>1.15</td>
<td>1.10</td>
<td>1.05</td>
<td>1.00</td>
<td>0.95</td>
</tr>
<tr>
<td>FP00</td>
<td>1.10</td>
<td>1.05</td>
<td>1.00</td>
<td>0.95</td>
<td>0.90</td>
</tr>
<tr>
<td>INT00</td>
<td>1.05</td>
<td>1.00</td>
<td>0.95</td>
<td>0.90</td>
<td>0.85</td>
</tr>
<tr>
<td>WEB</td>
<td>0.95</td>
<td>0.90</td>
<td>0.85</td>
<td>0.80</td>
<td>0.75</td>
</tr>
<tr>
<td>MM</td>
<td>0.90</td>
<td>0.85</td>
<td>0.80</td>
<td>0.75</td>
<td>0.70</td>
</tr>
<tr>
<td>PROD</td>
<td>0.85</td>
<td>0.80</td>
<td>0.75</td>
<td>0.70</td>
<td>0.65</td>
</tr>
<tr>
<td>SERV</td>
<td>0.80</td>
<td>0.75</td>
<td>0.70</td>
<td>0.65</td>
<td>0.60</td>
</tr>
<tr>
<td>WS</td>
<td>0.75</td>
<td>0.70</td>
<td>0.65</td>
<td>0.60</td>
<td>0.55</td>
</tr>
<tr>
<td>AVG</td>
<td>0.70</td>
<td>0.65</td>
<td>0.60</td>
<td>0.55</td>
<td>0.50</td>
</tr>
</tbody>
</table>
Runahead vs. A (Real) Large Window

- When is one beneficial, when is the other?
- Pros and cons of each
- Which can tolerate floating-point operation latencies better?
- Which leads to less wasted execution?
Runahead on In-order vs. Out-of-order

Micro-operations Per Cycle

- in-order baseline
- in-order + runahead
- out-of-order baseline
- out-of-order + runahead

S95: 15% 10%
FP00: 73% 23%
INT00: 73% 16%
WEB: 17% 13%
MM: 20% 22%
PROD: 14% 12%
SERV: 28% 15%
WS: 50% 47%
AVG: 39% 20%
Effect of Runahead in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.

![Graph showing the effect of L2 cache size on normalized IPC with and without Scout feature.](image_url)
Generalizing the Idea

- Runahead on different long-latency operations?
More on Runahead Execution

Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"
One of the 15 computer architecture papers of 2003 selected as Top Picks by IEEE Micro.

Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com
More on Runahead in Sun ROCK

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.

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 in IBM POWER6

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
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]
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

- **bzip2**: 235%
- **crafty**: 6.2%
- **eon**: 6.2%
- **gap**:
- **gcc**:
- **gzip**:
- **mcf**:
- **parser**:
- **perlbmk**:
- **twolf**:
- **vortex**:
- **vr**:
- **ammp**:
- **applu**:
- **apsi**:
- **art**: 26.5%
- **equake**:
- **facerec**:
- **fma3d**:
- **galgel**:
- **lucas**:
- **mesa**:
- **mgid**:
- **sixtrack**:
- **swim**:
- **wupwise**:
- **AVG**:

Overall Impact on Executed Instructions:
- Baseline runahead: 6.2%
- All techniques: 26.5%
Overall Impact on IPC

Increase in IPC

baseline runahead
all techniques

22.6%
22.1%

AVG
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**
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?
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 Problem: Dependent Cache Misses

- 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

**Value Predicted**

Load 1 Miss  Load 2 Miss  Load 1 Hit  Load 2 Hit

Compute  Runahead  Miss 1  Miss 2

**Cannot Compute Its Address!**

Load 1 Miss  Load 2 INV  Load 1 Hit  Load 2 Miss

Compute  Runahead  Miss 1  Miss 2

**Can Compute Its Address**

Saved Speculative Instructions

Saved Cycles
AVD Prediction [MICRO’05]

- Address-value delta (AVD) of a load instruction defined as:
  \[ \text{AVD} = \text{Effective Address of Load} - \text{Data Value of Load} \]

- For some address loads, AVD is stable
- An AVD predictor keeps track of the AVDs of address loads
- When a load is an L2 miss in runahead mode, AVD predictor is consulted

- If the predictor returns a stable (confident) AVD for that load, the value of the load is predicted
  \[ \text{Predicted Value} = \text{Effective Address} - \text{Predicted AVD} \]
Why Do Stable AVDs Occur?

- Regularity in the way data structures are allocated in memory AND traversed

- Two types of loads can have stable AVDs
  - Traversal address loads
    - Produce addresses consumed by address loads
  - Leaf address loads
    - Produce addresses consumed by data loads
Traversing Address Loads

Regularly-allocated linked list:

A traversal address load loads the pointer to the next node:

\[ \text{node} = \text{node} \rightarrow \text{next} \]

AVD = Effective Addr – Data Value

<table>
<thead>
<tr>
<th>Effective Addr</th>
<th>Data Value</th>
<th>AVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A+k</td>
<td>-k</td>
</tr>
<tr>
<td>A+k</td>
<td>A+2k</td>
<td>-k</td>
</tr>
<tr>
<td>A+2k</td>
<td>A+3k</td>
<td>-k</td>
</tr>
</tbody>
</table>

Striding data value Stable AVD
Leaf Address Loads

Sorted dictionary in **parser**: Nodes point to strings (words)
String and node allocated consecutively

Dictionary looked up for an input word.
A **leaf address load** loads the pointer to the string of each node:

```
lookup (node, input) {
    ptr_str = node->string;
    m = check_match(ptr_str, input);
    // ...
}
```

**AVD = Effective Addr – Data Value**

<table>
<thead>
<tr>
<th>Effective Addr</th>
<th>Data Value</th>
<th>AVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+k</td>
<td>A</td>
<td>k</td>
</tr>
<tr>
<td>C+k</td>
<td>C</td>
<td>k</td>
</tr>
<tr>
<td>F+k</td>
<td>F</td>
<td>k</td>
</tr>
</tbody>
</table>

No stride! Stable AVD
Identifying Address Loads in Hardware

- Insight:
  - If the AVD is too large, the value that is loaded is likely not an address

- Only keep track of loads that satisfy:
  - \(-\text{MaxAVD} \leq \text{AVD} \leq +\text{MaxAVD}\)

- This identification mechanism eliminates many loads from consideration for prediction
  - No need to value- predict the loads that will not generate addresses
  - Enables the predictor to be small
An Implementable AVD Predictor

- Set-associative prediction table
- Prediction table entry consists of
  - Tag (Program Counter of the load)
  - Last AVD seen for the load
  - Confidence counter for the recorded AVD
- Updated when an address load is retired in normal mode
- Accessed when a load misses in L2 cache in runahead mode
- **Recovery-free**: No need to recover the state of the processor or the predictor on misprediction
  - Runahead mode is purely speculative
AVD Update Logic

Effective Address  Data Value

computed AVD = Effective Addr - Data Value

\[ \text{MaxAVD?} \]

\[ \text{MaxAVD?} \]

valid AVD?

Tag  Conf  AVD

PC of Retired Load
AVD Prediction Logic

Predicted? (not INV?)

Predicted Value
= Effective Addr − AVD

Program Counter of L2-miss Load

Effective Address of L2-miss Load

Tag

Conf

AVD
Performance of AVD Prediction

Normalized Execution Time and Executed Instructions

- **Execution Time**
- **Executed Instructions**

Normalized Execution Time and Executed Instructions for bisort, health, mst, perimeter, treeadd, tsp, voronoi, mcf, parser, twolf, vpr, and AVG.

- **runahead**
  - 14.3%
  - 15.5%
More on AVD Prediction


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
Wrong Path Events
An Observation and A Question

• In an out-of-order processor, some instructions are executed on the mispredicted path (wrong-path instructions).

• Is the behavior of wrong-path instructions different from the behavior of correct-path instructions?
  – If so, we can use the difference in behavior for early misprediction detection and recovery.
What is a Wrong Path Event?

An instance of illegal or unusual behavior that is more likely to occur on the wrong path than on the correct path.

Wrong Path Event = WPE

Probability (wrong path | WPE) ~ 1
Why Does a WPE Occur?

• A wrong-path instruction may be executed before the mispredicted branch is executed.
  – Because the mispredicted branch may be dependent on a long-latency instruction.

• The wrong-path instruction may consume a data value that is not properly initialized.
WPE Example from eon:
NULL pointer dereference

```c
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
```

for (int i=0; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
First iteration

```c
for (int i=0; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```
```c
1:  for (int i=0 ; i< length(); i++) {
2:    structure *ptr = array[i];
3:    if (ptr->x) {
4:      // . . .
5:    }
6:  }
```
Loop branch correctly predicted

Array boundary

Array of pointers to structs

```
1:   for (int i=0; i< length(); i++) {
2:       structure *ptr = array[i];
3:       if (ptr->x) {
4:           // . . .
5:       }
6:   }
```
Second iteration

```c
for (int i=0 ; i< length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // . . .
    }
}
```
Second iteration

Array boundary

i = 1
ptr = xEFF8B0

Array of pointers to structs

x8ABCD0  xEFF8B0  x0  x0

1: for (int i=0; i< length(); i++) {
2: structure *ptr = array[i];
3: if (ptr->x) {
4:     // . . .
5: }
6: }

*ptr
Loop exit branch mispredicted

Array boundary

Array of pointers to structs

x8ABCD0  xEFF8B0  x0  x0

1 : for (int i=0 ; i< length(); i++) {
2 :     structure *ptr = array[i];
3 :     if (ptr->x) {
4 :         // . . .
5 :     }
6 : }
Third iteration on wrong path

Array boundary

i = 2
ptr = 0

1: for (int i=0; i< length(); i++) {
2:     structure *ptr = array[i];
3:     if (ptr->x) {
4:          // . . .
5:     }
6: }

Array of pointers to structs

x8ABCD0  xEFF8B0  x0  x0
Wrong Path Event

Array of pointers to structs

Array boundary

i = 2
ptr = 0

NULL pointer dereference!

```
for (int i = 0; i < length(); i++) {
    structure *ptr = array[i];
    if (ptr->x) {
        // ...
    }
}
```
Types of WPEs

• Due to memory instructions
  – NULL pointer dereference
  – Write to read-only page
  – Unaligned access (illegal in the Alpha ISA)
  – Access to an address out of segment range
  – Data access to code segment
  – Multiple concurrent TLB misses
Types of WPEs (continued)

- Due to control-flow instructions
  - Misprediction under misprediction
    - If three branches are executed and resolved as mispredicts while there are older unresolved branches in the processor, it is almost certain that one of the older unresolved branches is mispredicted.
  - Return address stack underflow
  - Unaligned instruction fetch address (illegal in Alpha)

- Due to arithmetic instructions
  - Some arithmetic exceptions
    - e.g. Divide by zero
Two Empirical Questions

1. How often do WPEs occur?

2. When do WPEs occur on the wrong path?
More on Wrong Path Events

David N. Armstrong, Hyesoon Kim, Onur Mutlu, and Yale N. Patt,
"Wrong Path Events: Exploiting Unusual and Illegal Program
Behavior for Early Misprediction Detection and Recovery"
Proceedings of the 37th International Symposium on
Microarchitecture (MICRO), pages 119-128, Portland, OR, December
2004. Slides (pdf) Slides (ppt)

Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early
Misprediction Detection and Recovery

David N. Armstrong  Hyesoon Kim  Onur Mutlu  Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{dna,hyesoon,onur,patt}@ece.utexas.edu
Why Is This Important?

- A modern processor spends significant amount of time fetching/executing instructions on the wrong path.

**Fig. 1.** Percentage of fetch cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the baseline processor for SPEC 2000 integer benchmarks.
A Lot of Time Spent on The Wrong Path

- A runahead processor, much more so...

Fig. 20. Percentage of total cycles spent on the wrong path, percentage of instructions fetched on the wrong path, and percentage of instructions (memory and nonmemory) executed on the wrong path in the runahead processor.
4 Wrong Path: To Model or Not to Model

In this section, we measure the error in IPC if wrong-path memory references are not simulated. We also evaluate the overall effect of wrong-path memory references on the IPC (retired Instructions Per Cycle) performance of a processor.

1. How important is it to correctly model wrong-path memory references? What is the error in the predicted performance if wrong-path references are not modeled?

2. Do wrong-path memory references affect performance positively or negatively? What is the relative significance on performance of prefetching, bandwidth consumption, and pollution caused by wrong-path references?

3. What kind of code structures lead to the positive effects of wrong-path memory references?

4. How do wrong-path memory references affect the performance of a runahead execution processor [7], [18] which implements an aggressive form of speculative execution?
Wrong Path Is Often Useful for Performance

Fig. 7. Error in the IPC of the baseline processor with a stream prefetcher for three different memory latencies if wrong-path memory references are not simulated.
More So In Runahead Execution

![Graph showing IPC improvement due to runahead execution](image)

**Fig. 19.** IPC improvement of adding runahead execution to the baseline processor if wrong-path memory references are or are not modeled.
Why is Wrong Path Useful? (I)

- Control-independence: e.g., wrong-path execution of future loop iterations

```c
1:    arc_t *arc; // array of arc_t structures
2:    // initialize arc (arc = ...)
3:    
4:    for ( ; arc < stop_arcs; arc += size) {
5:        if (arc->ident > 0) { // frequently mispredicted br.
6:            // function calls and
7:            // operations on the structure pointed to by arc
8:            // ...
9:        }
10:    }
```

Fig. 16. An example from mcf showing wrong-path prefetching for later loop iterations.
Why is Wrong Path Useful? (II)

1:  l = min; r = max;
2:  cut = perm[ (long)( l+r ) / 2 ] -> abs_cost;
3:  
4:  do {
5:      while( perm[l] -> abs_cost > cut )
6:          l++;
7:      while( cut > perm[r] -> abs_cost )
8:          r--;
9:  
10:     if( l < r ) {
11:         xchange = perm[l];
12:         perm[l] = perm[r];
13:         perm[r] = xchange;
14:     }
15:     if( l <= r ) {
16:         l++; r--;
17:     }
18: } while( l <= r );

Fig. 17. An example from mcf showing wrong-path prefetching between different loops.
Why is Wrong Path Useful? (III)

- Same data used in different control flow paths

```c
1:    node_t *node;
2:    // initialize node
3:    // ...
4:    
5:    while (node) {
6:        
7:    if (node->orientation == UP) { // mispredicted branch
8:        node->potential = node->basic_arc->cost
9:            + node->pred->potential;
10:    } else { /* == DOWN */
11:        node->potential = node->pred->potential
12:            - node->basic_arc->cost;
13:        // ...
14:    }
15:    // control-flow independent point (re-convergent point)
16:    node = node->child;
17:    }
```

Fig. 18. An example from mcf showing wrong-path prefetching in control-flow hammocks.
More on Wrong Path Execution (I)

- Onur Mutlu, Hyesoon Kim, David N. Armstrong, and Yale N. Patt, "Understanding the Effects of Wrong-Path Memory References on Processor Performance". Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI), pages 56-64, Munchen, Germany, June 2004. Slides (pdf)

Understanding The Effects of Wrong-Path Memory References on Processor Performance

Onur Mutlu  Hyesoon Kim  David N. Armstrong  Yale N. Patt

Department of Electrical and Computer Engineering
The University of Texas at Austin
{onur,hyesoon,dna,patt}@ece.utexas.edu
More on Wrong Path Execution (II)


An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors

Onur Mutlu, *Student Member, IEEE*, Hyesoon Kim, *Student Member, IEEE*, David N. Armstrong, and Yale N. Patt, *Fellow, IEEE*
What If …

- The system learned from wrong-path execution and used that learning for better execution of the program/system?

- An open research problem…
More on Runahead Enhancements
Eliminating Short Periods

Mechanism to eliminate short periods:
- Record the number of cycles $C$ an L2-miss has been in flight
- If $C$ is greater than a threshold $T$ for an L2 miss, disable entry into runahead mode due to that miss
- $T$ can be determined statically (at design time) or dynamically

- $T=400$ for a minimum main memory latency of 500 cycles works well
Eliminating Overlapping Periods

- Overlapping periods are not necessarily useless
  - The availability of a new data value can result in the generation of useful L2 misses
- But, this does not happen often enough

- Mechanism to eliminate overlapping periods:
  - Keep track of the number of pseudo-retired instructions $R$ during a runahead period
  - Keep track of the number of fetched instructions $N$ since the exit from last runahead period
  - If $N < R$, do not enter runahead mode
Properties of Traversal-based AVDs

- Stable AVDs can be captured with a **stride value predictor**
- Stable AVDs disappear with the **re-organization of the data structure** (e.g., sorting)

Stability of AVDs is dependent on the behavior of the memory allocator
- Allocation of contiguous, fixed-size chunks is useful
Properties of Leaf-based AVDs

- Stable AVDs **cannot** be captured with a stride value predictor
- Stable AVDs **do not** disappear with the re-organization of the data structure (e.g., sorting)
- Stability of AVDs is dependent on the behavior of the memory allocator

Distance between node and string still constant! ✓
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
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

![Graph showing performance normalized to no throttling for different workloads, with a significant improvement indicated by 56%].

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's goal: Maximize the prefetching performance of core \( i \) independently.

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

- Local Control
  - Pref. \( i \)
  - Throttling Decision
  - Local Throttling Decision
  - Core \( i \)
  - Accuracy

- Global Control
  - Final Throttling Decision
  - Local Throttling Decision
  - Bandwidth Feedback
  - Cache Pollution Feedback
  - Shared Cache

Local control's goal: Accepts or overrides decisions made by local control to improve overall system performance.
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

Pol. Filter i

Shared Cache

Local Throttling Decision

Enforce Throttle Down

High Pol (i)
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></td>
<td>Others’ low 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></td>
<td>Others’ high BW need</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Pol (i): Policy
- Acc (i): Accuracy
- BW (i): Bandwidth
- BWNO (i): Bandwidth Needed Others
- Action: Throttle down
HPAC Evaluation

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

Normalized to system with no prefetching

- 9% improvement in normalized system performance
- 15% decrease in normalized system unfairness
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

Verifier needs Y, X, and Z

DRAM Controller
Row Buffer

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

Demand-first

2 row-conflicts, 1 row-hit

Stall
Execution
When Prefetches are Useful

<table>
<thead>
<tr>
<th>Demand-first</th>
<th>Demand-pref-equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 row-conflicts, 1 row-hit</td>
<td>2 row-hits, 1 row-conflict</td>
</tr>
</tbody>
</table>

**Demand-pref-equal outperforms demand-first**

- 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

Graph showing IPC normalized to no prefetching for different benchmarks.
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)


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

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

†HPS Research Group  Chang Joo Lee chang.joo.lee@intel.com
The University of Texas at Austin  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
Problem: Existing caching policies for prefetched blocks result in significant cache pollution.

Are these insertion and promotion policies good for prefetched blocks?
Prefetch Usage Experiment

Monitor L2 misses

Prefetch into L3

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

Promote to MRU

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

Prefetch Miss: Insertion Policy?

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

Fraction of Useful Prefetches

Prefetch Miss

Predict Usefulness of Prefetch

Accurate

Inaccurate

Cache Set

MRU

LRU
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