Software Prefetching for Indirect Memory Accesses

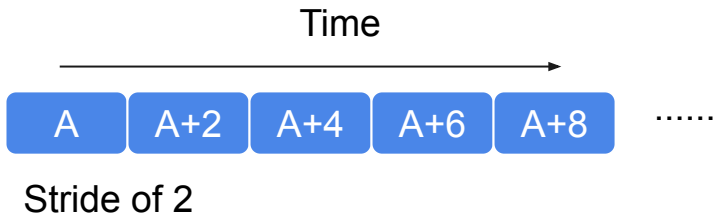
Authors: Sam Ainsworth, Timothy M. Jones

Presented By: Group 3
Arham Jain, Hyung Rae Cho, Michael Alvin, Zihao Deng

Background + Motivation

Many data processing and HPC workloads are memory-latency bound.

- Solution? Prefetching
- Hardware stride prefetcher



What if the data access pattern is not regular? e.g. indirect memory access

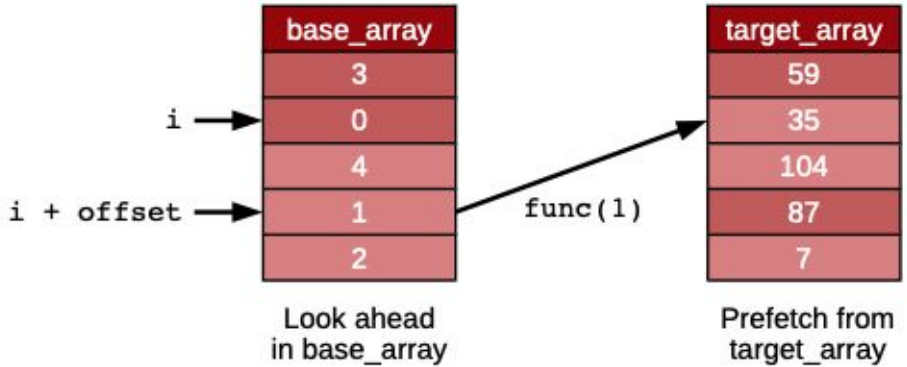
- Rather than A[i], we want to access A[B[i]]
- Hardware prefetchers fail - they fetch B[i + 1] quickly but not data in A

Intro to Software Prefetching

Analyze the code and insert prefetching instructions at compile time.

```
1 for (i=0; i<base_array_size; i++) {
2   target_array[func(base_array[i])]++;
3 }
```

(a) Code containing stride-indirect accesses



(b) Prefetching from target_array

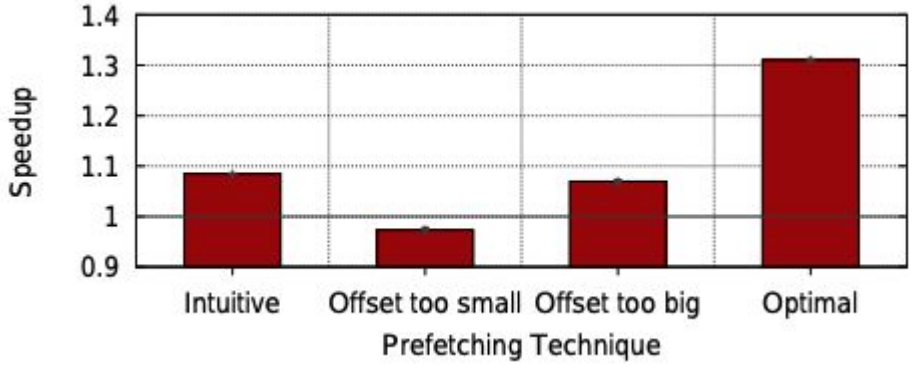
- Many workloads (e.g. graphs) perform stride-indirect traversals starting from an array
- We can look ahead in the base array and prefetch future values from the target array

Challenges in Software Prefetching

```
for (i = 0; i < NUM_KEYS; i++) {
   key_buff1[ key_buff2[ i ] ]++;
}
```

```
for (i = 0; i < NUM_KEYS; i++) {
   SWPF(key_buff1[ key_buff2[ i + offset ] ]); // intuitive
   SWPF(key_buff2[ i + offset * 2 ]); // required for optimal performance
   key_buff1[ key_buff2[ i ] ]++;
}
```

- Inserting prefetch instructions manually for maximal performance is challenging even in simple cases.
- Choosing a good prefetch distance is critical to avoid fetching the data too late (when the offset is too small), or polluting the cache (when the offset is too large).



Software prefetching performance for code above on an Intel Haswell micro-architecture

Prefetch Generation Overview

1. Backwards depth-first search to find induction variables and instructions that reference the induction variables
2. Perform safety analysis
    a. Remove instructions with (side effect) function calls and non-induction-variable phi nodes
    b. Perform address bounds check to limit the range of induction variables to known valid values
    c. Disallow unsafe prefetches
3. Calculate offsets and generate prefetch instructions

1. Find induction variables and associated instructions

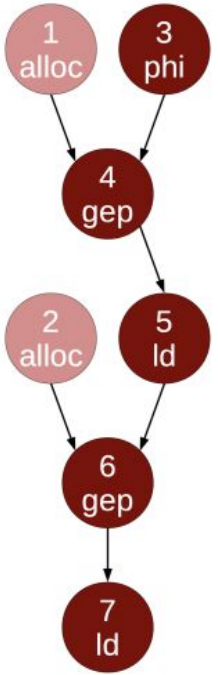
- For each load in a loop, DFS is called.
- Find induction variable (IV) while walking through the data dependence graph (Use-Def) using DFS. (2~10)
- When multiple IVs exist, only the IV in closest (innermost) loop to load is picked (21).

```
1  DFS(inst) {
2    candidates = {}
3    foreach (o: inst.src_operands):
4      // Found induction variable, finished this path.
5      if (o is an induction variable):
6        candidates ∪= {(o, {inst})}
7      // Recurse to find an induction variable.
8      elif (o is a variable and is defined in a loop):
9        if (((iv, set) = DFS(loop_def(o))) != null):
10         candidates ∪= {(iv, {inst}∪set)}
11
12   // Simple cases of 0 or 1 induction variable.
13   if (candidates.size == 0):
14     return null
15   elif (candidates.size == 1):
16     return candidates[0]
17
18   // There are paths based on multiple induction
19   // variables, so choose the induction variable in
20   // the closest loop to the load.
21   indvar = closest_loop_indvar(candidates)
22
23   // Merge paths which depend on indvar.
24   return merge_instructions(indvar, candidates)
25 }
```

1. Find induction variables and associated instructions (ex)

```
for(i = 0; i < size; i++) {
        b[a[i]]++
        prefetch(b[a[i + ?]])
        prefetch(a[i + ?])
}
```

```
1 start: alloc a, asize
2        alloc b, bsize
3 loop:  phi i, [#0, i.1]
4        gep t1, a, i
5        ld t2, t1
6        gep t3, b, t2
7        ld t4, t3
8        add t5, t4, #1
9        str t3, t5
10       add i.1, i, #1
11       cmp size, i.1
12       bne loop
```



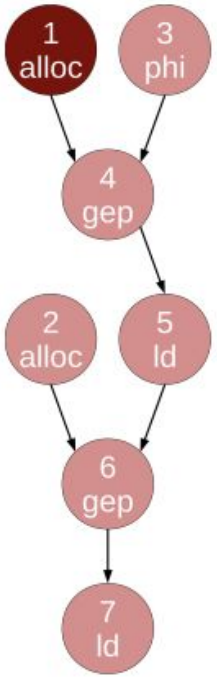
2. Perform safety analysis

- Prefetches can't cause faults, but loads to calculate the address of the prefetch can
- **Need to limit the range of the prefetching induction variable (can't prefetch past the max size of the induction variable)**
- Give up if stores happen to data structure that is used to generate the loads
    - Eg, x[y[z[i]]] - if we store to z, we can't safely prefetch on x as x[y[z[i + 1]]] might not make sense anymore

2. Perform safety analysis (ex)

```
for(i = 0; i < size; i++) {
        b[a[i]]++
        if(i + ? < asize) {
                prefetch(b[a[i + ?]])
                prefetch(a[i + ?])
        }
}
```

```
1 start: alloc a, asize
2        alloc b, bsize
3 loop:  phi i, [#0, i.1]
4        gep t1, a, i
5        ld t2, t1
6        gep t3, b, t2
7        ld t4, t3
8        add t5, t4, #1
9        str t3, t5
10       add i.1, i, #1
11       cmp size, i.1
12       bne loop
```



3. Calculate offset and generate prefetch instructions

- Prefetching too early risks cache pollution and data being evicted before use
- Prefetching too late risks the data not being fetched early enough to mask the cache miss
- Therefore, calculate offset to schedule prefetch

$$\text{offset} = \frac{c\,(t - l)}{t}$$

c = Micro-architecture specific constant

t = Number of loads in total sequence

l = Position of a given load in its sequence

3. Calculate offset and generate prefetch instructions (ex)

```
for(i = 0; i < size; i++) {
        b[a[i]]++
        if(i + 32 < asize) {
                prefetch(b[a[i + 32]]) // l = 1
        }
        if(i + 64 < asize) {
                prefetch(a[i + 64]) // l = 0
        }
}
```

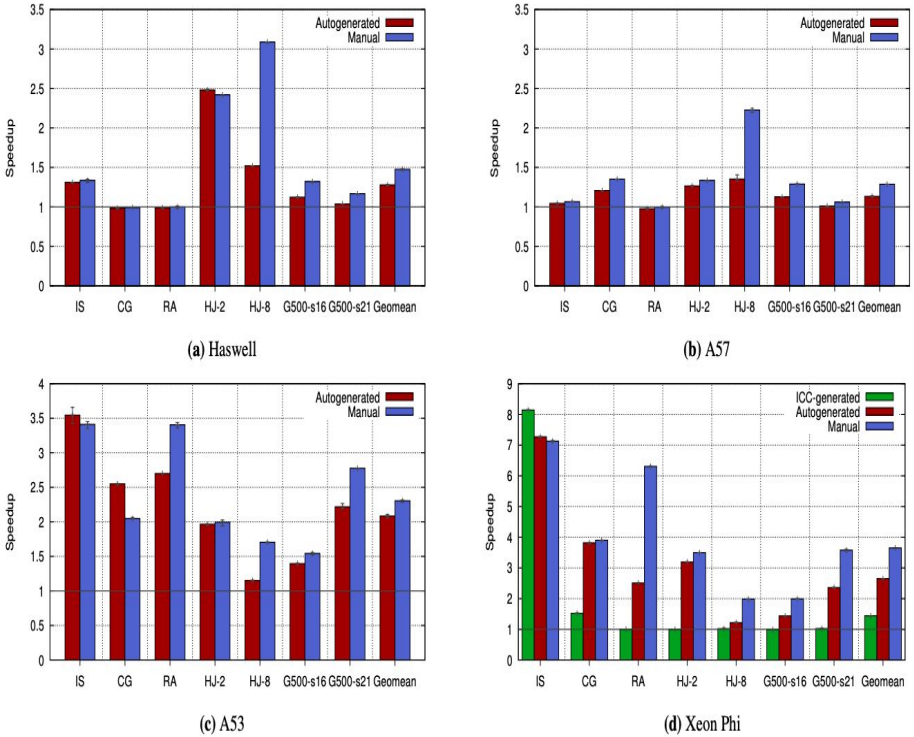
$$\text{offset} = \frac{c\,(t - l)}{t}$$

c = Micro-architecture specific constant = 64

t = Number of loads in total sequence = 2

l = Position of a given load in its sequence

Result - Performance gain



(a) Haswell (b) A57

(c) A53 (d) Xeon Phi

- Based on the architectural features of the system, the pass shows different speedup.
- Shows near optimal performance for Hash Join (database access), Conjugate gradient and Integer Sort regardless of system.
- Pass accomplishes speed up in most of benchmarks.

Result-Look Ahead Distance

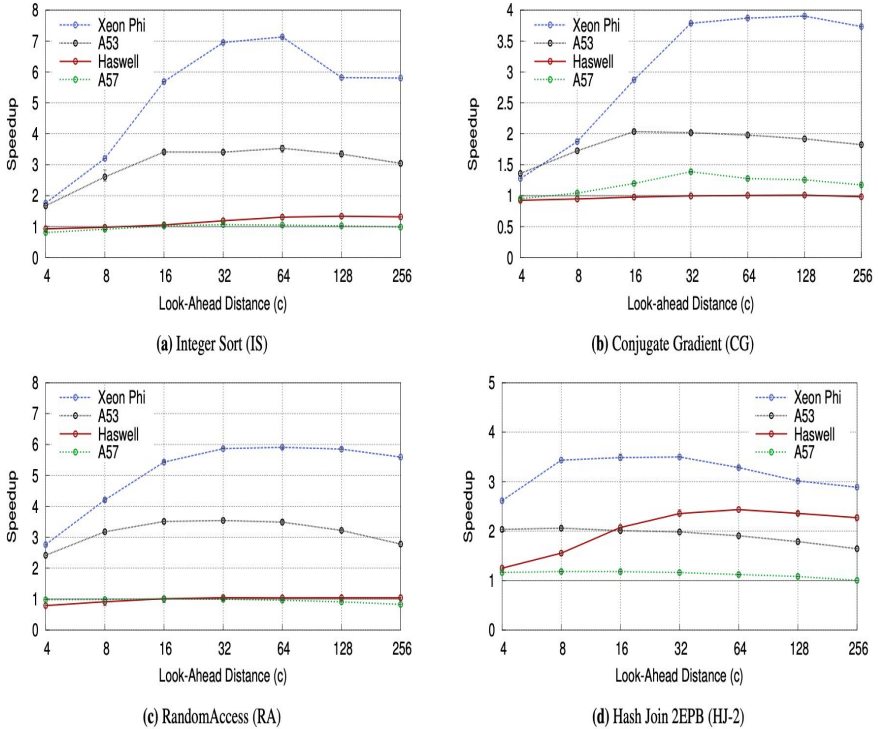


Figure 6: Varying look-ahead distance shows the best is usually the consistent across systems.

- Short distance results in cache miss and long distance would result in cache pollution.
- Optimal when distance is 64, but larger distances can also show near optimal performance.
- Cache miss is more detrimental than cache pollution.

Strengths

- Safe optimization that avoids faults (as long as the original code is fine)
- Generally provides a speedup to memory bound code
- Real world benefits - hashmaps, graphs, databases, game engines
- Optimal lookahead is fairly consistent among architectures

Limitations

- Very specific set of conditions to do it safely
  - Prefetched loads need bounds checking
  - Even then, faults can occur earlier due to the prefetch address calculation
- Ignores additional performance in exchange for simplicity
- Works well on benchmarks, real world code is not always that straightforward
- Software prefetching is slower and uses more energy than hardware prefetching

Experiment Setup

Implement
Manual(Almost Optimal)
Pass Generated

Benchmarks
Integer Sort(IS)
Conjugate Gradient(CG)
Random Access(RA)
Hash Join(HJ-2,HJ-8)
Graph 500

System
Haswell(IS)
Xeon Phi(CG)
A57
A53