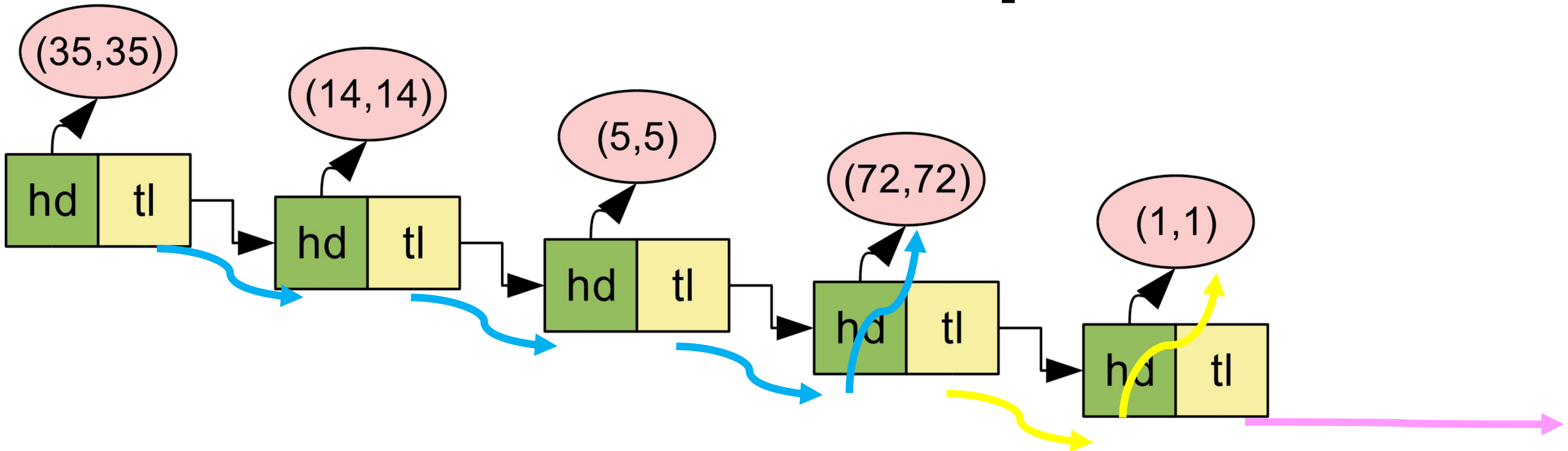
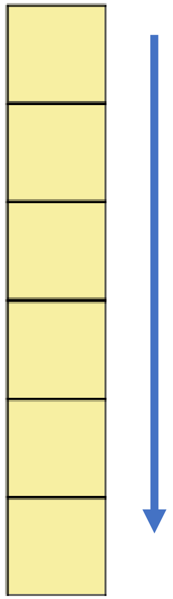
Prefetching in Functional Languages: A Hardware-Software Retrospective



Sam Ainsworth

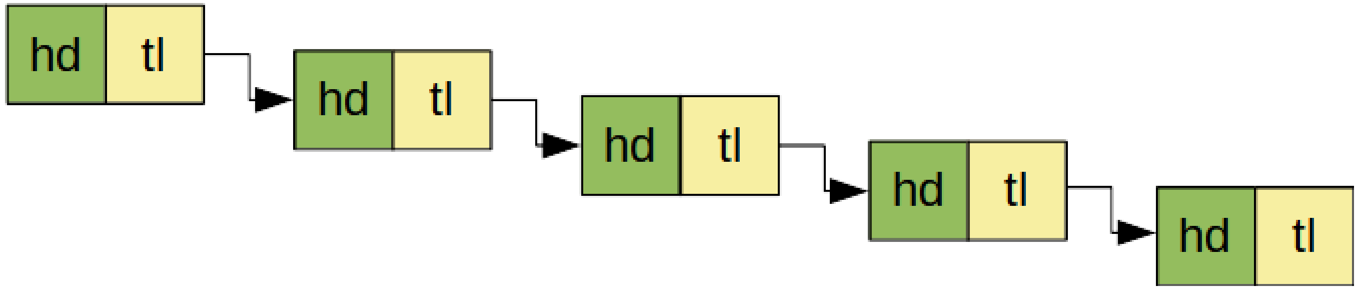
(Based in-part on a paper with Timothy M. Jones at ISMM 2020)

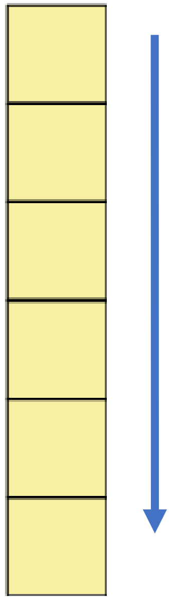
Motivation: Everybody knows functional data structures are scary



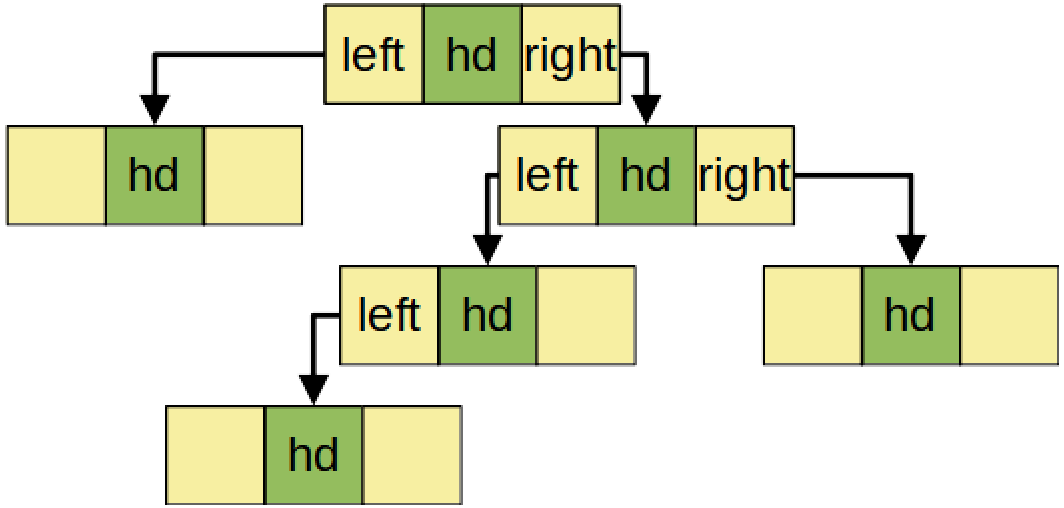
Those nice prefetchable arrays are, sadly, mutable

Motivation: Everybody knows functional data structures are scary



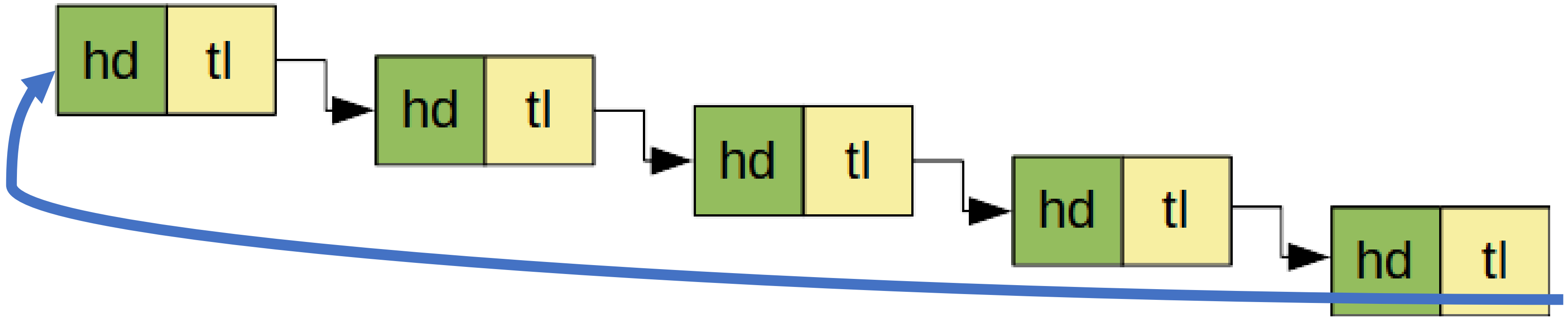


Where’s the Memory-Level Parallelism for linked lists or trees?



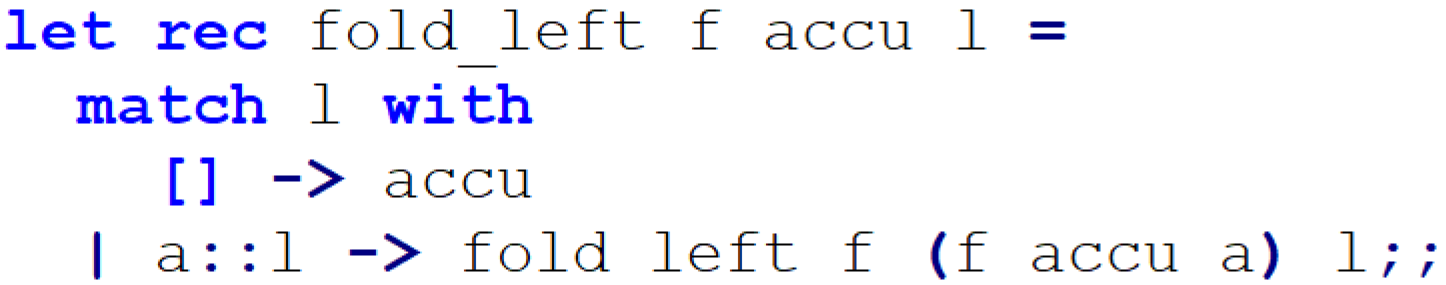
Those nice prefetchable arrays are, sadly, mutable

One answer: temporal prefetching

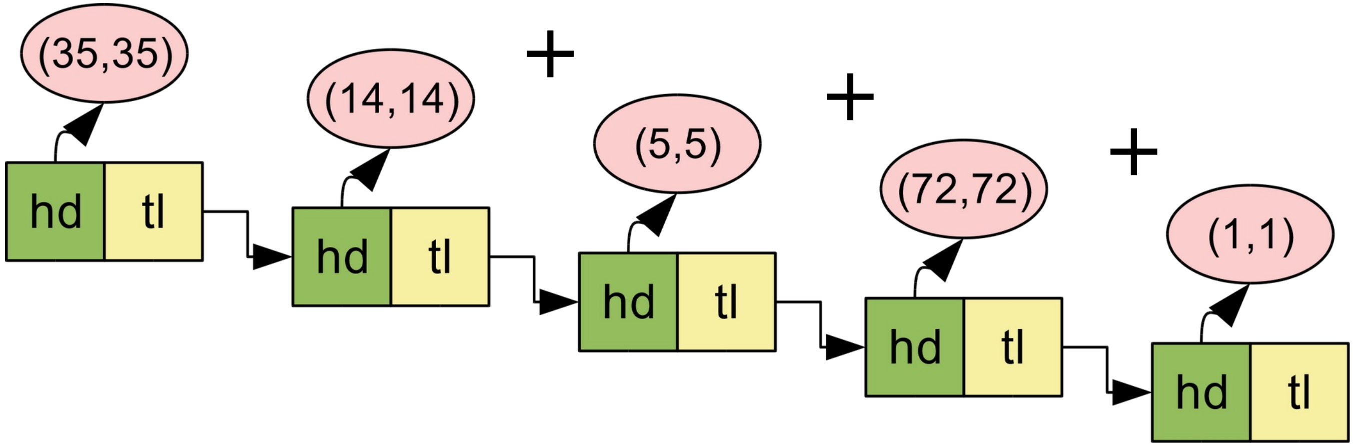


- If your pattern repeats, store and replay it for prefetching!
- Recently become viable in hardware: see the Arm Cache Miss Chaining (CMC) Prefetcher https://hc33.hotchips.org/assets/program/conference/day1/20210818_Hotchips_NeoverseN2.pdf

…but actually, linked lists aren’t as bad as you’d expect

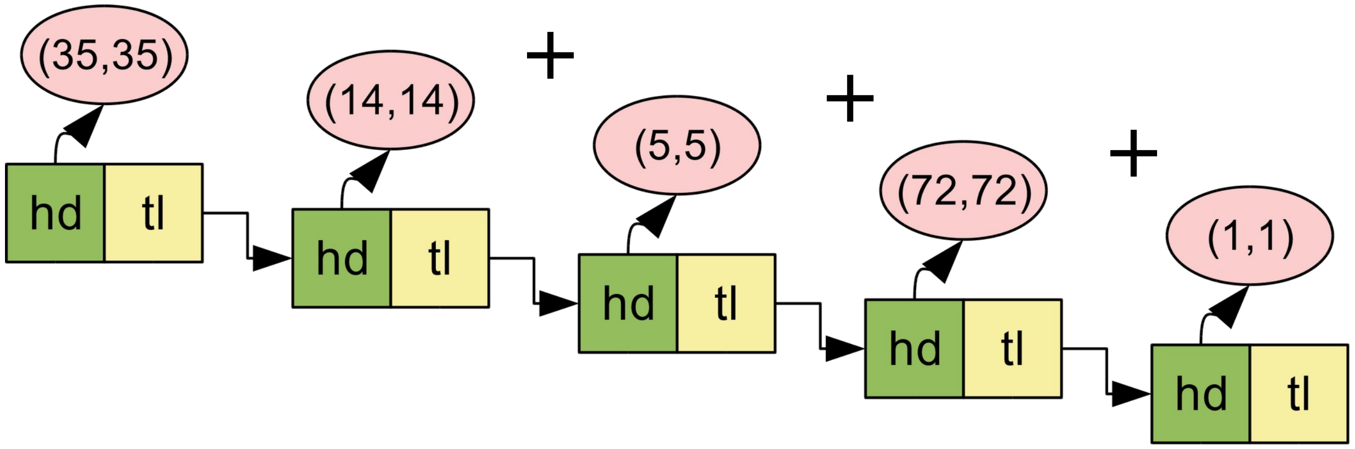


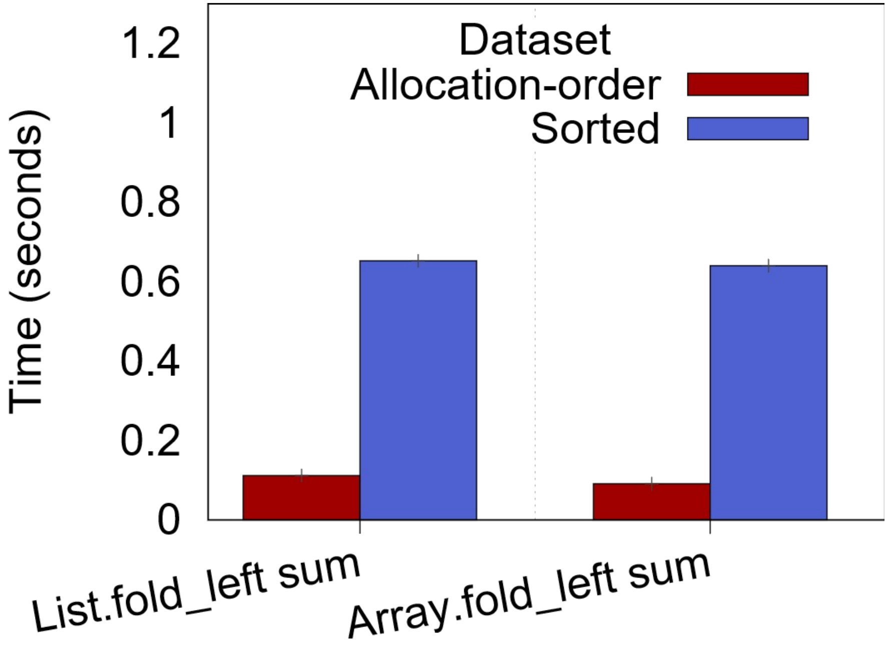
```
let rec fold_left f accu l =
  match l with
    [] -> accu
  | a::l -> fold left f (f accu a) l;;
```



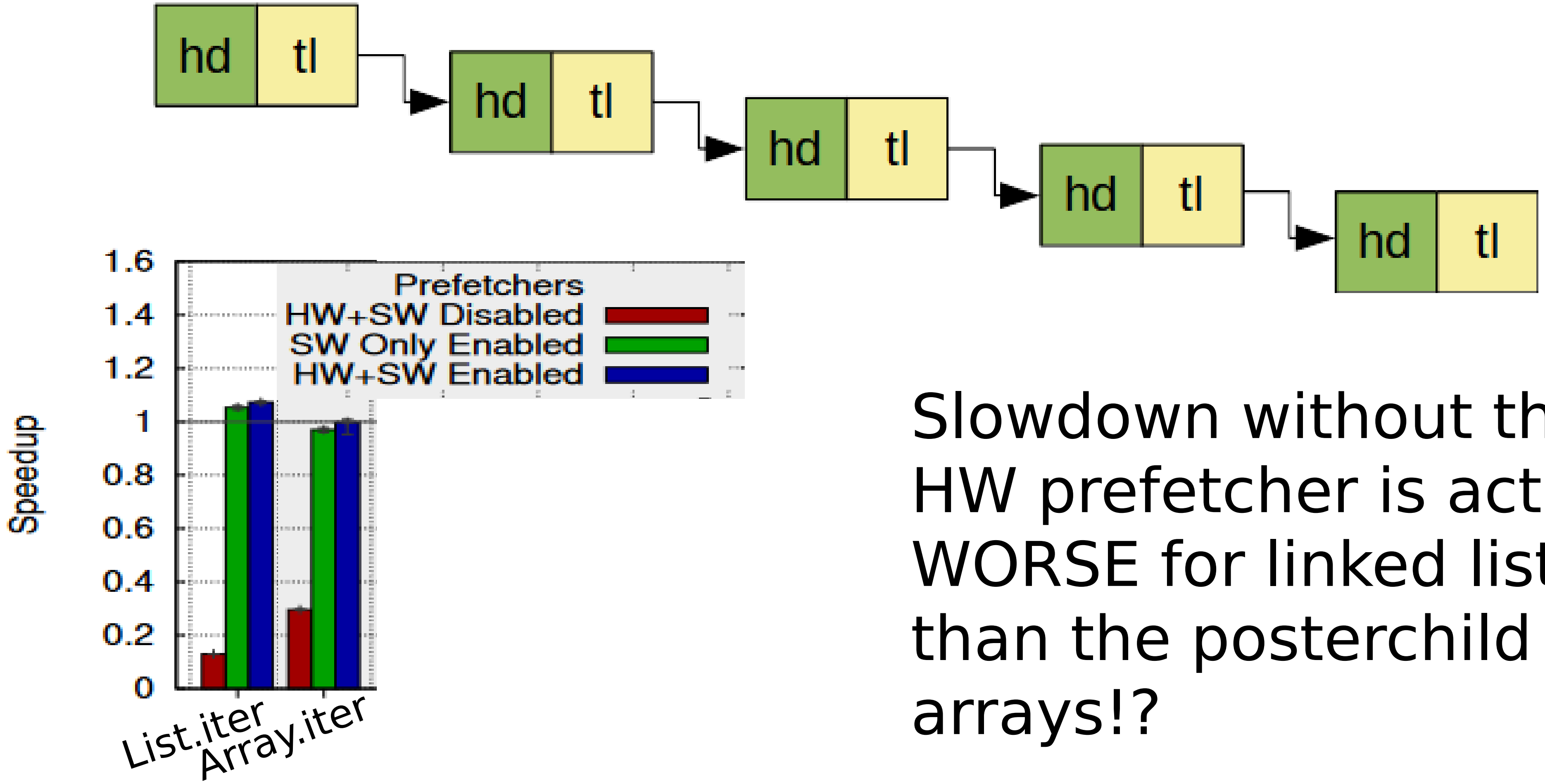
...but actually, linked lists aren’t as bad as you’d expect

```
let rec fold_left f accu l =
  match l with
    [] -> accu
  | a::l -> fold left f (f accu a) l;;
```



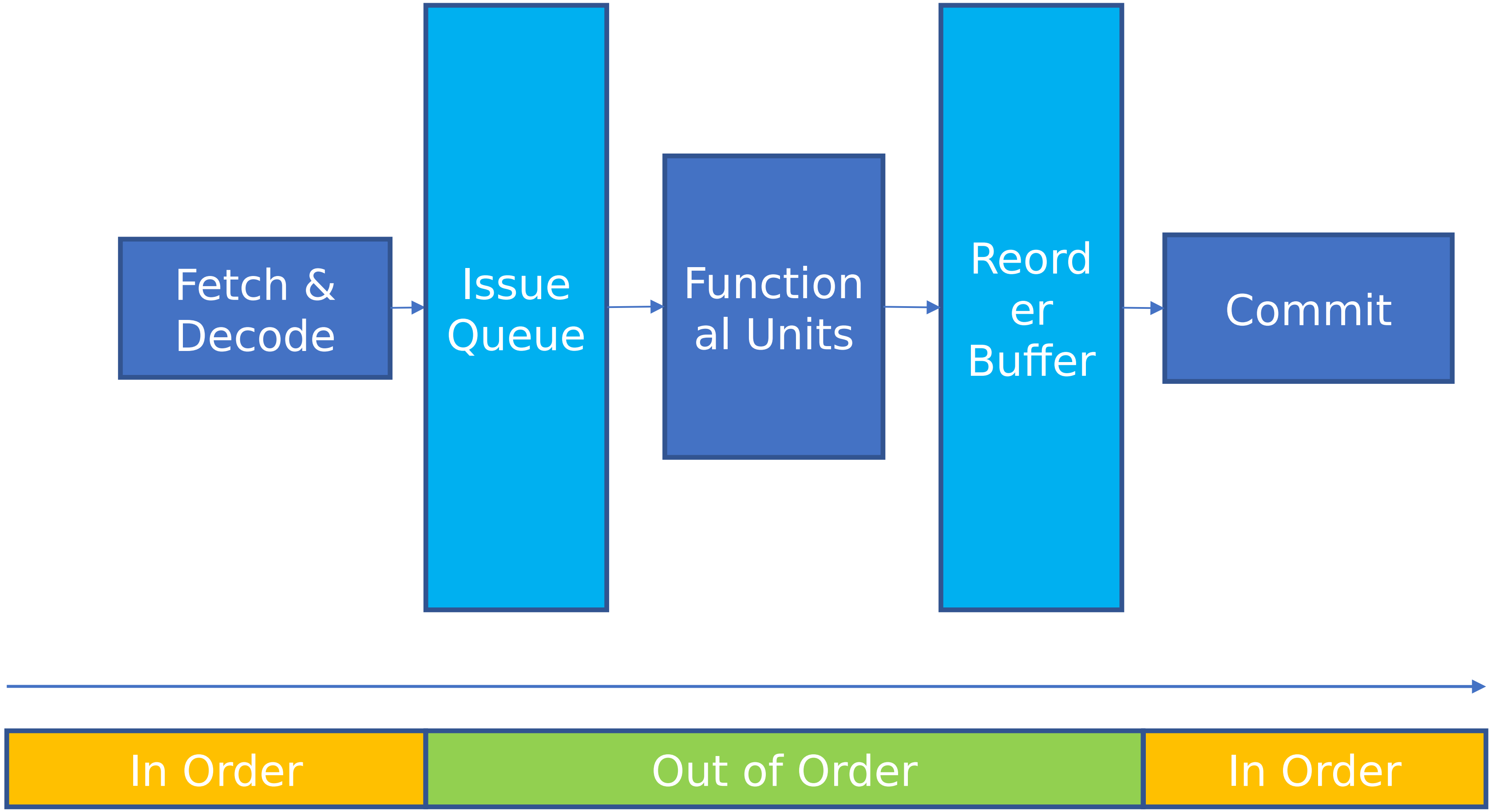


Our next clue: List.iter on unboxed integers with/without hardware prefetcher

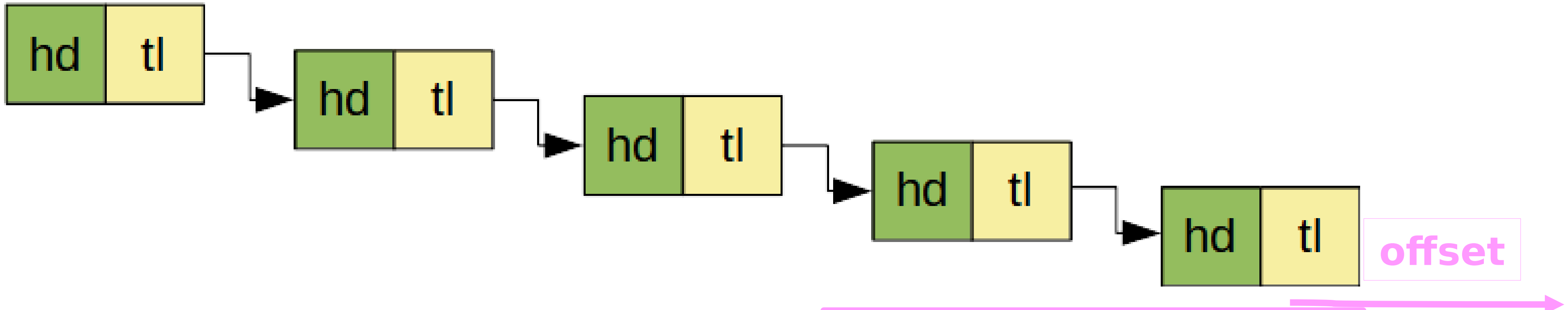


Slowdown without the HW prefetcher is actually WORSE for linked lists than the posterchild arrays!?

Out-Of-Order Execution



Idea: Jump off the deep end



```
prefetchoffset l (-256);
```

Idea: Jump off the deep end

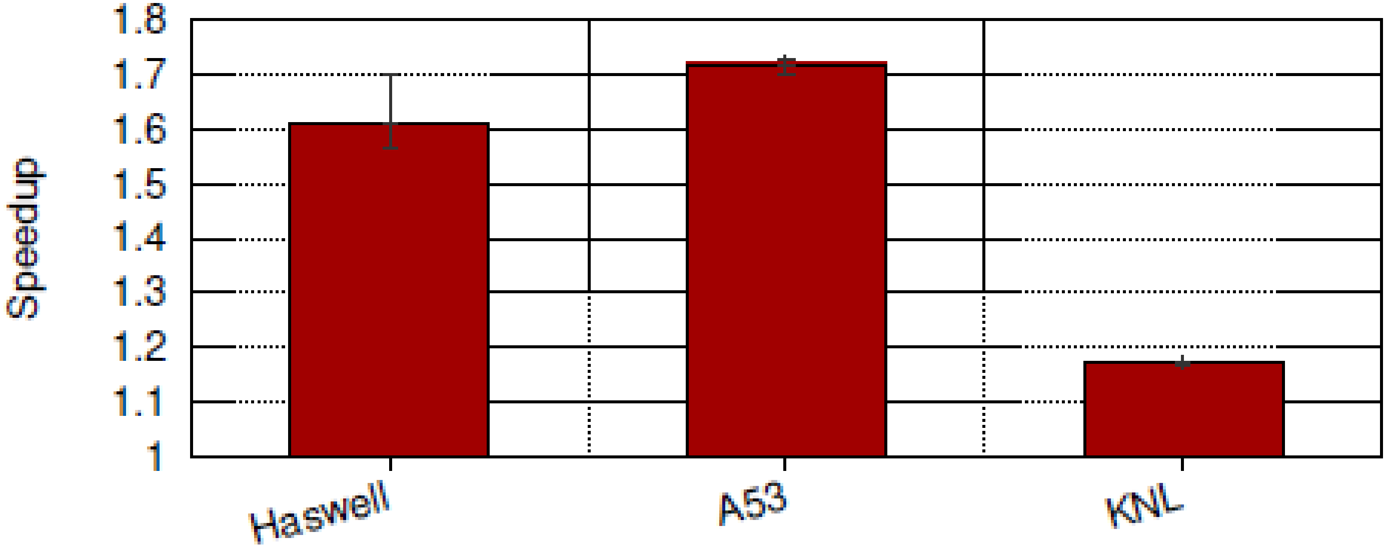
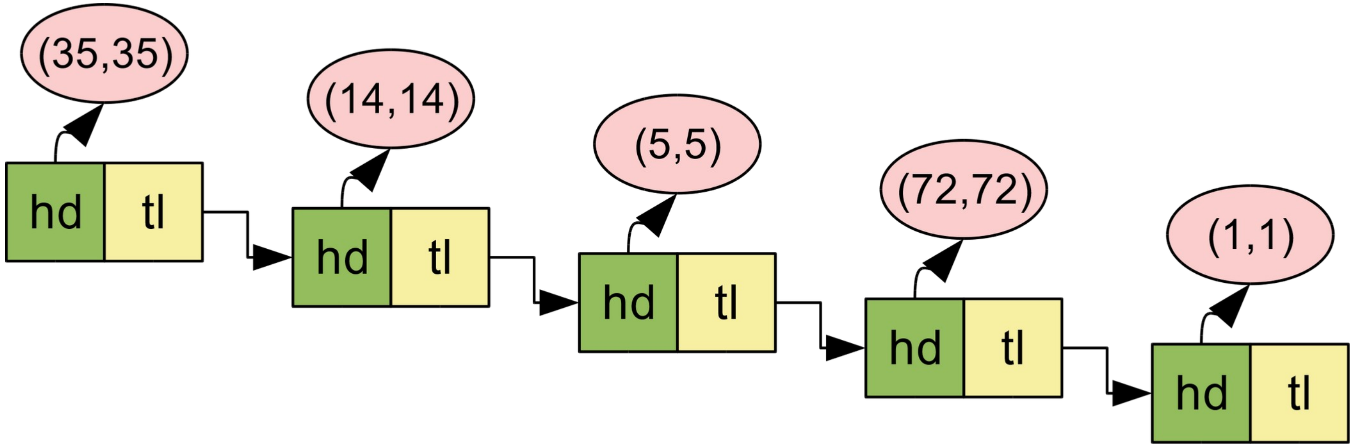
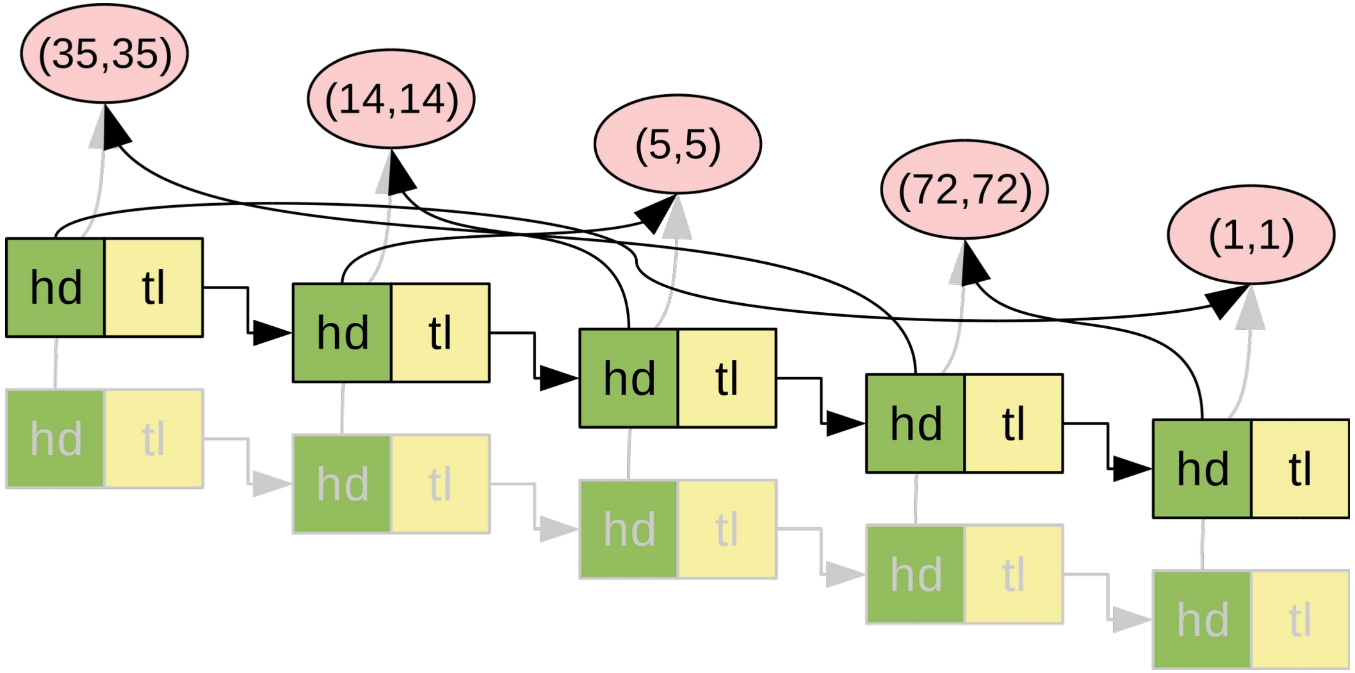


Figure 9. Speedup for prefetching *List.nth* on each system. Performance improvement remains consistent regardless of the linked list type, as the linked list data is not accessed.

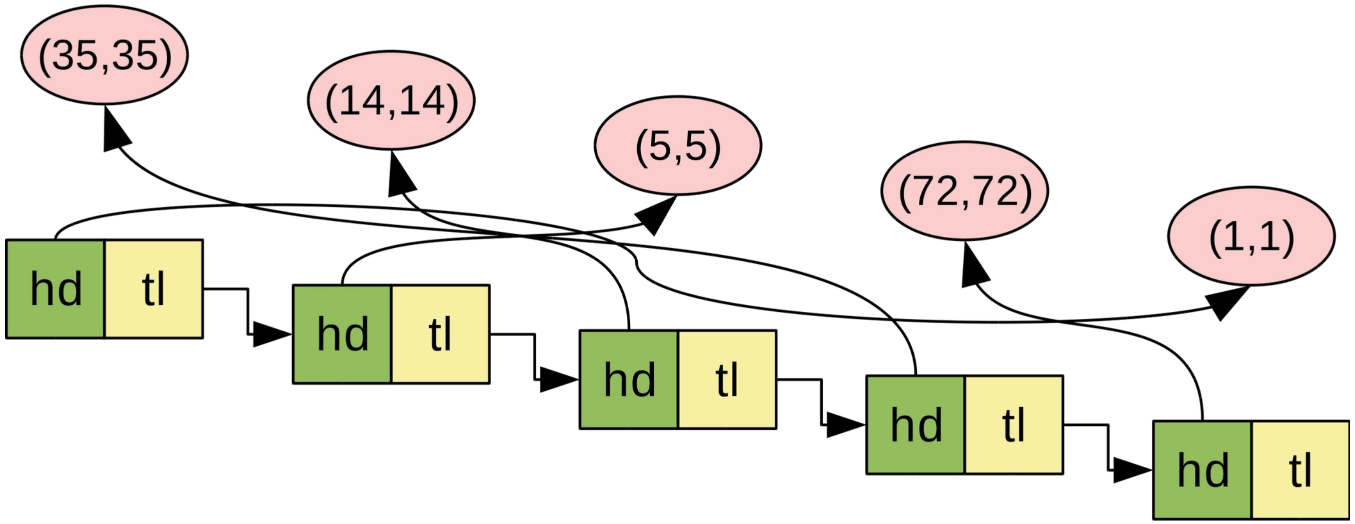
Unsorted Lists



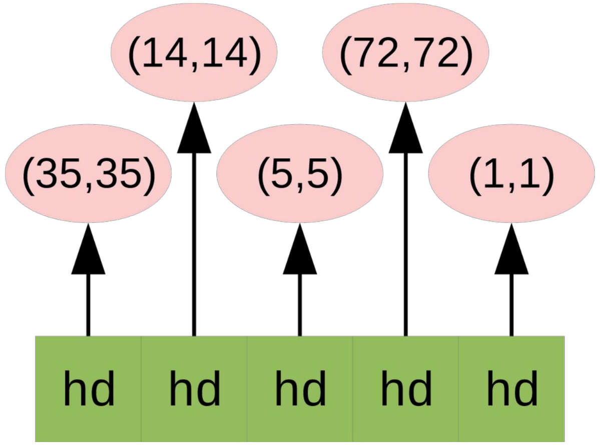
Sorted Lists



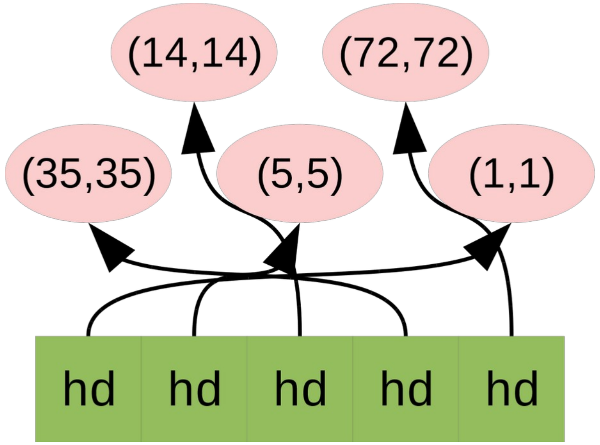
Sorted Lists



Unsorted Arrays

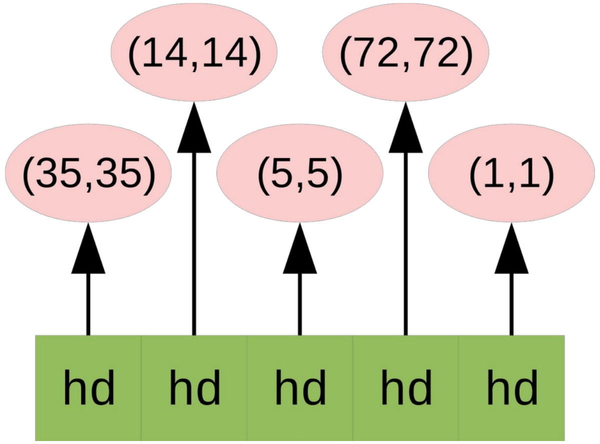


Sorted Arrays



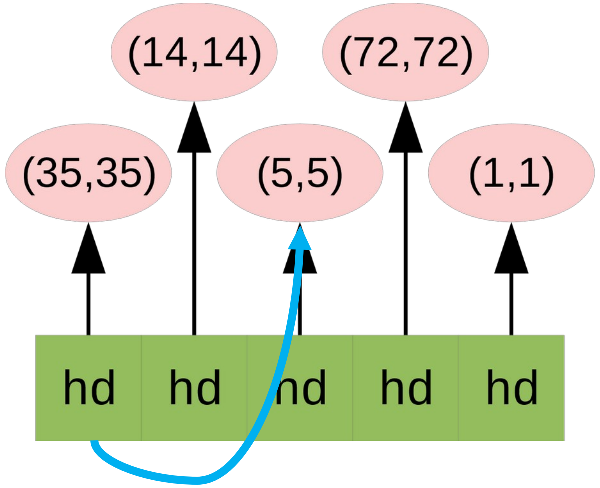
Array Prefetching

```
let fold_left_array f x a =
  let r = ref x in
  for i = 0 to length a - 1 do
    r := f !r (Array.unsafe_get a i)
  done;
  !r;;
```



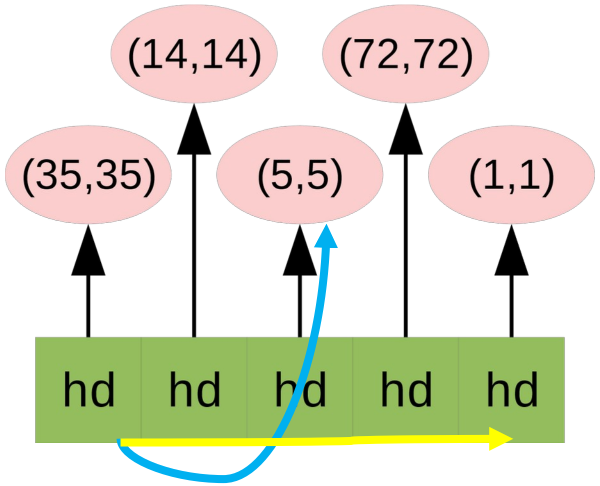
Array Prefetching

```
let fold_left_array f x a =
  let r = ref x in
  for i = 0 to length a - 1 do
    prefetch(Array.unsafe_get a (min (i+16)
    ((length a) - 1) ) );
    r := f !r (Array.unsafe_get a i)
  done;
  !r;;
```

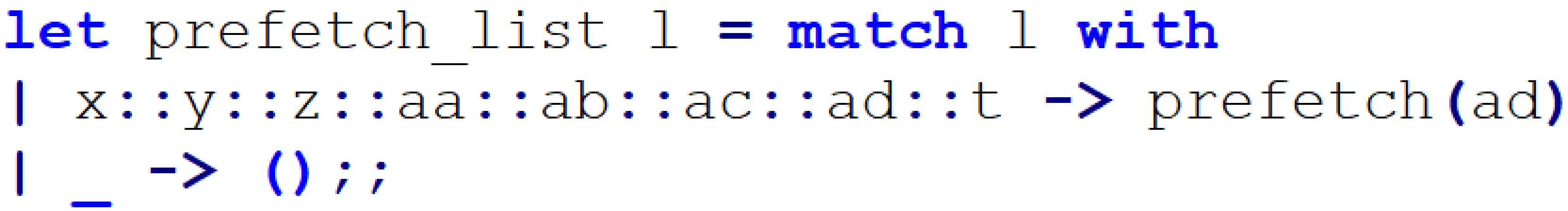


Array Prefetching

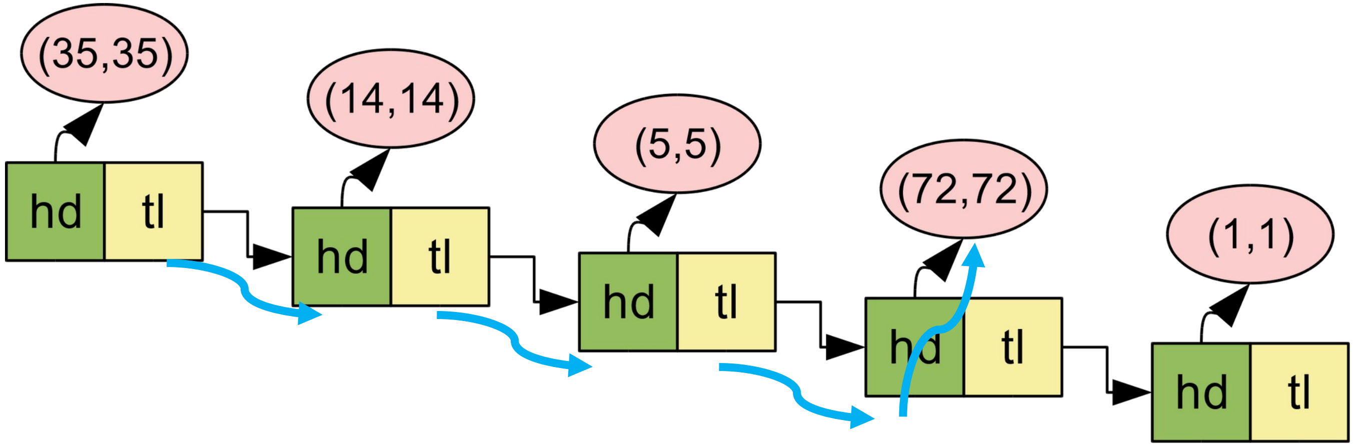
```
let fold_left_array f x a =
  let r = ref x in
  for i = 0 to length a - 1 do
    Array.array_prefetch a (i+32);
    prefetch(Array.unsafe_get a (min (i+16)
    ((length a) - 1) ) );
    r := f !r (Array.unsafe_get a i)
      ;
  !r;;
```



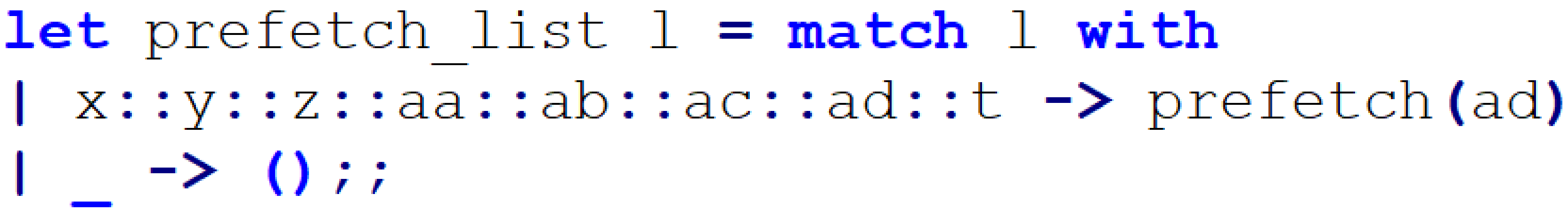
List Simple Prefetching



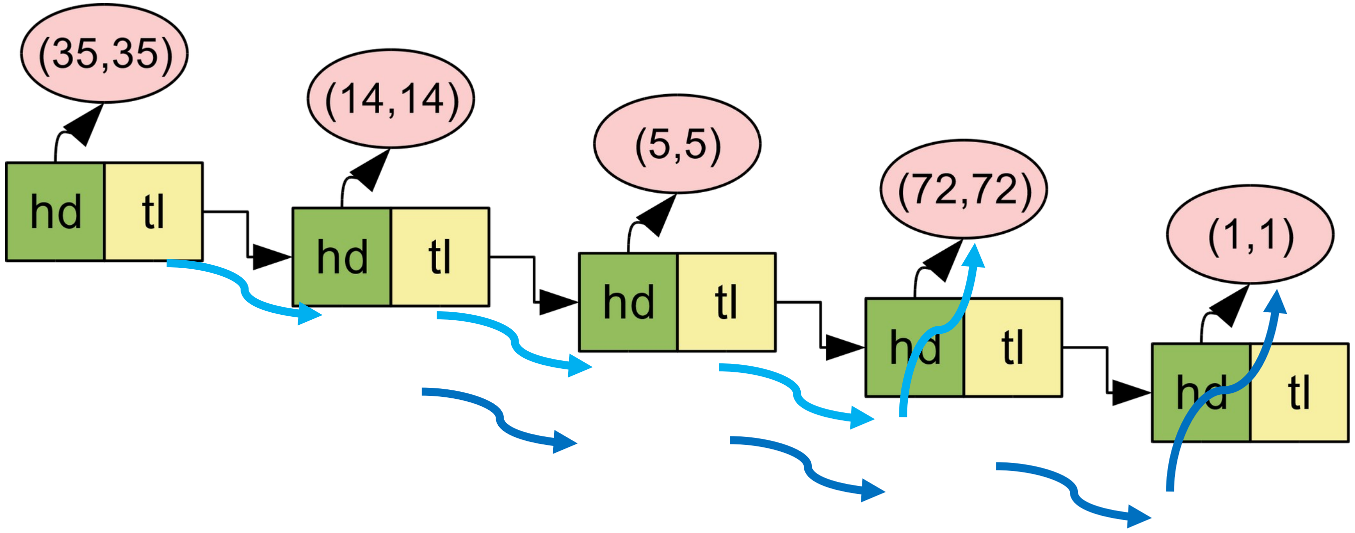
```
let prefetch_list l = match l with
| x::y::z::aa::ab::ac::ad::t -> prefetch(ad)
| _ -> ();;
```



List Simple Prefetching

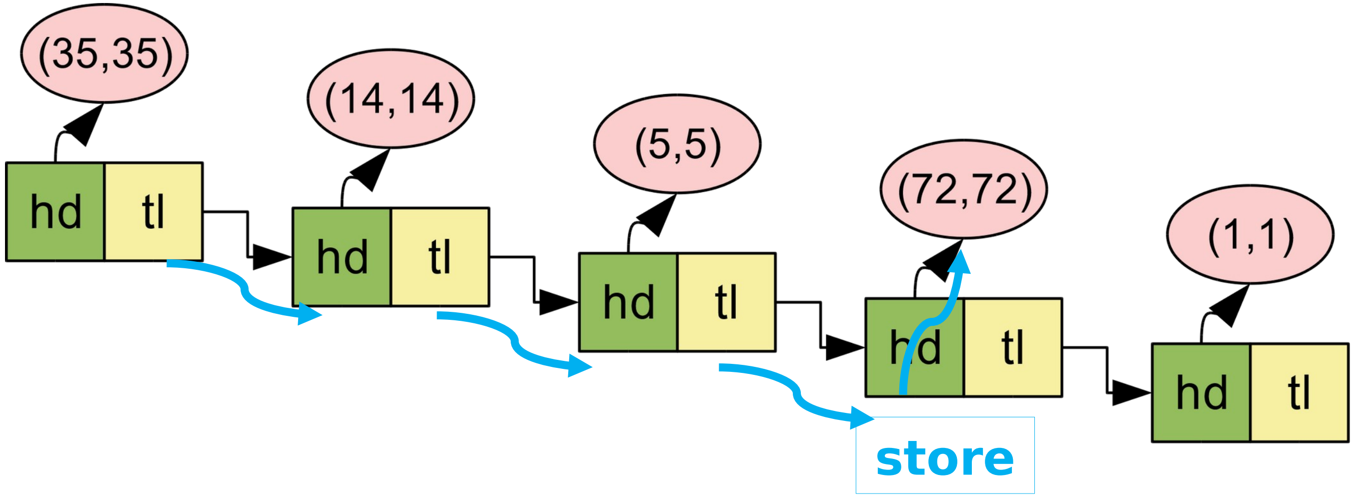


```
let prefetch_list l = match l with
| x::y::z::aa::ab::ac::ad::t -> prefetch(ad)
| _ -> ();;
```



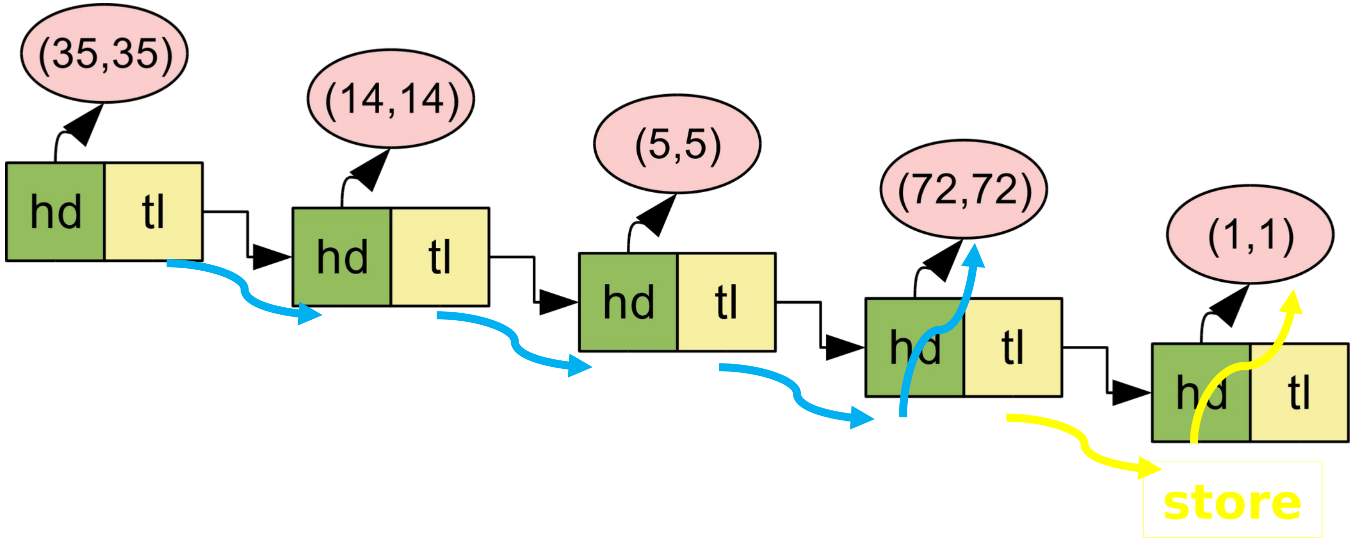
List Complex Prefetching

```
let prefetch_prefind l = match l with
|cx::cy::cz::caa::cab::cac::cae::cbad::cbx::cby::cbz::cbaa::cbab::cbac::cbae::cad::x
::y::z::aa::ab::ac::ae::bad::bx::by::bz::baa::bab::bac::bae::ad::t -> prefetch(ad); t
| _ -> [];;
```



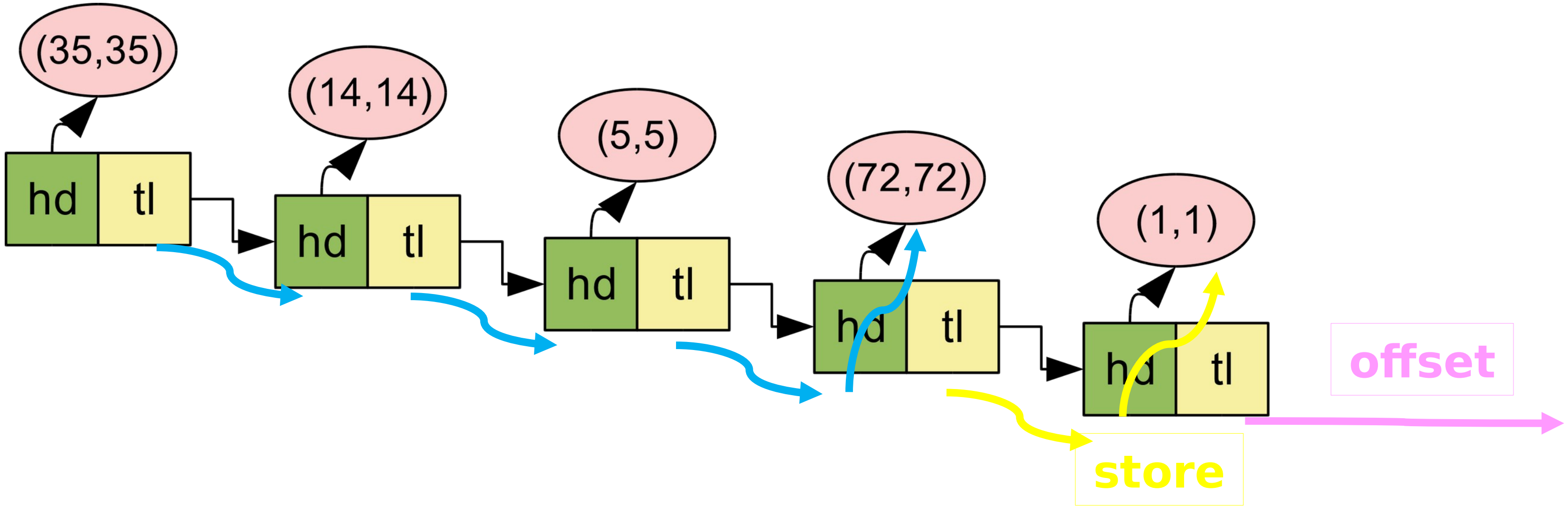
List Complex Prefetching

```
let prefetch_prefind l = match l with
|cx::cy::cz::caa::cab::cac::cae::cbad::cbx::cby::cbz::cbaa::cbab::cbac::cbae::cad::x
::y::z::aa::ab::ac::ae::bad::bx::by::bz::baa::bab::bac::bae::ad::t -> prefetch(ad); t
| _ -> [];;

let prefetch_prefound m = match m with
| x::ys -> prefetch(x);ys
| _ -> [];;
```



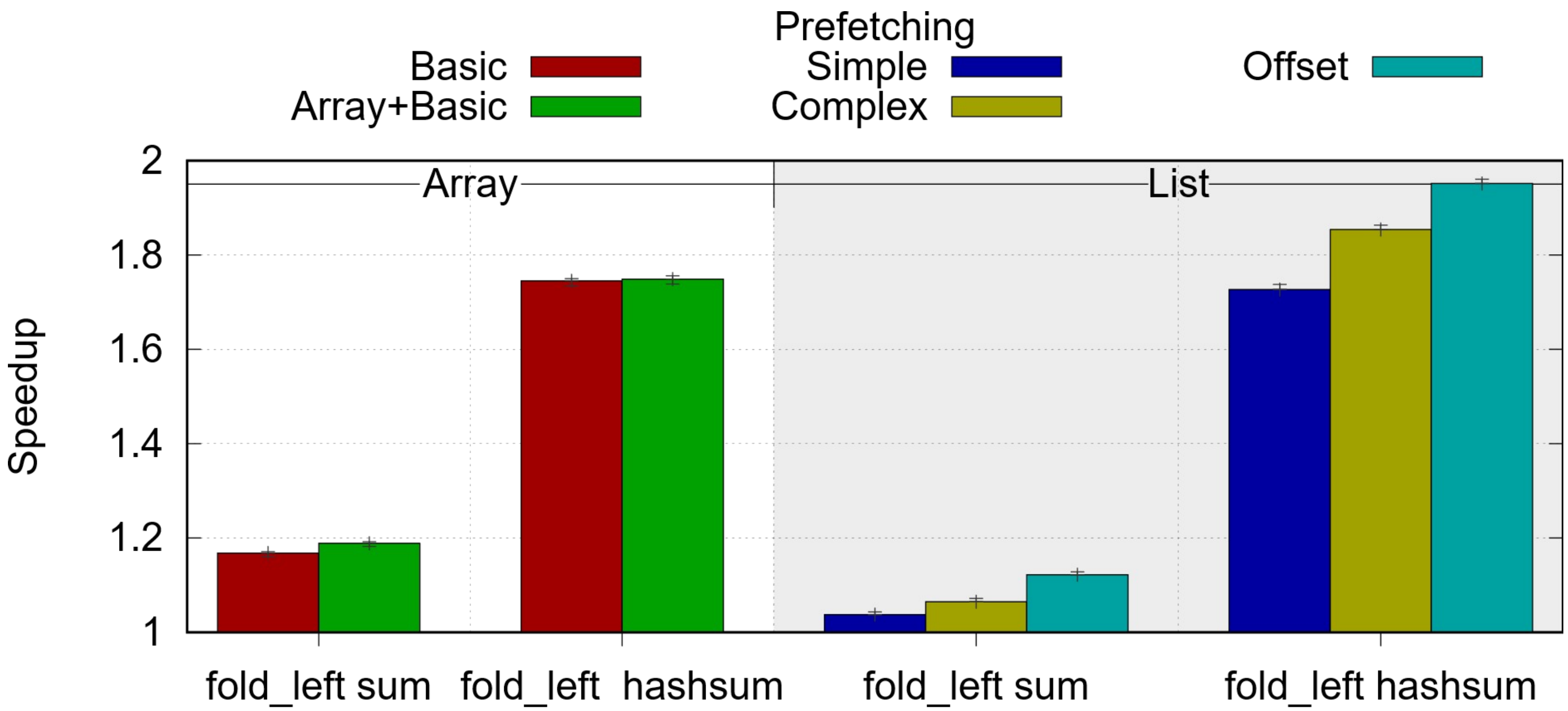
List Complex + Offset Prefetching

```
let prefetch_prefind l = match l with
|cx::cy::cz::caa::cab::cac::cae::cbad::cbx::cby::cbz::cbaa::cbab::cbac::cbae::cad::x
::y::z::aa::ab::ac::ae::bad::bx::by::bz::baa::bab::bac::bae::ad::t -> prefetch(ad); t
| _ -> [];;

let prefetch_prefound m = match m with
| x::ys -> prefetch(x);ys
| _ -> [];;
```

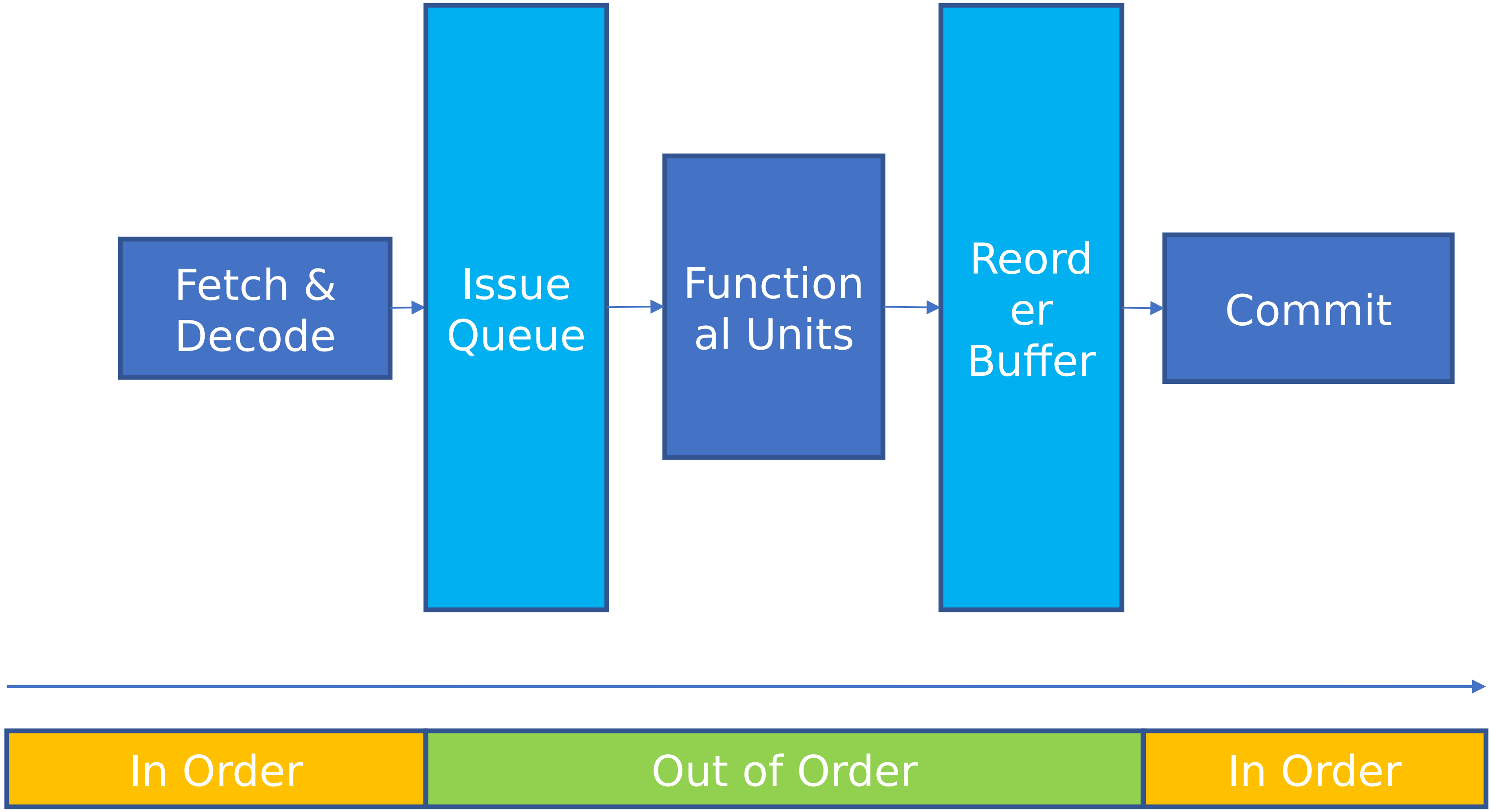


```
prefetchoffset l (-256);
```

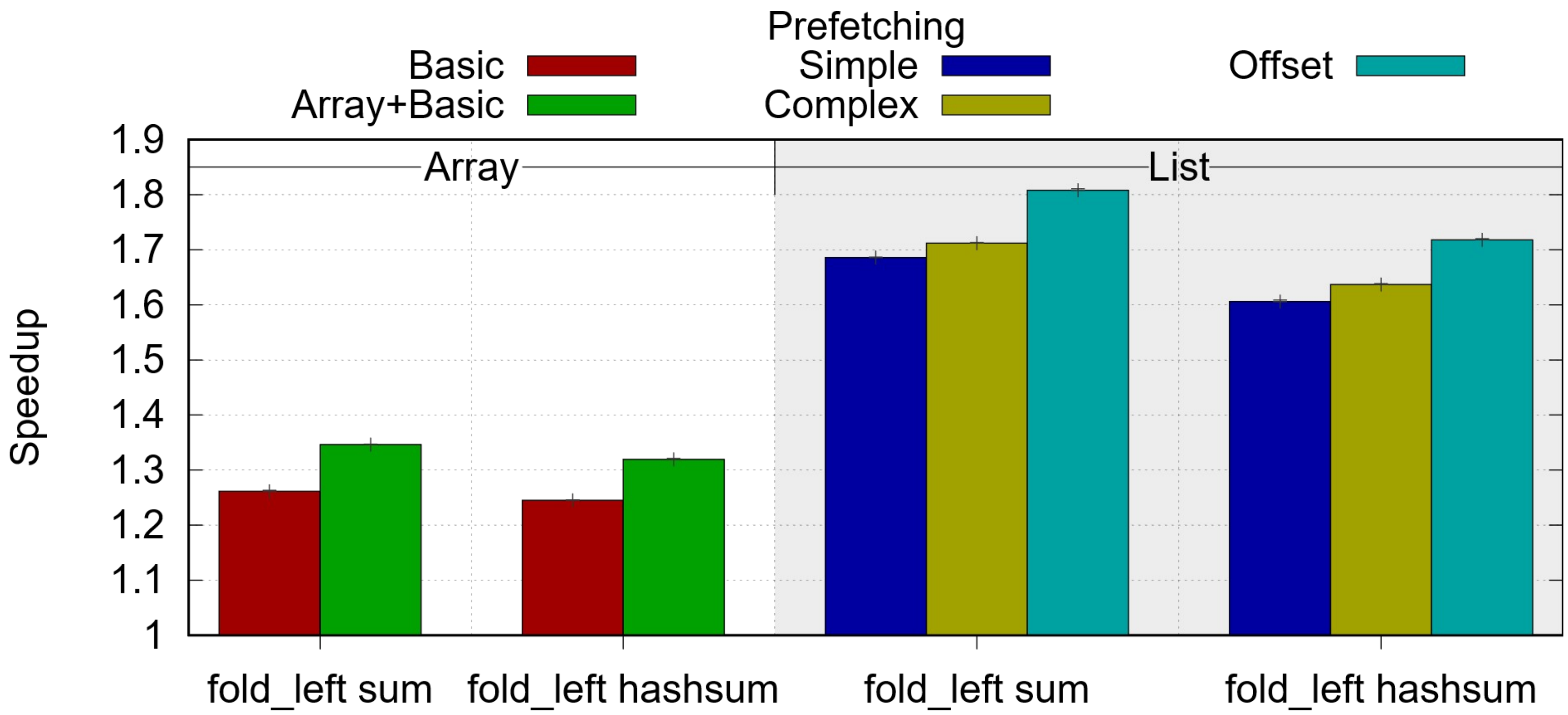
Fold_Left (Intel Haswell)



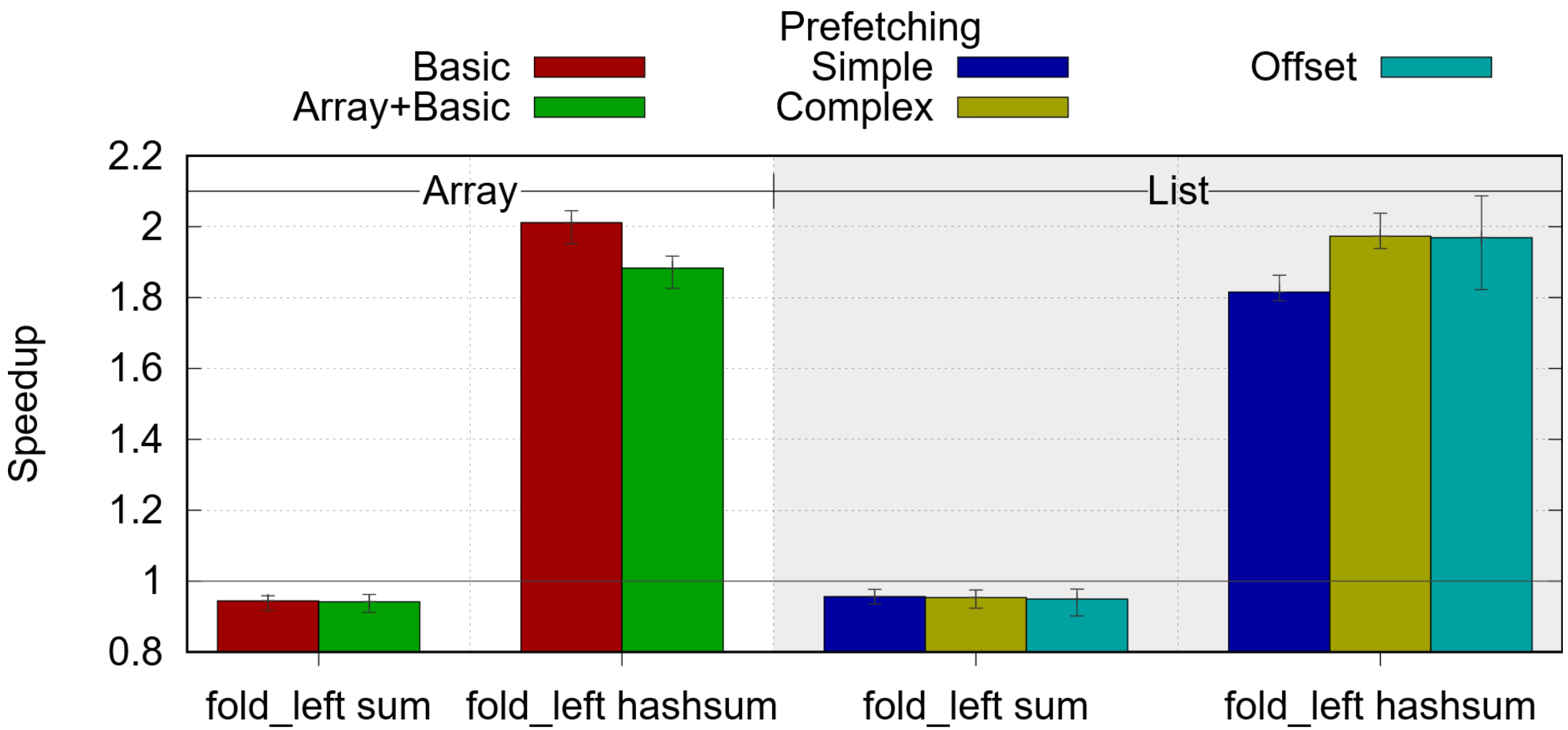
Out-Of-Order Execution



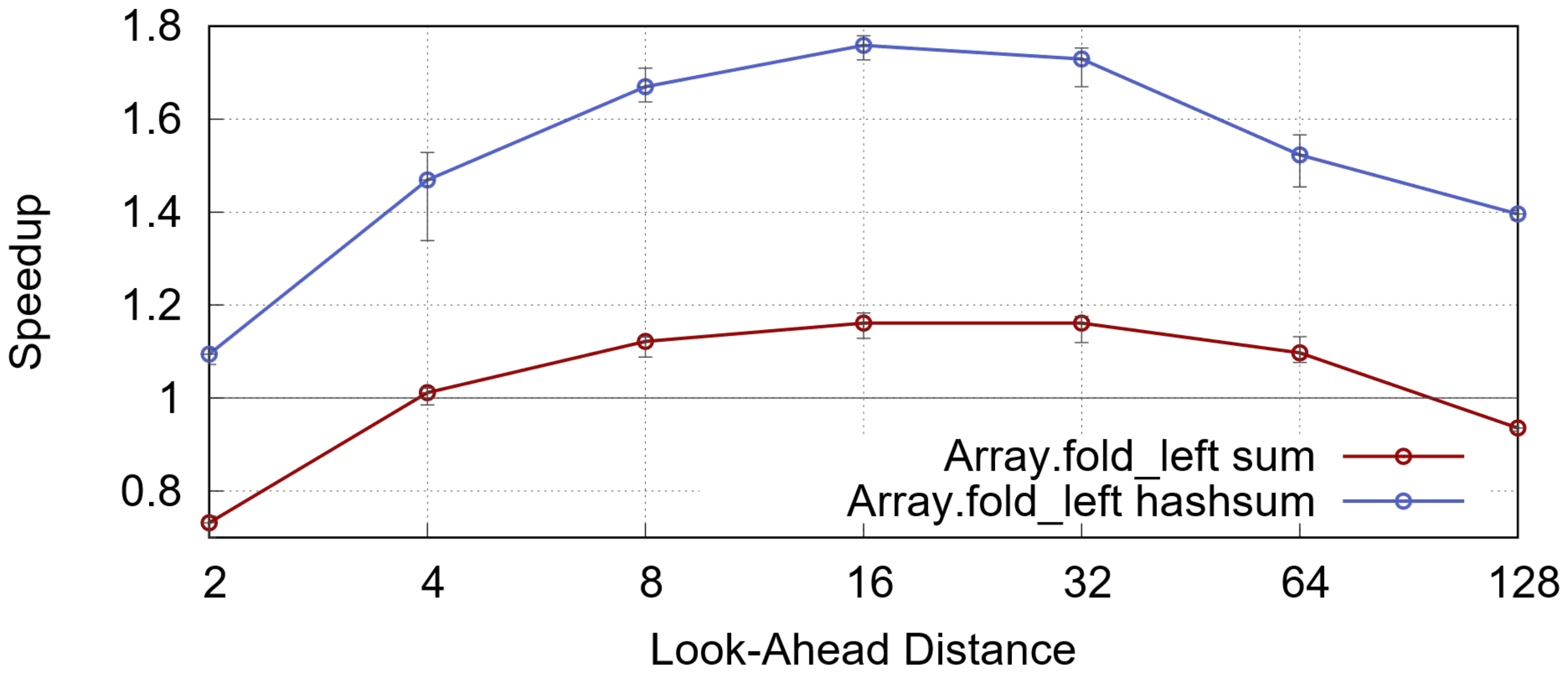
Fold_Left (Arm Cortex A53)



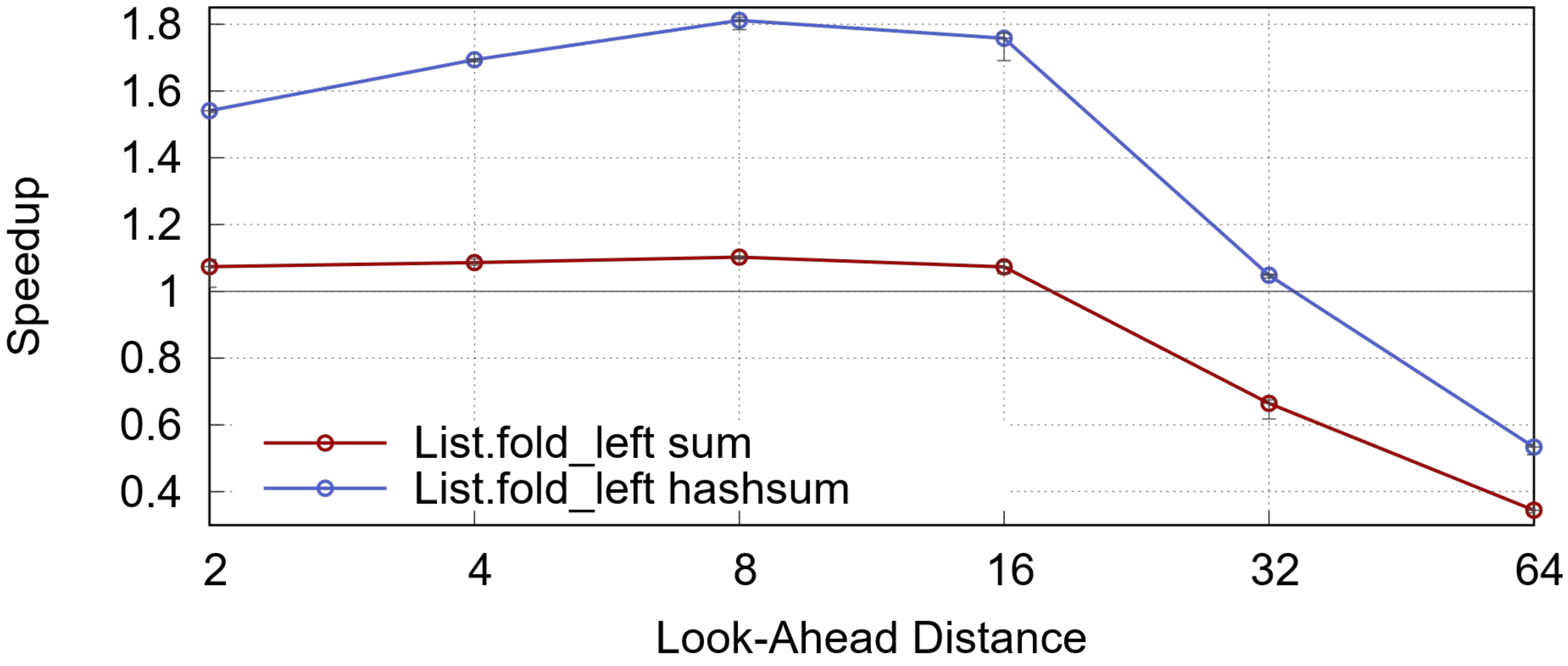
Fold_Left (Intel Xeon Phi Knights Landing)



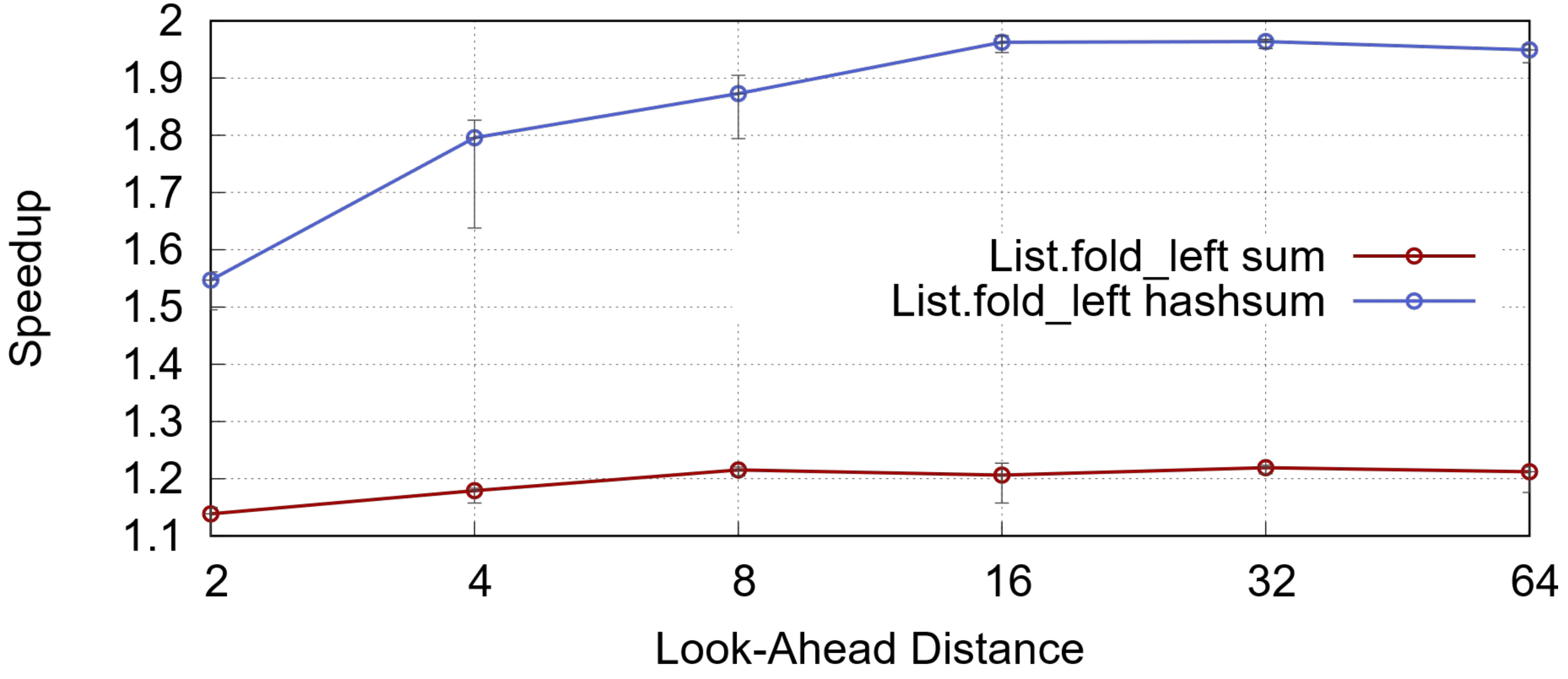
Fetch Distances (Arrays)



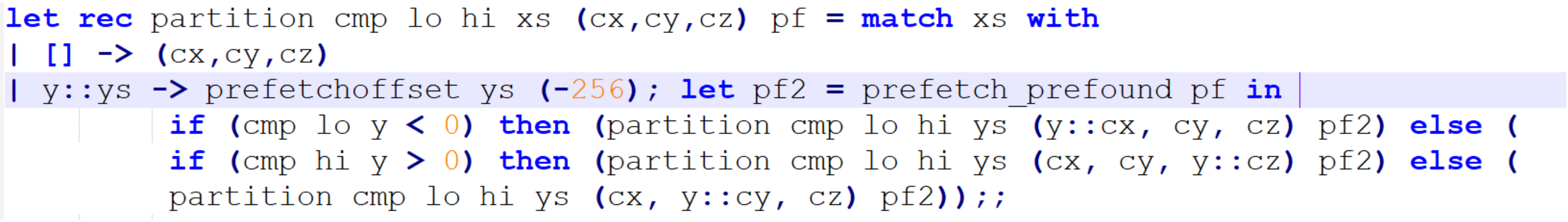
Fetch Distances (List Simple)



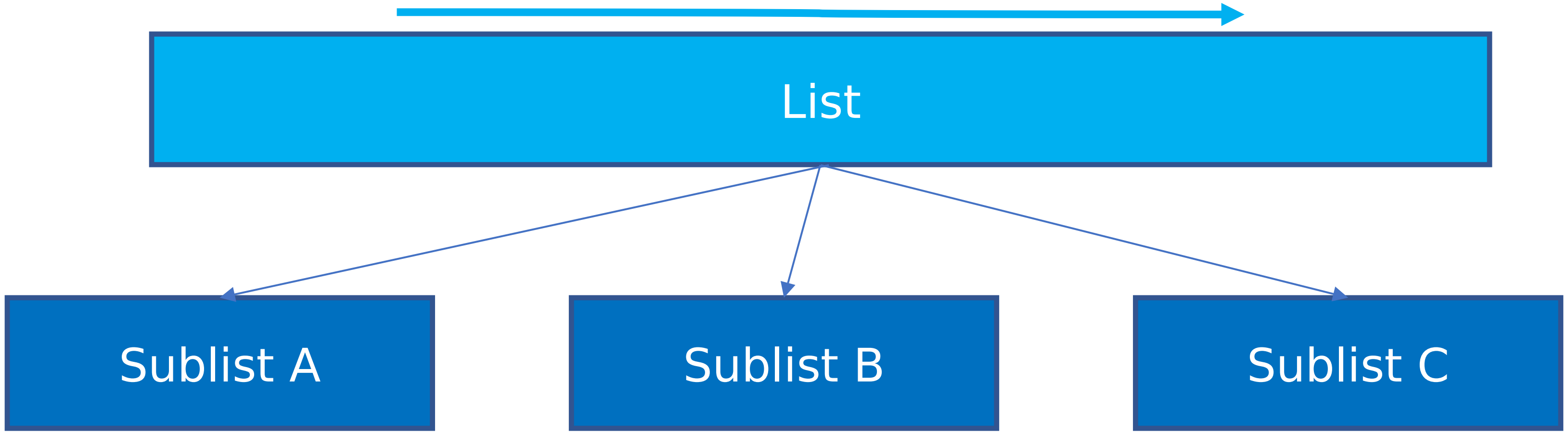
Fetch Distances (List Complex)



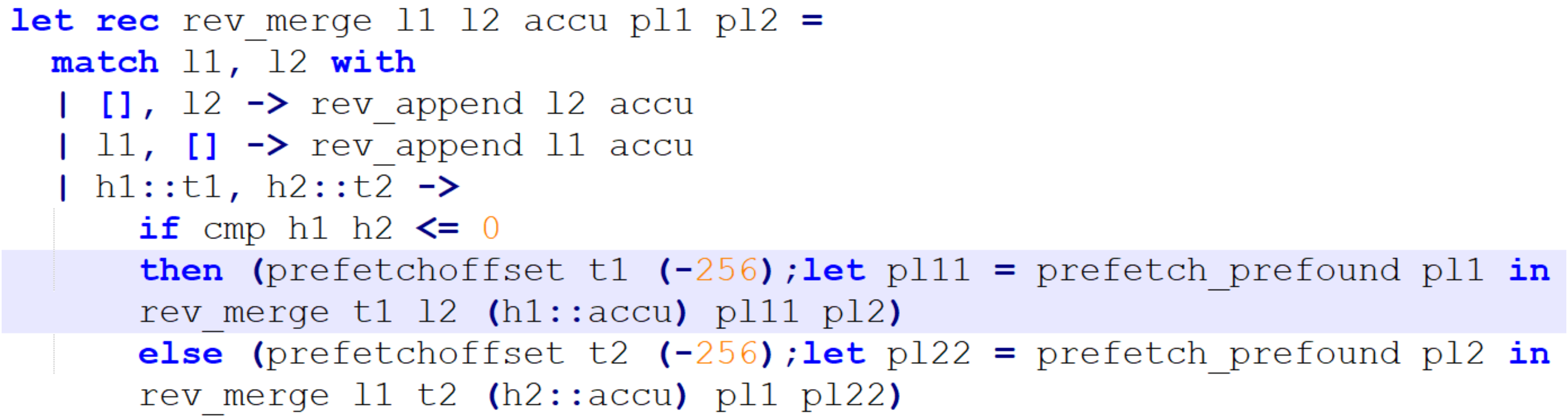
Quicksort



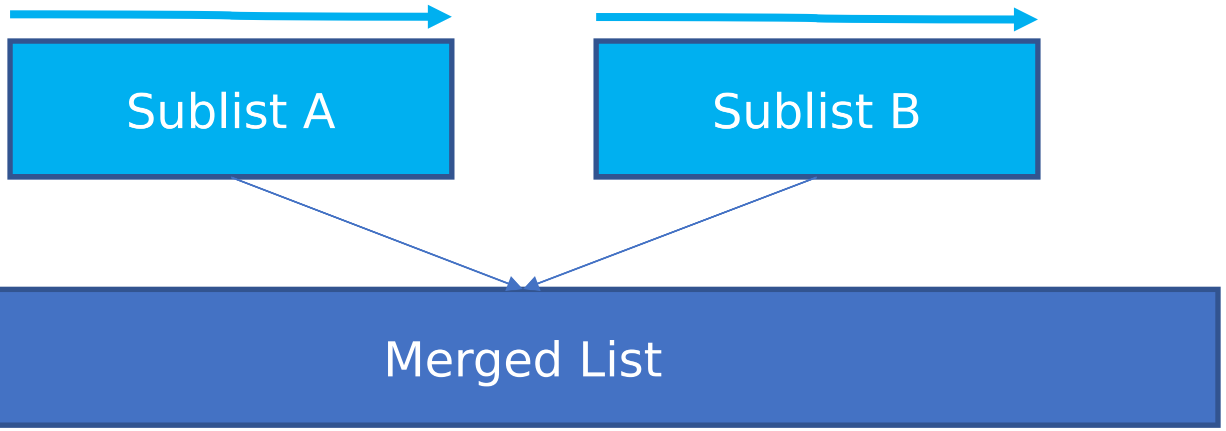
```
let rec partition cmp lo hi xs (cx,cy,cz) pf = match xs with
| [] -> (cx,cy,cz)
| y::ys -> prefetchoffset ys (-256); let pf2 = prefetch_prefound pf in
         if (cmp lo y < 0) then (partition cmp lo hi ys (y::cx, cy, cz) pf2) else (
         if (cmp hi y > 0) then (partition cmp lo hi ys (cx, cy, y::cz) pf2) else (
         partition cmp lo hi ys (cx, y::cy, cz) pf2));;
```



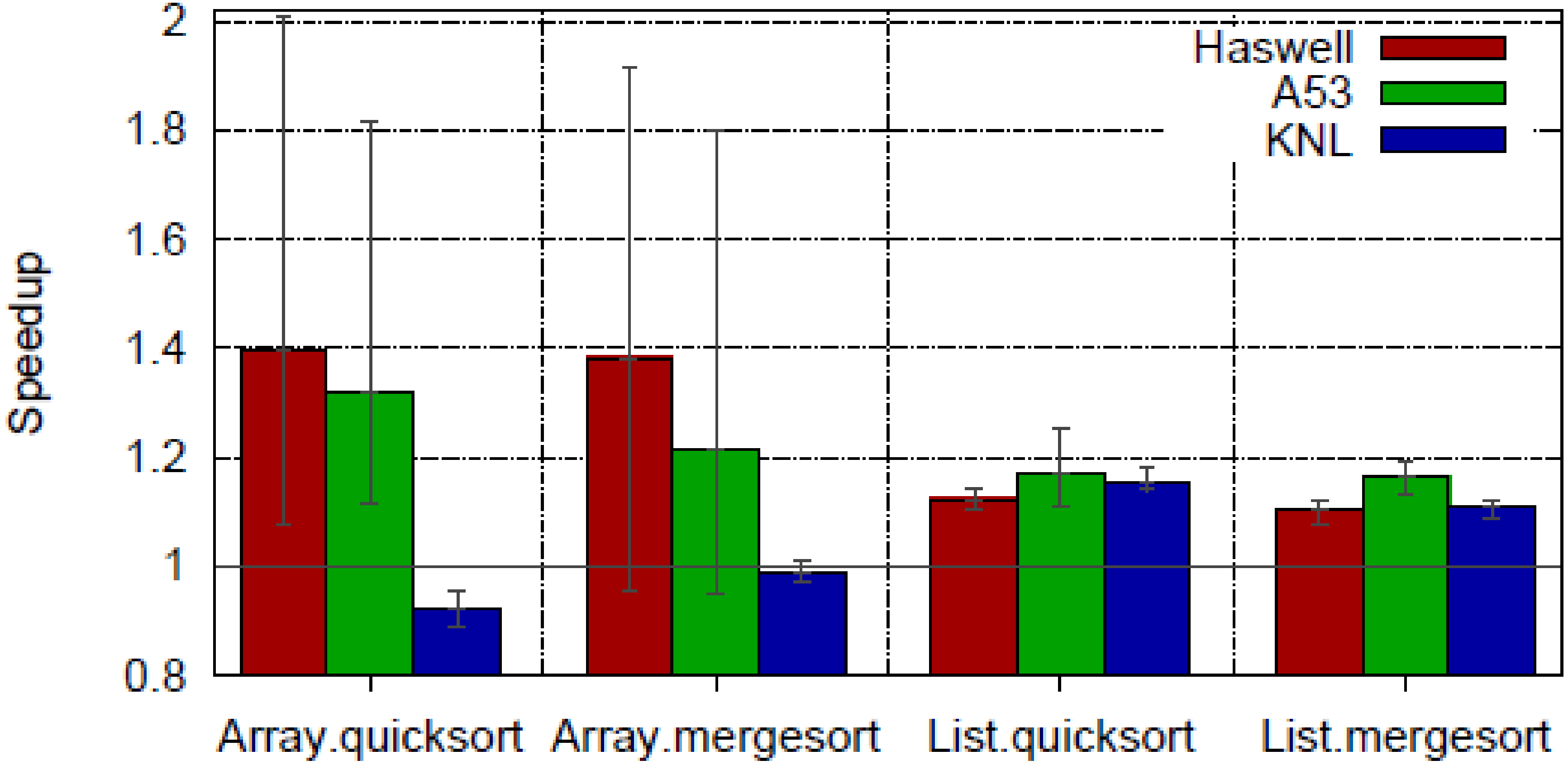
Mergesort



```
let rec rev_merge l1 l2 accu pl1 pl2 =
  match l1, l2 with
  | [], l2 -> rev_append l2 accu
  | l1, [] -> rev_append l1 accu
  | h1::t1, h2::t2 ->
      if cmp h1 h2 <= 0
      then (prefetchoffset t1 (-256);let pl11 = prefetch_prefound pl1 in
      rev_merge t1 l2 (h1::accu) pl11 pl2)
      else (prefetchoffset t2 (-256);let pl22 = prefetch_prefound pl2 in
      rev_merge l1 t2 (h2::accu) pl1 pl22)
```



See it, Say it, Sorted

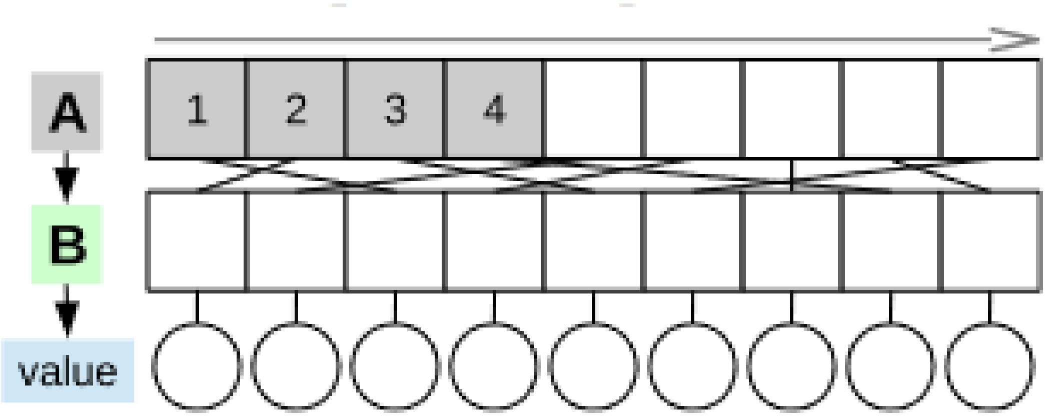


What does this all mean for microarchitecture and compilers on functional languages?

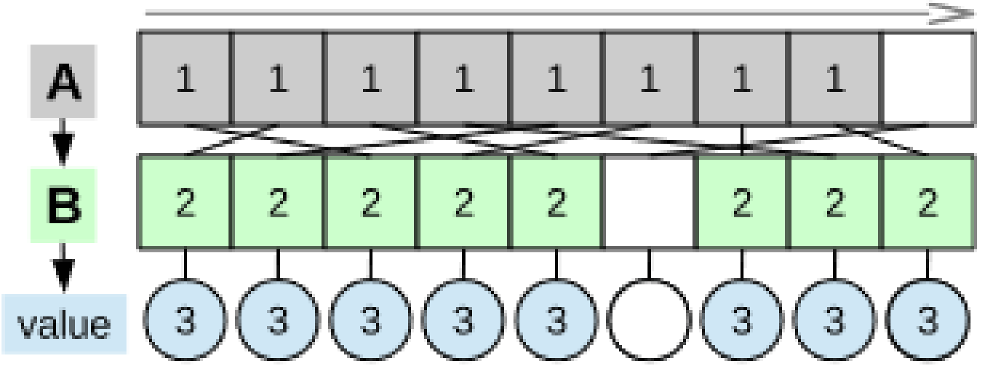
- Out-of-order cores are bad at linked data structures – too many data dependencies
- Hardware prefetchers are good at linked lists – they can value-predict past the data dependencies if the allocator does its job
- Software prefetching can do much better, both in terms of aggression, and following new patterns… and it has no (hard) state so kind-of suitable for functional langs
- BUT – the correct software prefetch depends not only on the code, but the dataset distribution and size (so run-time information…)

Another thought: Vector Runahead
(ISCA 2021, MICRO 2023, Top Picks 2022 and 2024)

```
for (int x=0; x<N; x++)
y += B[hash(A[x])]->value;
```



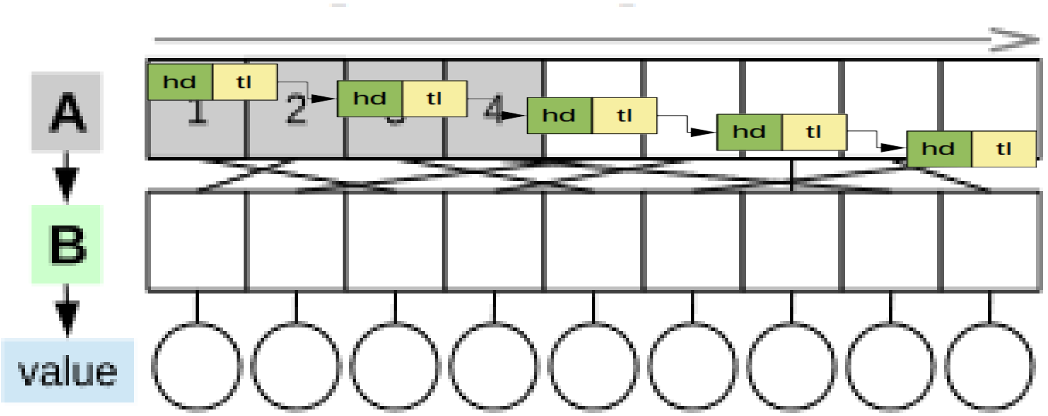
Stride prefetcher



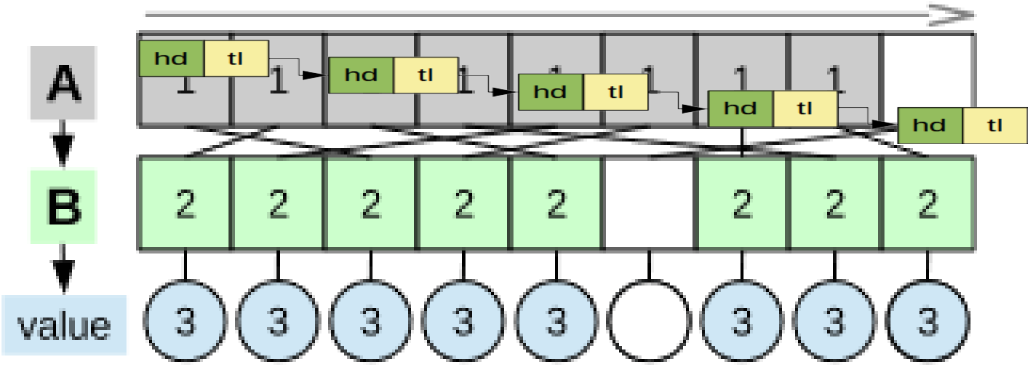
Vector Runahead: learns from the stride prefetcher, spawns new gather code in vectors

Another thought: Vector Runahead
(ISCA 2021, MICRO 2023, Top Picks 2022 and 2024)

```
for (int x=0; x<N; x++)
y += B[hash(A[x])]->value;
```

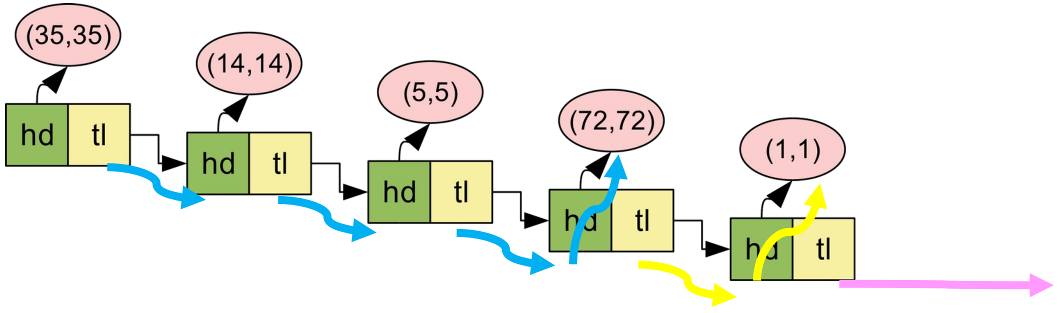


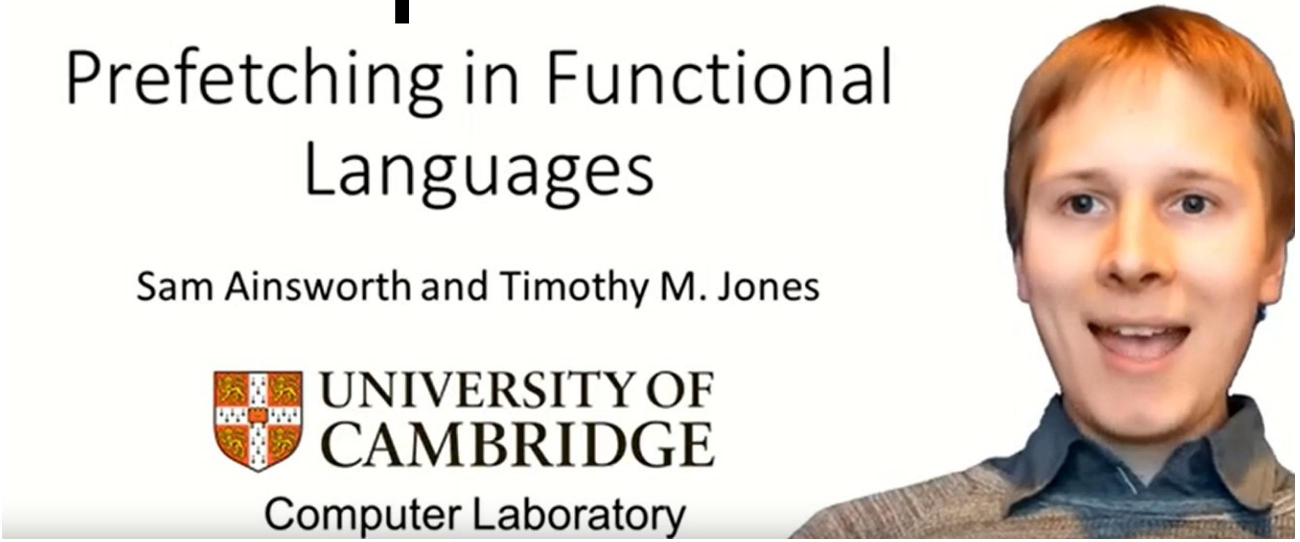
Stride prefetcher



Vector Runahead: learns from the stride prefetcher, spawns new gather code in vectors

Prefetching in Functional Languages: A Hardware-Software Retrospective





Prefetching in Functional Languages

Sam Ainsworth and Timothy M. Jones

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Watch the ISMM Talk!

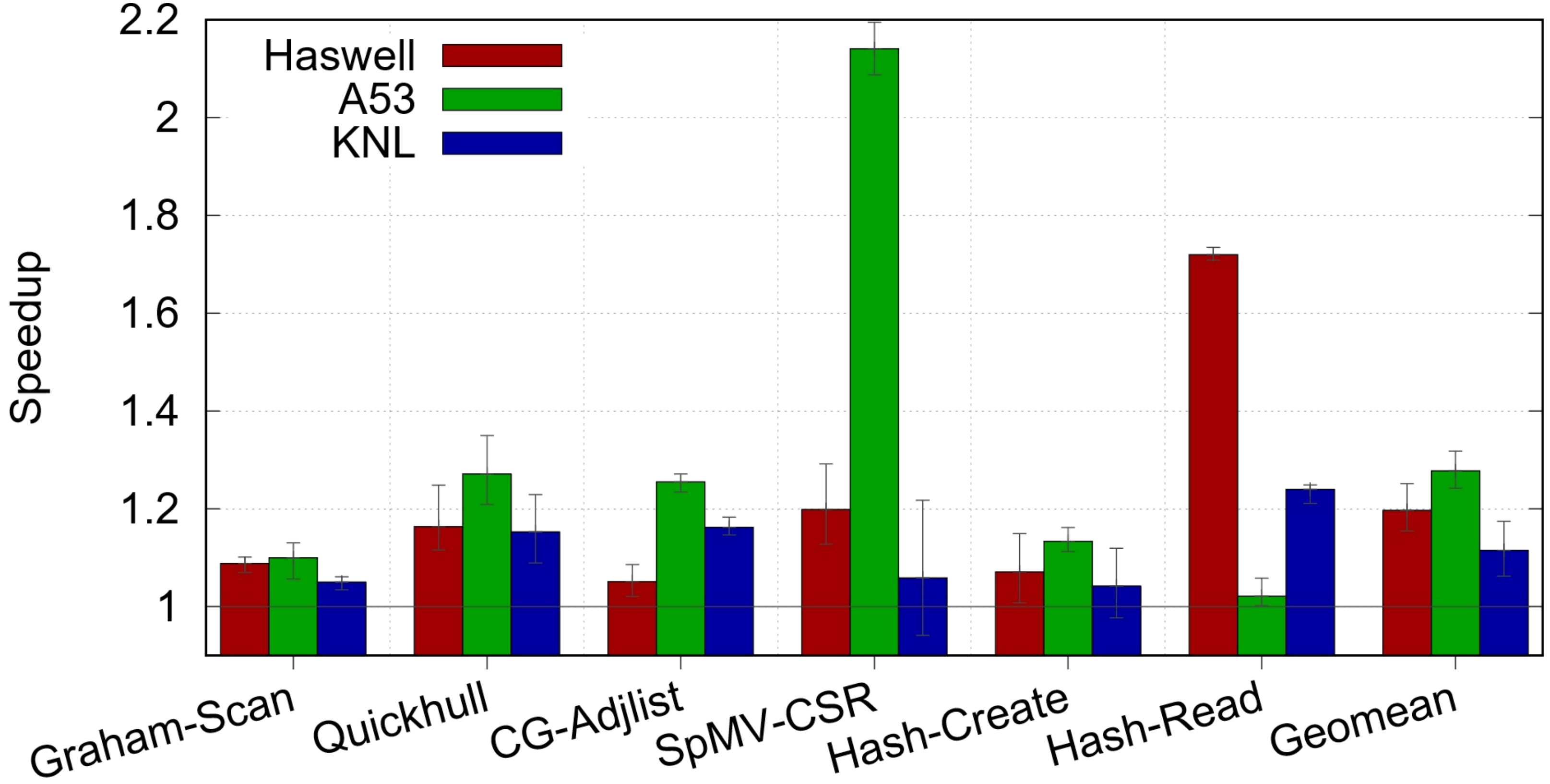
https://www.youtube.com/watch?v=5fTt1I1ePCg

Sam Ainsworth

OCaml memory-bound benchmark suite

Workload	Prefetching	Description
Graham Scan	List Complex, Offset (via sorting algorithm)	Calculates the convex hull of a list of integer pairs, by sorting the points then categorizing them.
Quickhull	List Complex, Offset	Calculates the convex hull of a list of integer pairs, using a quicksort-style divide-and-conquer approach.
CG-Adjlist	Array, Basic, Offset	Conjugate-gradient solving for graphs in adjacency-list format.
SpMV-CSR	Array, Basic	Performs sparse matrix-vector multiplication on graphs based on an efficient compressed sparse-row (CSR) representation.
Hash-Create	Array	Times the creation and filling of a large hash table.
Hash-Read	Array, Basic	Times reading all the elements of a large hash table.

OCaml memory-bound benchmark suite


Speedup
Haswell
A53
KNL
Graham-Scan
Quickhull
CG-Adjlist
SpMV-CSR
Hash-Create
Hash-Read
Geomean