Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design

Nishil Talati[*], Kyle May[†], Armand Behroozi[*], Yichen Yang[*], Kuba Kaszyk[‡], Christos Vasiladiotis, Tarunesh Verma[*], Lu Li[‡], Brandon Nguyen[*], Jiwen Sun[‡], John Magnus Morton[‡], Agreen Ahmadi[*], Todd Austin[*], Michael F P O'Boyle[‡], Scott Mahlke[*], Trevor Mudge[*], Ronald Dreslinski[*]*

*University of Michigan †University of Wisconsin, Madison ‡University of Edinburgh

HPCA 2021, Seoul, South Korea

Presented by Paul Scheffler

Executive Summary

Problem	• Data-indirect irregular workloads are **bottlenecked by the memory system** • Common prefetchers **fail to accelerate** indirect memory accesses • Specialized prefetchers not **general, performant, or timely** enough
Goal	• A **general, effective, low-cost prefetcher** for data-indirect workloads
Key Idea	• Most irregular access patterns are composed of **two specific patterns**: **single-valued** and **ranged indirection**
Mechanism	• SW: encode indirect access patterns into **Data Indirection Graph** (DIG) • HW: prefetcher **traverses DIG** at runtime
Results	• **2.6× speedup**, **1.6× energy savings** over no prefetch at **negligible cost** • Notable speedup, savings over existing prefetchers

Overview

Paper Summary

- Background & Motivation
- Programming Model
- Hardware Design
- Results
- Conclusion

Critique & Discussion

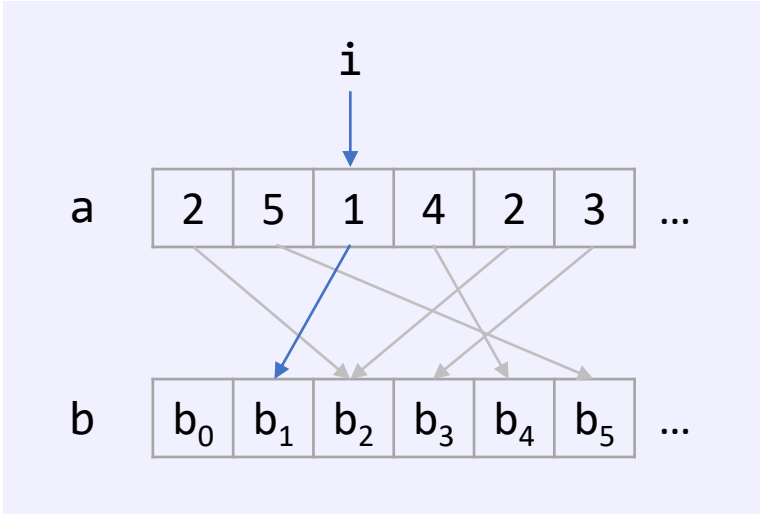
- Strengths
- Weaknesses
- Thoughts
- Discussion

Background & Motivation

Data-Indirect Irregular Workloads

- **Sparse irregular algorithms** ubiquitous
  - ML, Scientific computing, graph analytics, …
  - Usually involve **indirect memory accesses**
- Inefficient on CPUs
  - No temporal or spatial locality or correlation
  - ➢ Caching, common prefetchers ineffective
- Specialized prefetchers fall short
  - Linked structures: limited to single pointers
  - Irregular loads: only specific patterns and layouts
  - Software prefetch: static, untimely

```
for i in 0 .. N:
    c[i] = b[a[i]]
```



A Compressed Data Format: *CSR*

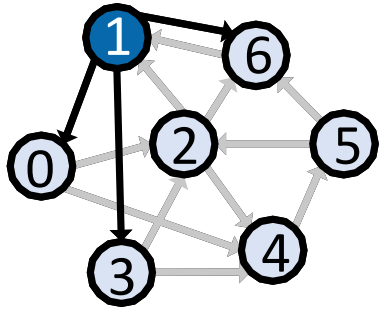
- Idea: **store only nonzeros** of a sparse matrix
  - `A_rowptrs`: indices delimiting **rows**
  - `A_colptrs`: **columns** of nonzeros
  - `A_vals`: **nonzeros**

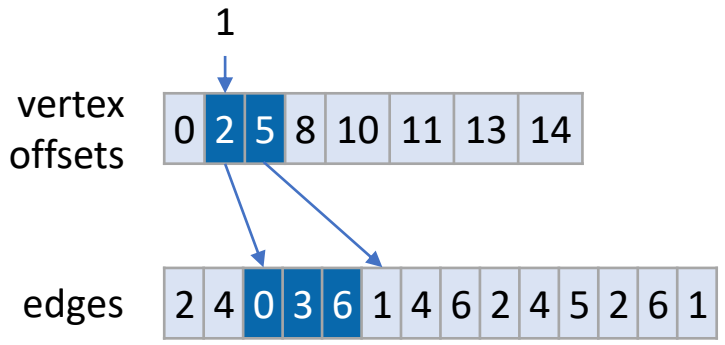
$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 7 & 9 \\ 3 & 0 & 4 & 1 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$

```
A_rowptrs[] = {0, 2, 5, 6}
A_colptrs[] = {2, 3, 0, 2, 3, 2}
  A_vals[] = {7, 9, 3, 4, 1, 7}
```

➢ Row contents accessed by **ranged indirection**

- **Various problems** encoded as sparse matrices
- Common: represent **graphs** as CSR adjacency matrices
  - No edge weights: value array redundant



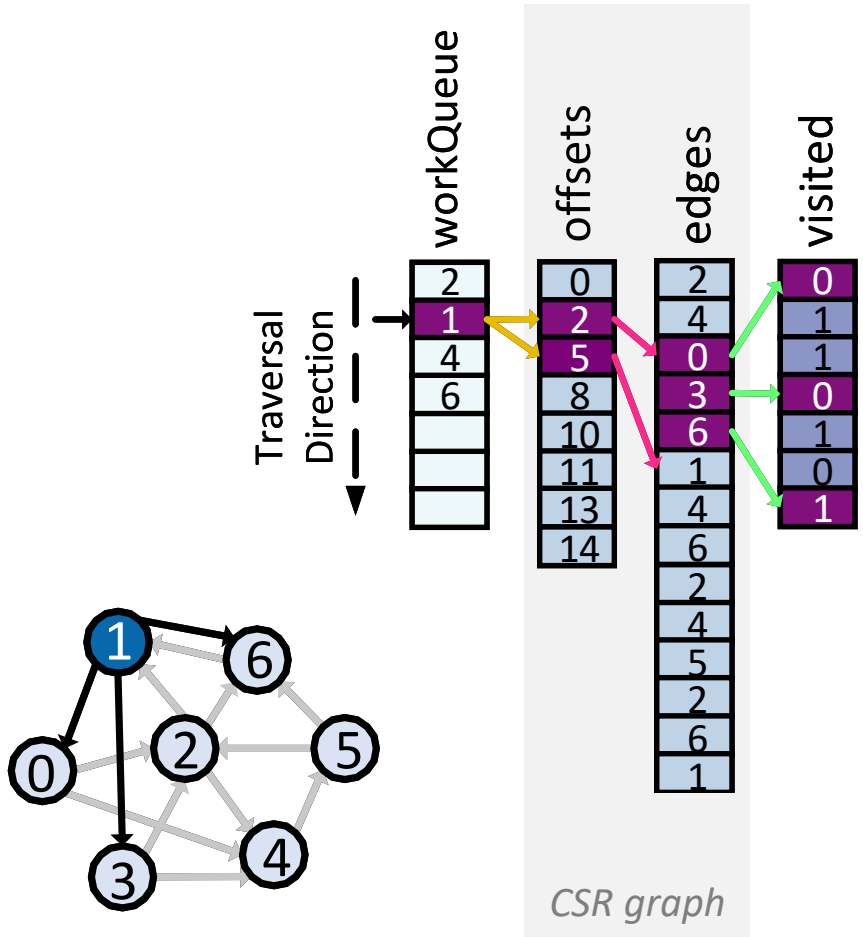


An Irregular Algorithm: *Breadth-First Search* (BFS)

- Traverse graph in order of **distance to start node**
  - Base of other graph algorithms
- Data: CSR graph + two helper arrays:
  - *workQueue*: found nodes to process next
  - *visited*: bitmap of seen nodes to avoid recursion

```
for node in workQueue:
  for i in range(offsets[node:node+1]):
    neigh = edges[i]
    if not visited[neigh]:
      workQueue.push(neigh)
      visited[neigh] = 1
```

➢ **3 levels** of indirection

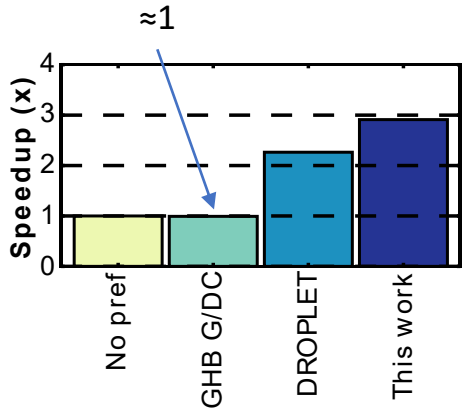


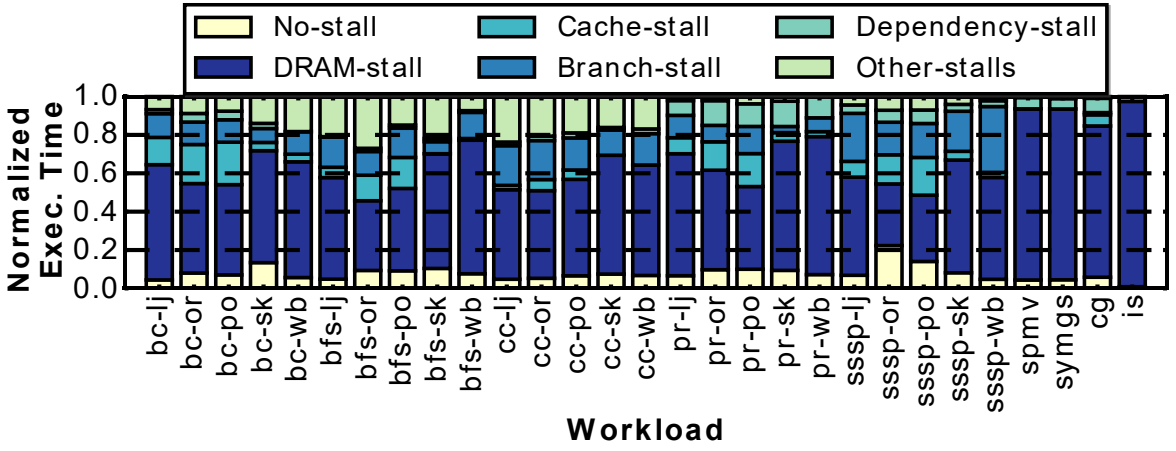
Bottlenecks in Current Systems

- **Data-dependent loads**: random patterns with low locality
  - ➢ Caches, common prefetchers ineffective
- **Load-dependent branches**: direction hard to predict
  - ➢ Expensive rollbacks

➢ Poor performance and efficiency
- **>50% stalled on DRAM**
- Significant branch stalls

➢ *Need an **effective, general** prefetcher for data-indirect access patterns*

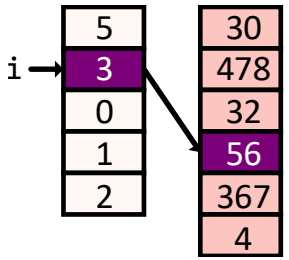




Programming Model

Indirection Primitives

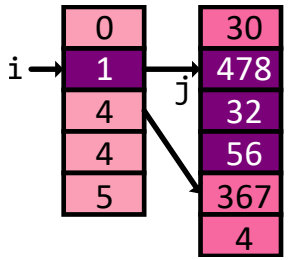
- Idea: two specific access patterns cover **wide range** of irregular workloads:




```
for i in 0 .. a_size:
  tmp += b[a[i]]
```

Single-valued indirection:
one index → one value

- *e.g. neighbors → visited map*




```
for i in 0 .. a_size:
  for j in a[i] .. a[i+1]:
    tmp += b[j]
```

Ranged indirection:
two index bounds → range of values

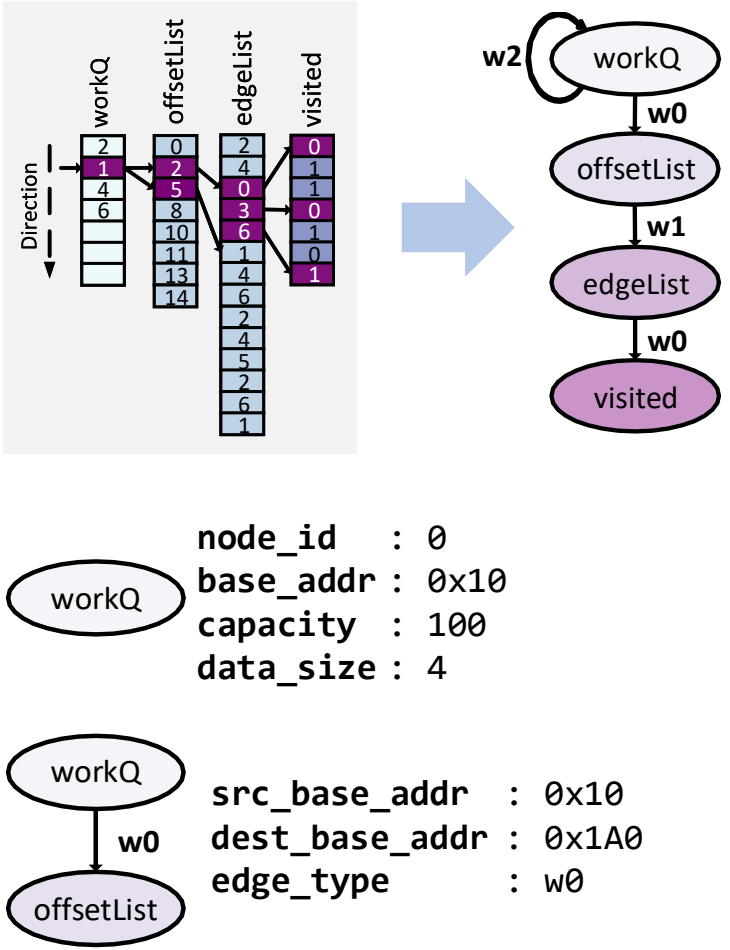
- *e.g. node → neighbors*

➢ **Combine and chain** to describe complex access patterns

- BFS: 1 *ranged* + 2 *single-valued* indirections

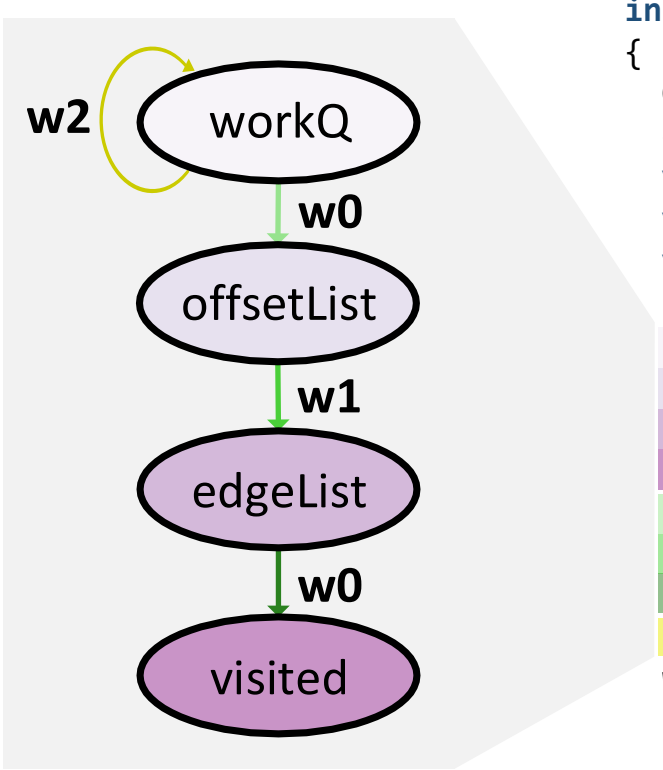
Data Indirection Graph (DIG)

- Encodes **data structures** and **indirections between them**
  - *Nodes*: data structure (array) metadata
  - *Edges*: indirections between nodes
- Three edge types
  - *w0: single-valued* indirection
  - *w1: ranged* indirection
  - w2: *trigger* edge; initiates prefetch sequence
- Trigger edges store **sequence initialization parameters**
- Captured **before runtime** by programmer or compiler
  - Included in binary



DIG Construction by Programmer

- Programmer adds **API calls** writing graph components to **prefetcher memory**:




```
int BFS(FILE* inputGraph, vtxID source)
{
  Graph g = readGraph(inputGraph);
  queue<vtxID> workQueue(g.numNodes());
  vtxID** offsetList = (vtxID**) malloc(g.numNodes()+1);
  vtxID*  edgeList   = (vtxID*)  malloc(g.numEdges());
  vtxID*  visited    = (vtxID*)  malloc(g.numNodes());
  populateDataStructures(g, offsetList, edgeList, visited);
  registerNode(&workQueue,  g.numNodes(),   4,  0);
  registerNode(offsetList,  g.numNodes()+1, 4,  1);
  registerNode(edgeList,    g.numEdges(),   4,  2);
  registerNode(visited,     g.numNodes(),   4,  3);
  registerTravEdge(&workQueue, offsetList,  w0);
  registerTravEdge(offsetList, edgeList,    w1);
  registerTravEdge(edgeList,   visited,     w0);
  registerTrigEdge(&workQueue, w2);
  workQueue.enqueue(source);
  […]
```

Allocate data structures

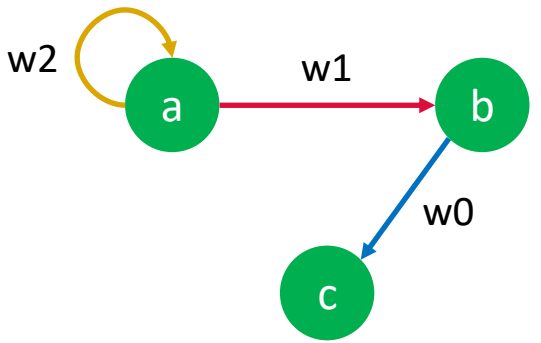
Write DIG to prefetcher

Irregular Algorithm

DIG Inference by Compiler

- Inserts **same API calls** at IR level
  - ➢ Can be combined with manual annotation
- Single-read LLVM pass infers:
  1. *Nodes* from **allocator calls**
  2. *Single-value* edges from **dependent loads** found by backtracking
  3. *Ranged edges* from loads in loops with **adjacent bounds**
  4. *Trigger edges* on nodes with **no inbound edges**
- **Negligible overhead** on compile time
  - Data dependencies resolved by prefetcher

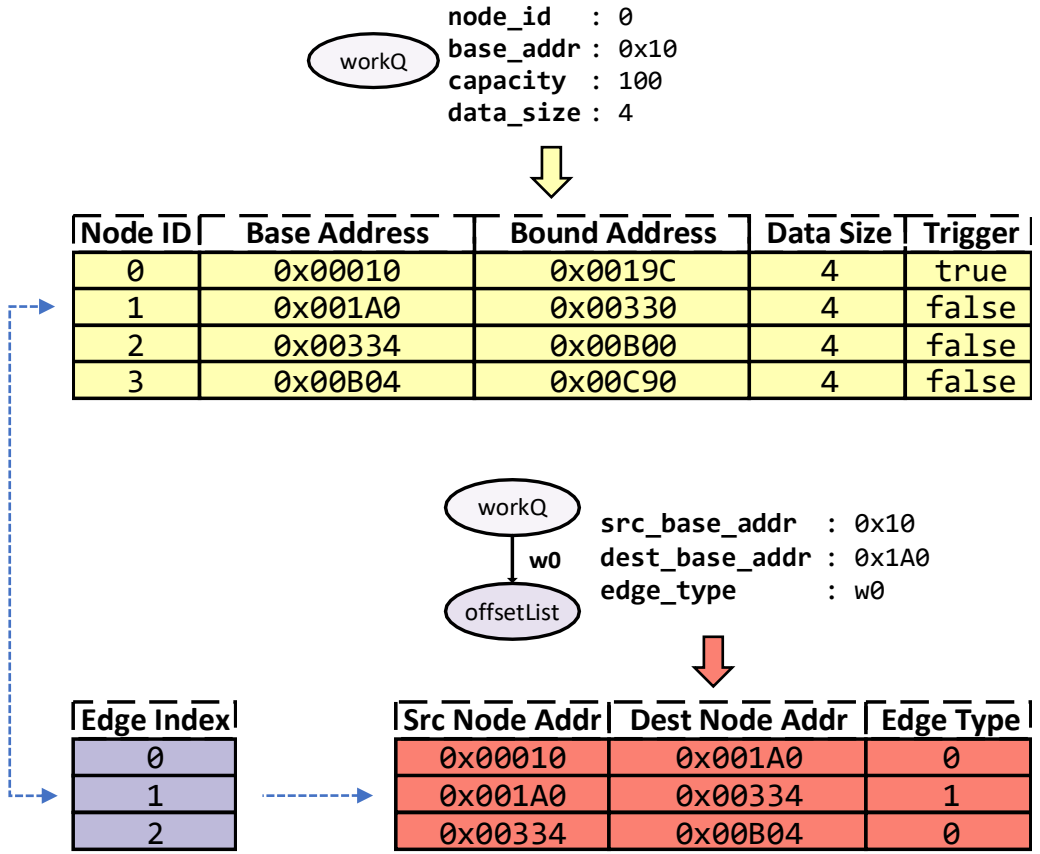
```
int tmp;
int *a = malloc(a_size);
int *b = malloc(b_size);
int *c = malloc(c_size);

for (i=0; i<a_elems; ++i)
  for (j=a[i]; j<a[i+1]; ++j)
    tmp += c[b[j]];
```



Hardware Design

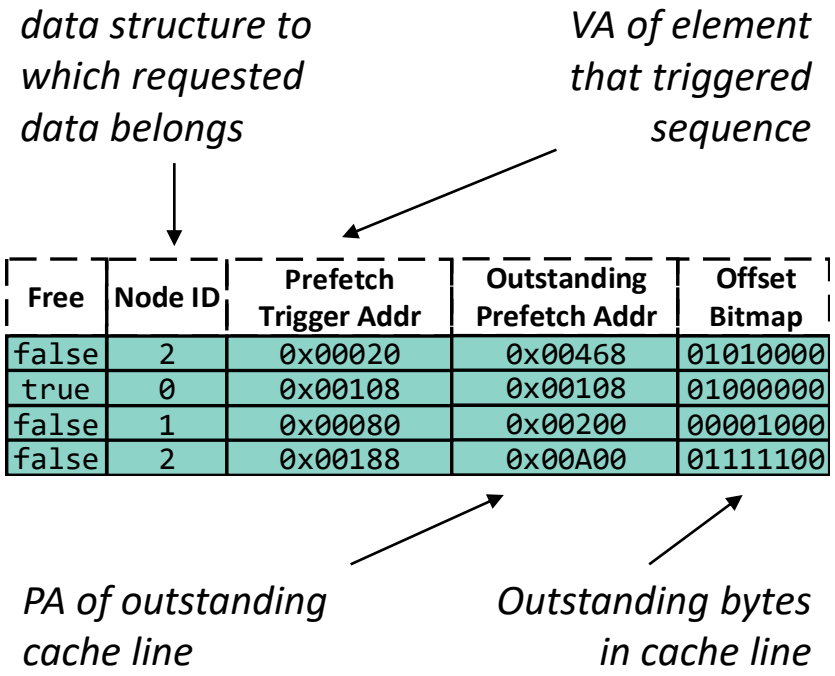
DIG Storage

- Prefetcher stores DIG in **dedicated SRAM**
- **Three tables** written by API calls
  - Node table
  - Edge table
  - Edge index table
- *Edge index table* keeps **source nodes of edges**
  - Used find outgoing edges for nodes
- Uses *virtual* addresses: set at compile time



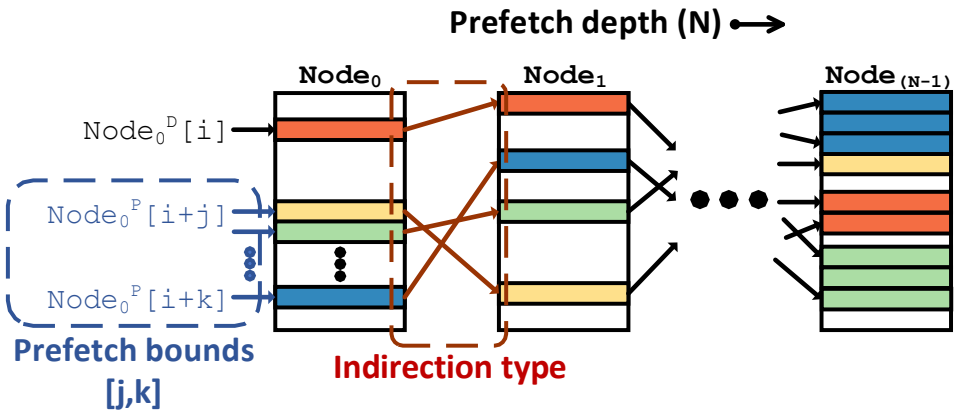
Prefetch Status Handling Registers (PFHR)

- Need to track **multiple outstanding** prefetches
  - Prefetch sequences can span 4+ structures
  - Blocking may waste opportunities
- Track prefetches in **PFHR File**
  - Like MSHRs in non-blocking caches
- Allocated on prefetch sequence trigger
- Updated or freed on prefetch cache fills



Prefetch: Sequence Initialization

- Launched on **core load on trigger node**
  - Window of sequences launched at once
- **Trigger edge** encodes initialization parameters
  - `[j,k]`: Lookahead *distance* and *bound*
  - *Direction*: ascending or descending addresses
- Heuristic: decrease j as *prefetch depth* increases
- Feedback loop: drop sequence when **core requests trigger element**
  - ➢ *Timely*: prefetch always ahead of core
  - ➢ *Efficient*: maximizes latency hiding



PFHRs

Free	Node ID	Prefetch Trigger Addr	Outstanding Prefetch Addr	Offset Bitmap
false	2	0x00020	0x00468	01010000
true	0	0x00108	0x00108	01000000
false	1	0x00080	0x00200	00001000
false	2	0x00188	0x00A00	01111100

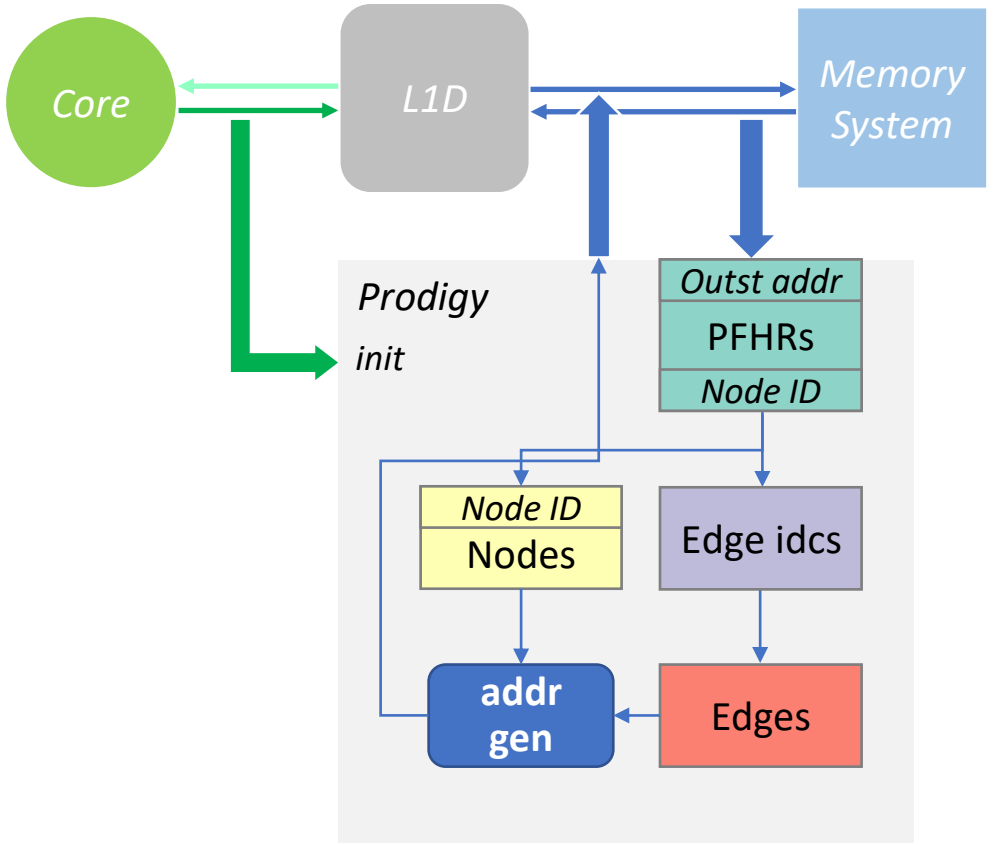
Prefetch: Sequence Advance

- On cache refill: check, update PFHRs

➢ If response to prefetch: read DIG

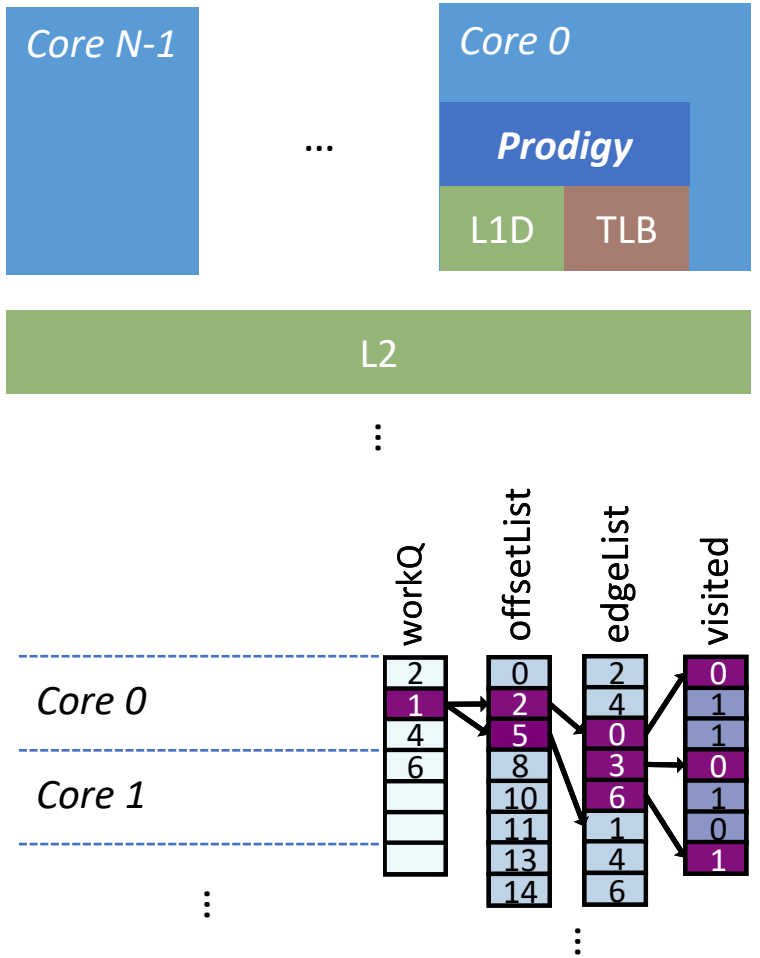
1. Look up *source node* of prefetch
2. Find outgoing edge(s) through index table
3. **Compute next prefetch address** (if any)
4. Allocate new PFHR and request

- Sequence ends once source node traversed
- New sequences initiated when core demands data in trigger nodes



System Integration

- One **private instance per core**
  - Prefetches into L1D cache
  - Reuses D-TLB for address translation
- Snoops cache bus to observe refills
  - No additional ports on cache
- Supports *contiguous* **partitioning of trigger node data** among cores (e.g. OMP)
- Some open problems
  - Coherency contentions at partition edges
  - Costly context switches in multiple threads
  - No prefetch throttling yet



Results

Evaluation Setup and Workloads

- Simulation configuration
  - **Sniper** x86 sim: **8 OoO cores,** built-in energy model
  - **32K/256K/2M** caches, CACTI access times
  - DRAM: **120 cyc. access**, controller queuing
  - Optimum for evaluated problems: **16 PFHRs**
- Algorithms: from benchmark suits
  - **GAPBS: graph algorithms** like BFS, PR, …
  - HPCG: SpMV, Symm. Gauss-Seidel soother
  - NAS: conj. gradient, integer sort
- Data: **real-world graphs** + suit generators

Component	Modeled Parameters
Core	8-OoO cores, 4-wide issue, 128-entry ROB, load/store queue size = 48/32 entries, 2.66GHz frequency
Cache Hierarchy	Three-level inclusive hierarchy, write-back caches, MESI coherence protocol, 64B cache line, LRU replacement
L1 I/D Cache	32KB/core private, 4-way set-associative, data/tag access latency = 2/1 cycles
L2 Cache	256KB/core private, 8-way set-associative, data/tag access latency = 4/1 cycles
L3 Cache	2MB/core slice shared, 16-way set-associative, data/tag access latency = 27/8 cycles
Main Memory	DDR3 DRAM, access latency = 120 cycles, memory controller queuing latency modeled

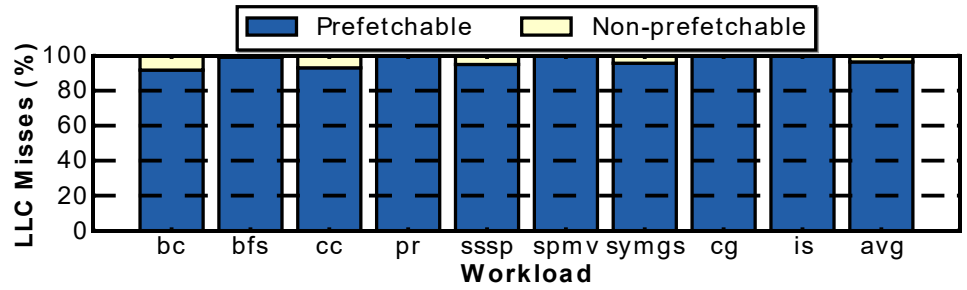
Graph	Number of vertices	Number of edges	Size (in MB)	× LLC capacity
pokec (po)	1.6M	30.6M	132.0	16.5
livejournal (lj)	4.8M	69.0M	300.0	37.5
orkut (or)	3.1M	117.2M	485.2	60.6
sk-2005 (sk)	50.6M	1930.3M	7749.6	968.7
webbase-2001 (wb)	118.1M	1019.9M	4791.6	598.9

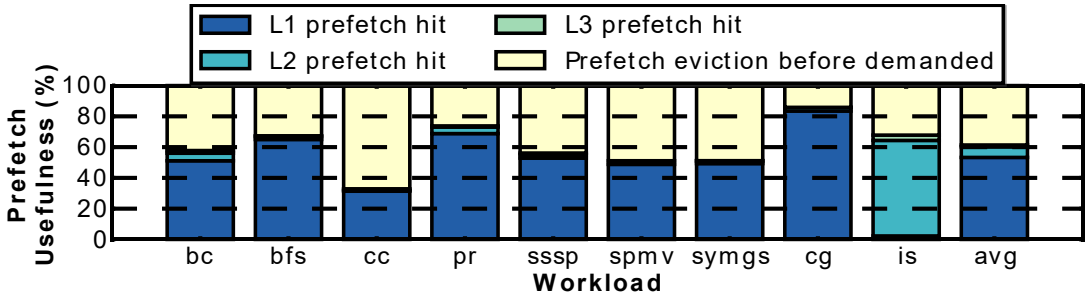
Prefetch Potential and Usefulness

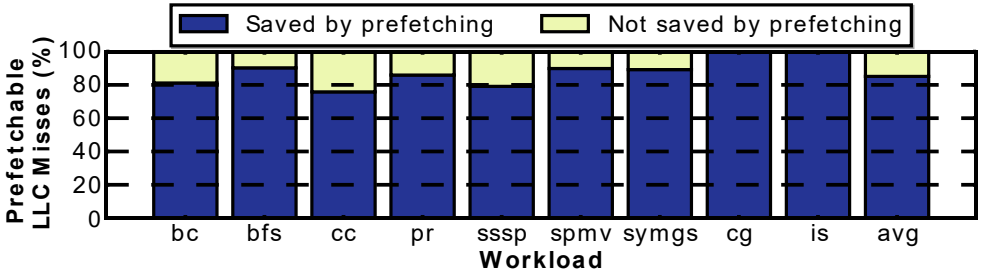
- No prefetch: measure LLC misses DIG covers
  - Upper bound on DRAM stall reduction
  - ➢ avg **96% DIG coverage**
- Notable variability in accuracy: 33 – 86%
  - ➢ avg **63% of prefetches demanded**
  - Hits predominantly in L1D
  - Most misses attributed to **timeliness**

➢ Avg. **85% of prefetchable LLC misses** converted into hits

➢ Ranged indirection *essential*: avg **55%** of prefetches





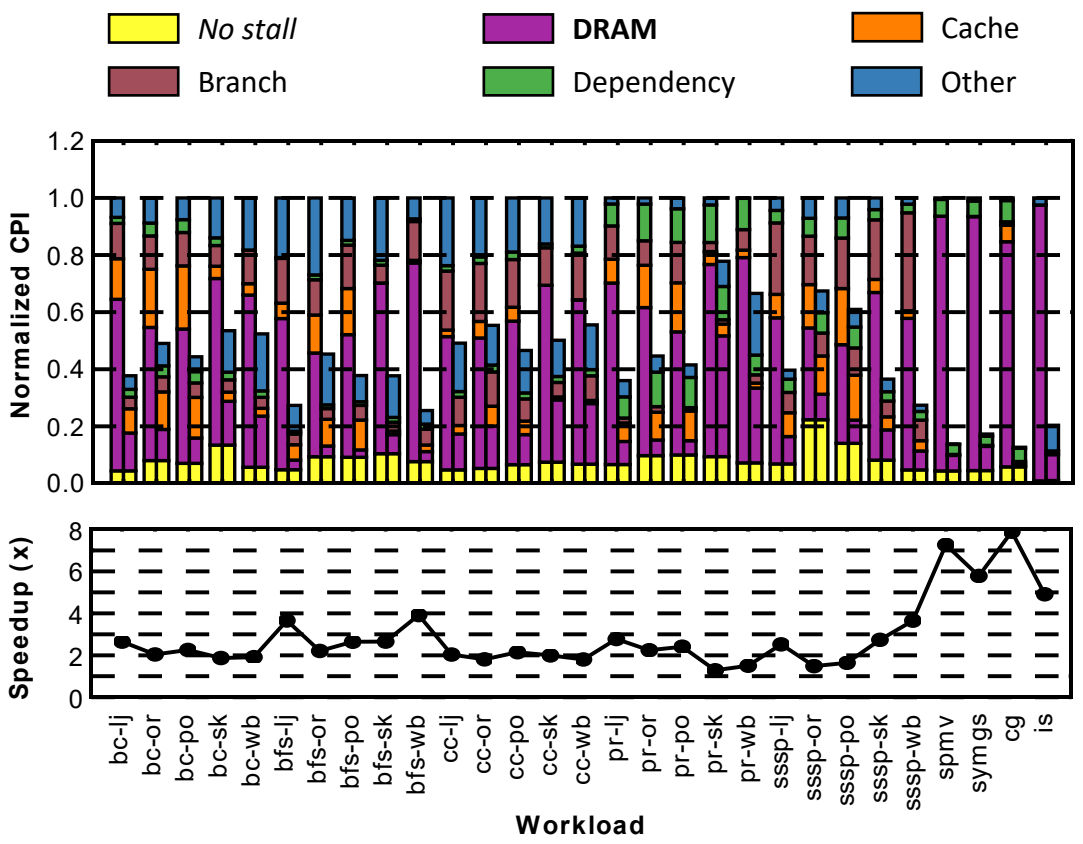


Performance vs No Prefetch Baseline

- Baseline: **DRAM stalls dominate: 84%**
- DRAM stalls down by avg **80%**
  - Slight increase in cache stalls: more traffic
- Branch stalls down by avg **65%**
  - Most notable in workloads with branch-dependent loads

➢ **Speedup of 2.6×** overall

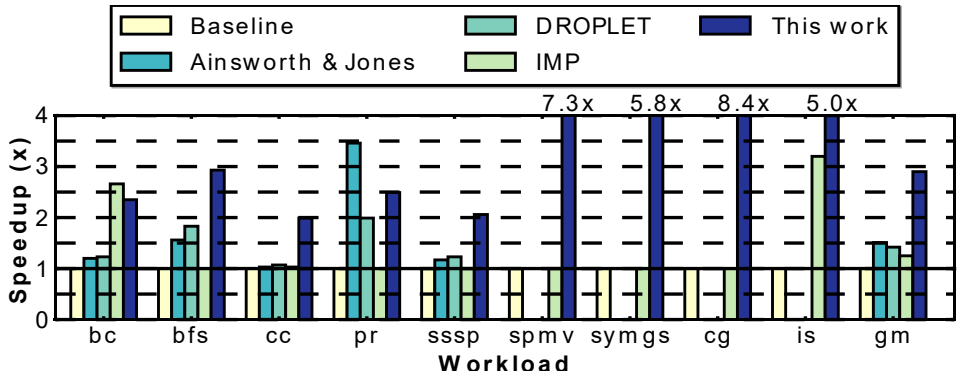
- Format-robust: speedups on CSR+CSC workloads similar to CSR-only tasks



Performance vs Existing Prefetchers

- Indirection SW PF on PageRank: **1.08× vs 2×** speedup
- Common HW PF (GHB-based G/DC): **2.6×** speedup
- Specialized HW PFs: could not reproduce results
  → compare best reported *and* measured
- Ainsworth & Jones: **1.2×** or **1.5×**
  - Less timely, less general: only BFS-like patterns
- DROPLET: **1.2×** or **1.6×**
  - Only single-valued indirection, limited triggering
- IMP: **2.5×** or **2.3×**
  - Only 2 levels of single-valued indirection

Measured by authors (A&J, DROPLET only support graph loads)

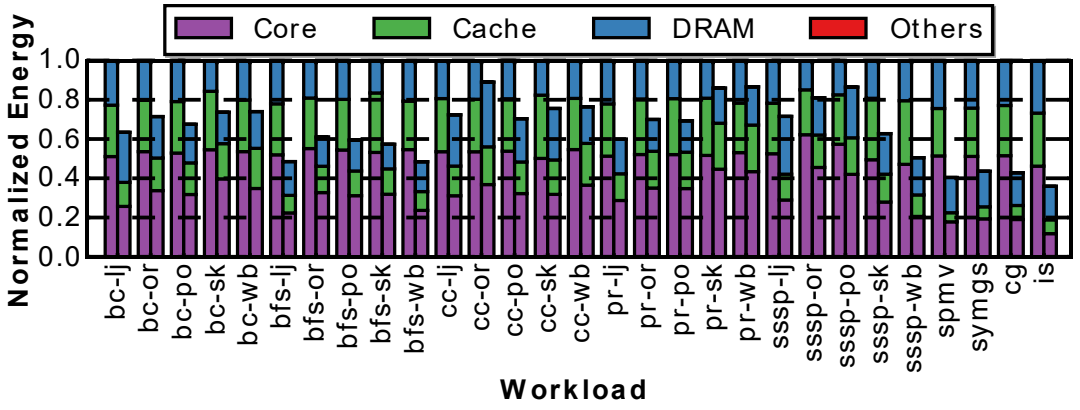


Reported in prior work

Common algorithms	Prior work		Prodigy
bc,bfs,bc,pr	Ainsworth & Jones [6]	2.4×	2.8×
bc,bfs,bc,pr,sssp	DROPLET [15]	1.9×	2.9×
bfs,pr,spmv,symgs	IMP [99]	1.8×	4.6×

Energy and Overhead

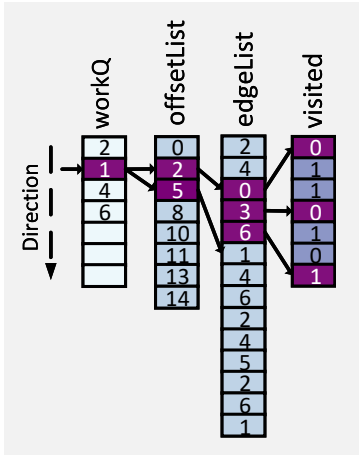
- Energy reduced in **all categories**: avg **1.6×**
  - Faster → less static energy
  - Less instructions, accesses, mispredictions
- HW Overhead: mainly storage (DIG, PFHRs)
  - ~ **0.8 KB** or **0.004%** of CPU die
  - 1.4 – 40× less than other solutions
- SW Overhead: negligible
  - Tiny binary size increase (API calls)
  - ~1 s added compile time

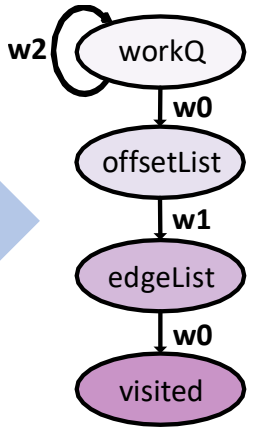


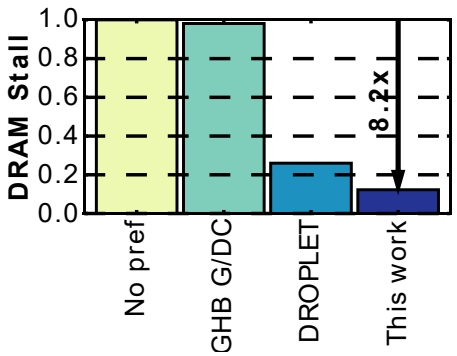
Conclusion

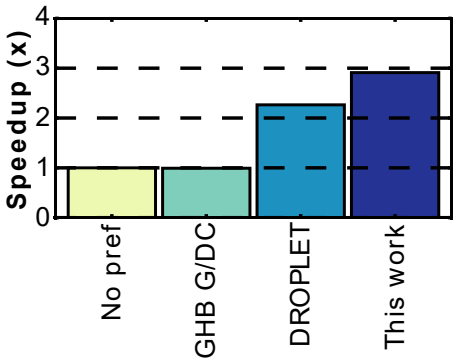
Conclusion

- *Prodigy* is a **HW/SW codesign** to prefetch *data-indirect irregular workloads*
- **DIG** encodes data structure *layout* and *traversal*
  - Composes *single-valued* and *ranged* indirection
  - Added *ahead-of-time* by programmer *or* compiler
- **Low-cost HW prefetcher** combines static DIG with dynamic runtime information
  - ➢ **2.6× speedup** over baseline
  - ➢ **1.6 × energy savings**
  - ➢ Negligible HW, SW overheads









Strengths and Weaknesses

Problems Identified by Authors

- **Suboptimal multithreading** for threads sharing core
  - DIG, PFHRs must be swapped
- **No prefetch throttling** mechanism (yet)
  - May further mitigate cache pollution
- Some algorithms need **additional data in indirection**
  - May cause cache thrashing in these cases
- DIG, PFHR parameters **optimized for shown workloads**

Strengths

- **Well-organized** and **well-explained** paper
  - Entire HW/SW stack exemplified using one problem (BFS)
- **General**, yet **performant** and **goal-oriented** solution
- **Extensive**, well explained **software integration**
  - End user API and LLVM passes for DIG construction
  - Complete, reproducible description of both
- Mindful use of **hardware resources**
  - Careful allocation of both memory and logic

Weaknesses

[1] S. Ainsworth and T. M. Jones: “Graph Prefetching Using Data Structure Knowledge”, ICS 2016.
[2] V. Dadu, J. Weng, S. Liu, and T. Nowatzki: “Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms”, MICRO ’52.
[3] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips: “SQRL: hardware accelerator for collecting software data structures”, PACT 2014
[4] A. Roth and G. S. Sohi: “Effective jump-pointer prefetching for linked data structures”, SIGARCH Comput. Archit. News 27, 2 (May 1999), 111–121.

- **Limited Novelty**: similar indirection, workload prefetch approaches in prior work [1-4..]
- **Hardware description** *vague* at best → not useful beyond high-level simulation
  - How does the prefetch “FSM” work?
  - How is position in intermediate nodes kept track of?
  - (How) can we defer traversal on multi-edge nodes? What if we run out of PFHRs?
- **Evaluation methodology** has serious flaws
  - SRAMs are not content-addressable: needs standard cell memory
  - HW area estimate seems *very* off: “FSM” clearly dominates 800B of ~~SRAM~~ SCM
  - Existing works *should* be reproducible, *no evidence* for result hypotheses
- **Timing** in core domain *critical*, but not considered
  - ➢ (Likely) **poor prefetch BW**: many steps to request single line

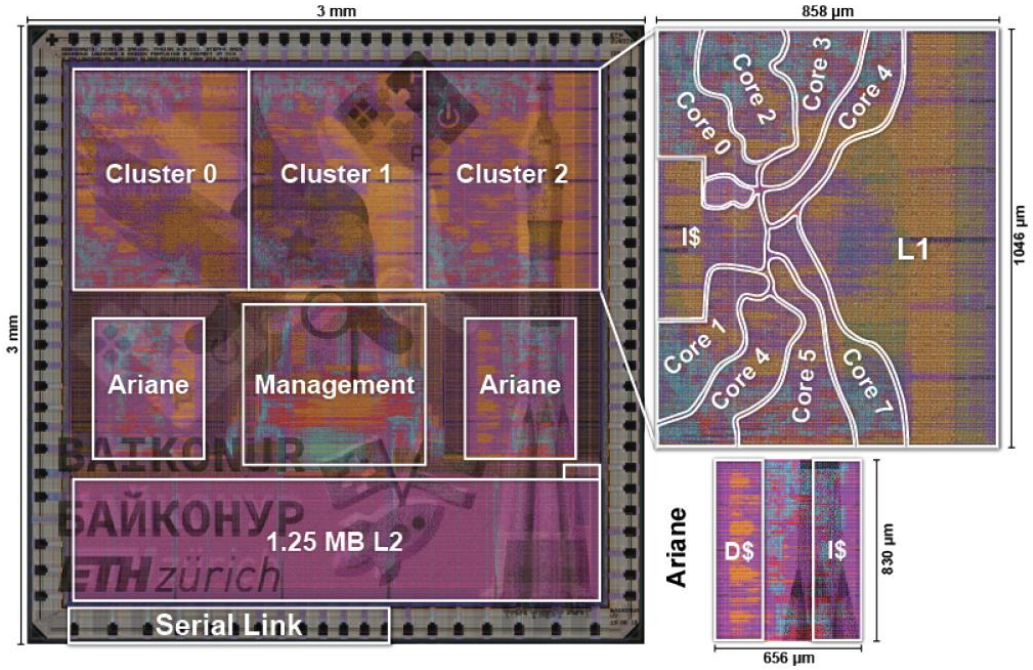
Thoughts and Ideas

Implement in RTL and Silicon

- Vague HW, timing info likely due to **heavy abstraction** → *go deeper*
- Implement at **Register-transfer Level**
  - Cycle-accurate simulation
  - 100% reproducible and implementable hardware description
- Implement in recent **silicon technology**
  - 100% accurate timing and area figures
  - Proven physical feasibility (P&R)

➢ Use implementation results to **optimize HW**

➢ Use high-level simulation with **proven characteristics** for performance evaluation



A heterogeneous manycore platform test chip implemented in GF22FDX [1]

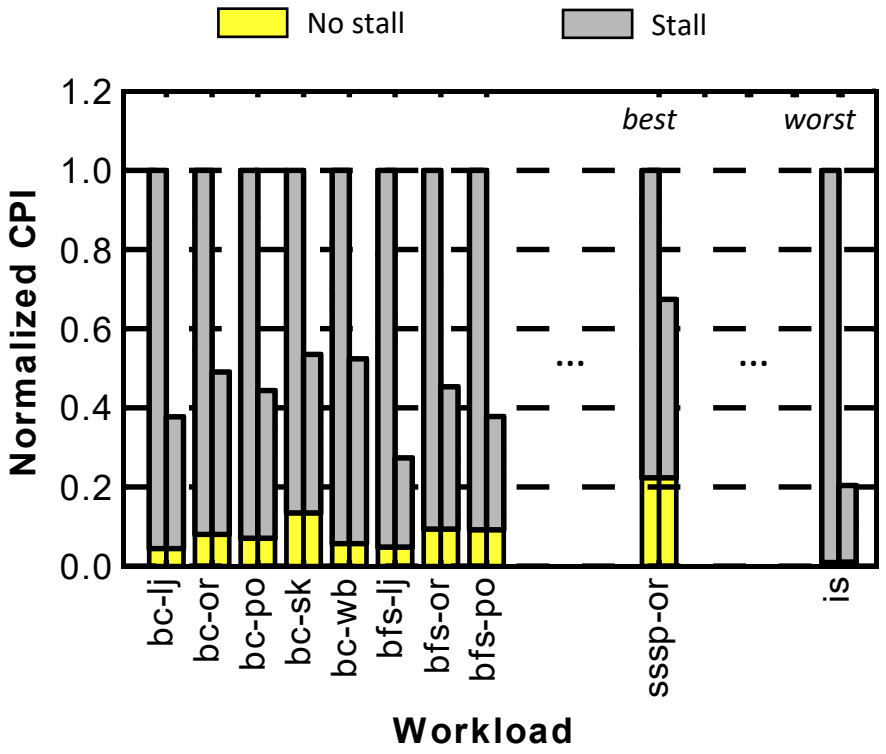
[1] F. Zaruba, F. Schuiki and L. Benini, "Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing," in *IEEE Micro*, vol. 41, no. 2, pp. 36-42, 1 March-April 2021, doi: 10.1109/MM.2020.3045564.

Couple to Core for Better Performance

- *Absolute* **IPC still poor**
  - Much of no-stall likely **bookkeeping**
- Core **duplicates all address calculation** steps done by Prodigy, but in SW → *slow*
- Compiler is fully aware of Prodigy
  - Prefetch sequence (DIG) known ahead-of-time

➢ Implement direct data streams into core
  - Prodigy directly provides prefetched data
  - Add ISA instruction to pop / push streams
  - ➢ Compiler coordinates core and Prodigy to **eliminate bookkeeping / stalls** and **maximize IPC**

➢ *Increases performance while saving energy*



Generalize Indirection Function

- Some algorithms need *different* indirect address transforms
  - May need additional data
- Plenty of potential to extend Prodigy "FSM"
  - Won't have much impact at this scale

➢ Generalize indirection functionality

- **Analyze workloads** to see which might pay off
- Implement **address transforms in HW**
  - ➢ Better prefetch coverage
  - ➢ Higher performance

```
t = a[i]
x_ptr = b + t
x = *x_ptr
```

```
for t in a[i]..a[i+1]:
  x_ptr = b + t
  x = *x_ptr
```

```
x_ptr = indir(b, t, l1_data)
```

Discussion

How could other components (DRAM schedulers, coalescers, cache eviction, …) benefit from known demand sequences?

Can we leverage ahead-of-time analysis to prefetch other access patterns? What about *arbitrary* regular patterns?

We can technically reprogram Prodigy at any time during runtime.
When would this make sense? What can we gain from it?

How can we adapt Prodigy to better integrate with multiple threads per core and the OS?

How does Prodigy affect timing channel attacks? Does it increase or decrease attack surface and bandwidth, and why?