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

Citation for published version:
Talati, N, May, K, Behroozi, A, Yang, Y, Kaszyk, K, Vasiladiotis, C, Verma, T, Li, L, Nguyen, B, Sun, J, Morton, JM, Ahmadi, A, Austin, T, O'Boyle, MFP, Mahlke, S, Mudge, T & Dreslinski, R 2021, Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* Institute of Electrical and Electronics Engineers (IEEE), pp. 654 - 667, The 27th IEEE International Symposium on High-Performance Computer Architecture, Seoul, Korea, Republic of, 27/02/21. https://doi.org/10.1109/HPCA51647.2021.00061

Digital Object Identifier (DOI):
10.1109/HPCA51647.2021.00061

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

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[*], Jiawen Sun[‡], John Magnus Morton[‡], Agreen Ahmadi[*], Todd Austin[*], Michael O'Boyle[‡], Scott Mahlke[*], Trevor Mudge[*], Ronald Drelinski[*]

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

Email: talatin@umich.edu

Abstract—**Irregular workloads are typically bottlenecked by the memory system. These workloads often use sparse data representations, *e.g.*, compressed sparse row/column (CSR/CSC), to conserve space at the cost of complicated, irregular traversals. Such traversals access large volumes of data and offer little locality for caches and conventional prefetchers to exploit.**

This paper presents Prodigy, a low-cost hardware-software co-design solution for intelligent prefetching to improve the memory latency of *several important irregular workloads*. Prodigy targets irregular workloads including graph analytics, sparse linear algebra, and fluid mechanics that exhibit *two specific* types of data-dependent memory access patterns. Prodigy adopts a "best of both worlds" approach by using static program information from software, and dynamic run-time information from hardware. The core of the system is the *Data Indirection Graph (DIG)*—a proposed compact representation used to express program semantics such as the layout and memory access patterns of key data structures. The DIG representation is agnostic to a particular data structure format and is demonstrated to work with several sparse formats including CSR and CSC. Program semantics are automatically captured with a compiler pass, encoded as a DIG, and inserted into the application binary. The DIG is then used to program a low-cost hardware prefetcher to fetch data according to an irregular algorithm's data structure traversal pattern. We equip the prefetcher with a flexible prefetching algorithm that maintains timeliness by dynamically adapting its prefetch distance to an application's execution pace.

We evaluate the performance, energy consumption, and transistor cost of Prodigy using a variety of algorithms from the GAP, HPCG, and NAS benchmark suites. We compare the performance of Prodigy against a non-prefetching baseline as well as state-of-the-art prefetchers. We show that by using just 0.8KB of storage, Prodigy outperforms a non-prefetching baseline by 2.6× and saves energy by 1.6×, on average. Prodigy also outperforms modern data prefetchers by 1.5–2.3×.

Index Terms—**DRAM stalls, irregular workloads, graph processing, hardware-software co-design, programming model, programmer annotations, compiler, and hardware prefetching.**

I. Introduction

Sparse irregular algorithms are widely deployed in several application domains including social networks [65], [76], online navigation systems [39], machine learning [42], and genomics [9], [34]. Despite their prevalence, current hardware-software implementations on the CPUs offer sub-optimal performance that can be further improved. This is due to the irregular nature of their memory access patterns over large data sets, which are too big to fit in the on-chip caches, leading to several costly DRAM accesses. Therefore, traditional techniques to improve memory latency—out-of-order processing, on-chip caching, and spatial/address-correlating data prefetching [13], [49], [52], [66], [95], are inadequate.

There is a class of prefetchers [11], [23], [26], [31], [43], [50], [79], [98] which focuses on linked data structure traversals using pointers. In graph algorithms, for example, these prefetchers fall short for two reasons. First, graph algorithms often use compressed data structures with indices instead of pointers. Second, graph traversals access a series of elements in a data structure within a range determined by another data structure. These prefetchers are not designed to accommodate such complex indirection patterns.

Recently, several prefetching solutions have been proposed targeting irregular workloads. Hardware prefetchers rely on capturing memory access patterns using explicit programmer support [5], [6], learning techniques [77], and intelligent hardware structures [99]. Limitations of these approaches include their limited applicability to a subset of data structures and indirect memory access patterns [6], [15], [99] or high complexity and hardware cost to support generalization [5], [77]. While software prefetching [7] can exploit static semantic view of algorithms, it lacks dynamic run-time information and struggles to maintain prefetch timeliness.

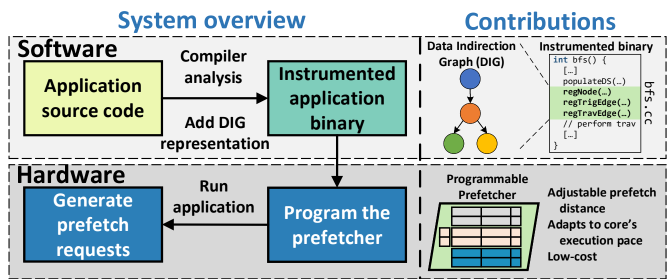


Figure 1. Overview of our design and contributions. Prodigy software efficiently communicates key data structures and algorithmic traversal patterns, encoded in the proposed compact representation called the Data Indirection Graph (DIG), to the hardware for informed prefetching.

In this paper, we propose a hardware-software co-design for improving the memory latency of several important irregular workloads exhibiting arbitrary combinations of two specific memory access patterns. The **goals** of this design are threefold: (a) automatically prefetch *all the key data structures* expressing irregular memory accesses, (b) exploit dynamic run-time information for *prefetch timeliness*, and (c) realize a *low-cost* hardware prefetching mechanism. To this end, we propose a compact representation called the **Data Indirection Graph (DIG)** to communicate workload attributes from software to the hardware. The DIG representation efficiently encodes the program semantics, *i.e.,* the layout and access patterns of key data structures, in a weighted directed graph structure. Fig. 1 presents the overview of our proposal. The relevant program semantics are extracted through a compile-time analysis, and this information is then encoded in terms of the DIG representation and inserted in the application binary. During run-time, the DIG is used to program the hardware prefetcher making it cognizant of the indirect memory access patterns of the workload so it can cater its prefetches accordingly.

Prodigy is a pattern-specific solution that targets *two* types of data-dependent indirect memory accesses, which we call **single-valued indirection** and **ranged indirection**. Single-valued indirection uses data from one data structure to index into another data structure; it is commonly used to find vertex properties in graph algorithms. Ranged indirection uses two values from one data structure as base and bounds to index into a series of elements in another data structure; this technique is commonly used to find neighbors of a vertex in graph algorithms. Based on this observation, we propose a compact DIG representation that abstracts this information in terms of a weighted directed graph (*unrelated to the input graph data set*). The nodes of the DIG represent the memory layout information of the data structures, *i.e.,* address bounds and data sizes of arrays. Weighted edges represent the type of indirection between data structures. We present a **compiler pass** to *automatically extract* this information and *instrument the binary* with API calls to generate the DIG at a negligible cost. Our results show that the DIG is agnostic to any particular data representation; it works well for various sparse data formats including compressed sparse row/column (CSR/CSC).

We design a **low-cost hardware prefetcher** that can be *programmed using the DIG representation* communicated from software. We store the DIG in prefetcher-local memory to make informed prefetching choices. The prefetcher reacts to demand accesses and prefetch fills[1] to the L1D cache and issues non-binding prefetches (*i.e.,* prefetched data placed in the L1D cache) based on an irregular algorithm's memory traversal pattern. To track the progress of the prefetch sequences and enable non-blocking prefetching, we introduce the *PreFetch status Handling Register (PFHR) file*. Additionally, we present an adaptive prefetching algorithm that selectively drops prefetch sequences when the core catches up to the prefetcher. We name our system **ProDIGy** as it uses software analysis coupled with hardware prefetcher using the program's DIG representation.

[1]We define a prefetch fill as the cache line brought into the cache as a response to a prefetch request.

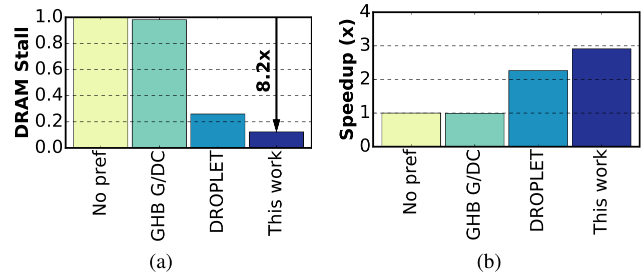


Figure 2. Reduction in ((a)) memory stalls and ((b)) speedup of different approaches normalized to a non-prefetching baseline for the PageRank algorithm on the `livejournal` data set.

We evaluate the benefits of Prodigy in terms of performance, energy consumption, and hardware overhead. For evaluation, we use five graph algorithms from the GAP benchmark suite [16] with five real-world large-scale data sets from [27], [59], two sparse linear algebra algorithms from the HPCG benchmark suite [29], and two computational fluid dynamics algorithms from the NAS parallel benchmark suite [12]. We compare our design with a non-prefetching baseline, GHB-based global/delta correlation (G/DC) data prefetcher, and state-of-the-art prefetchers, *i.e.,* IMP [99], Ainsworth and Jones' [5], [6], DROPLET [15], and software prefetching [8].

Fig. 2 presents a highlight of performance benefits of Prodigy on the PageRank algorithm running on the `livejournal` data set [59]. Compared to a non-prefetching baseline, Prodigy reduces the DRAM stalls by 8.2× resulting in a significant end-to-end speedup of 2.9× compared to the marginal speedups observed using a traditional G/DC prefetcher that cannot predict irregular memory access patterns and DROPLET [15] which only prefetches a subset of data structures. Section VI presents further comparisons with [5]–[7], [99]. Across a complete set of 29 workloads, we show a significant **average speedup of 2.6**× and **energy savings of 1.6**× compared to a non-prefetching baseline. Using our evaluation framework, we further show that Prodigy outperforms IMP [99], Ainsworth and Jones' prefetcher [6], and DROPLET [15] by 2.3×, 1.5×, and 1.6×, respectively. The compact DIG representation allows Prodigy to achieve high speedups at a mere *0.8KB of hardware storage overhead*. In comparison, by simply scaling the non-prefetching baseline to use more cores to maximize the memory bandwidth and achieve similar throughout would require 5× more cores.

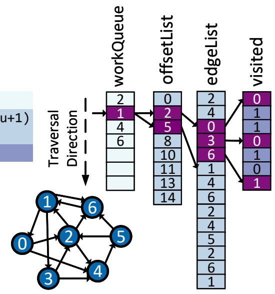
Prodigy is a specialized approach for critical memory latency-bound applications. When a processor is not running these applications, Prodigy will be turned off. In the age of dark silicon [35], state-of-the-art hardware frequently employs specialized accelerators for key applications. With Prodigy's low-cost design (0.8KB storage requirement), it is a modest price to pay for the efficiency it provides.

In summary, we make the following contributions:

- A compact representation of data traversal patterns, *called a DIG (Data Indirection Graph)*, for irregular workloads

```
workQueue.enqueue(source)
while workQueue not empty:
  #pragma omp parallel
  for all u : workQueue()
    for w : offsetList(u) to offsetList(u+1)
      for all v : edgeList(w)
        if !v.visited()
          compare_and_swap
              (v.visited(), 0, 1)
          workQueue.enqueue(v)
        endif
      endfor
    endfor
  endfor
endwhile
```

(a) Parallel implementation of BFS algorithm



(b) Data structures and algorithmic traversal pattern for a toy graph example

Figure 3. BFS algorithm: (a) pseudo-code for a parallel implementation of BFS, and (b) a toy example of BFS traversal on a graph stored in a compressed sparse row (CSR) format.

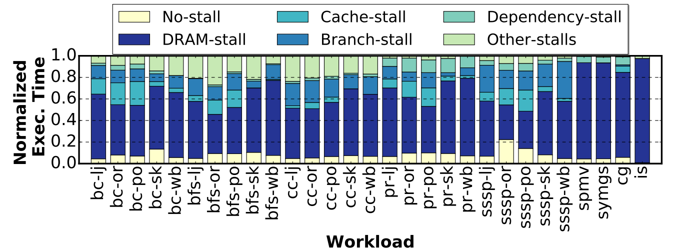


Figure 4. Normalized execution time of irregular workloads, without prefetching, broken down into: no-stall, and stalls due to DRAM, cache, branch mispredictions, data dependencies, and others. *The goal of this work is to reduce the DRAM stalls (dark blue portion of the bar).*

with any combination of two specific data-dependent memory access patterns.

- A novel programming model and associated compiler pass that analyzes the program, extracts key data structures and algorithmic traversal patterns, and generates instrumented code to create the DIG representation.
- A low-cost hardware prefetching design that uses this representation to prefetch data based on an irregular algorithm's memory traversal pattern in a timely manner.
- A resulting hardware-software co-designed system with an average speedup of 1.7× compared to the state-of-the-art prefetchers; average speedup and energy savings of 2.6× and 1.6× compared to a non-prefetching baseline at a negligible storage requirement of 0.8KB.

II. Background and Motivation

In this section, we use breadth-first search (BFS) graph algorithm as a representative irregular algorithm and discuss its data structures and algorithmic traversal pattern that leads to sub-optimal performance on CPUs.

Compressed sparse row (CSR) is a space-efficient technique for representing a sparse matrix, and it is commonly used to represent in-memory graph data sets. It uses two arrays to store a graph: an *edge list* that stores the non-zero elements of the graph's adjacency matrix in a one-dimensional array, and an *offset list* that contains the base index/pointer of the edge list elements for each vertex. For example, consider a graph and its CSR structure as shown in Fig. 3(b).

Typically, BFS graph traversal uses CSR format to conserve space by storing non-zero values. BFS traverses all vertices at the current depth (*i.e.,* distance from the source vertex) before moving onto the next depth. BFS is a fundamental algorithm, and is the basis of other graph algorithms (*e.g.,* BC and SSSP). In addition to the offset and edge lists, BFS also uses two software arrays called the *work queue* and the *visited list*. The work queue[2] stores a set of vertices to be processed in the future. The visited list keeps track of already processed vertices to avoid processing them again.

[2] An alternate implementation of work queue uses dual buffering with two frontier data structures (current and next); this paper focuses on a sliding queue based work queue structure that is conceptually same as frontiers.

Fig. 3(a) describes the traversal pattern of the BFS algorithm. We assume that offset list and edge list data structures are populated in memory. In addition, memory is allocated for work queue and visited list. As a first step, the source vertex (`source`) is pushed onto the work queue. Then, the algorithm chooses a vertex from the work queue and scans its neighbors (by indexing into offset list and edge list). If any of the scanned neighbors has not already been visited, then it is marked visited and is added to the work queue. A graphical representation of this traversal is shown in Fig. 3(b).

We observe two major bottlenecks in this algorithm: (a) data-dependent loads to the offset, edge, and visited lists and (b) a load-dependent branch instruction. Data-dependent reads for large-scale graphs are costly latency-wise because of their massive data footprint and random memory access patterns. Due to lack of locality, data for most of these loads are not found in caches. Moreover, control-flow instructions incur high penalty for two reasons. First, their data-dependent nature makes it challenging for branch predictors to predict the correct branch outcomes. Second, as reported by Srinivasan and Lebeck [89], in the case of an incorrectly predicted branch, much unnecessary work is performed while waiting for the load operation to return its data and correct the mispredicted branch. To better understand this bottleneck, Fig. 4 shows the breakdown of execution times for various irregular workloads running on an eight-core machine with three levels of cache hierarchy using the methodology shown in Section V. The figure clearly shows that these applications are stalled on DRAM for more than 50% of the time and have non-negligible branch misprediction stalls.

III. Proposed Programming Model

Prodigy's novel programming model captures an algorithm's semantic behavior, including its data structure layout and memory access patterns, in a compact graph representation which is communicated to the hardware. We present two techniques to construct this representation within the program—(a) manual code insertion by the programmer, and (b) automatic code generation using compiler analysis.

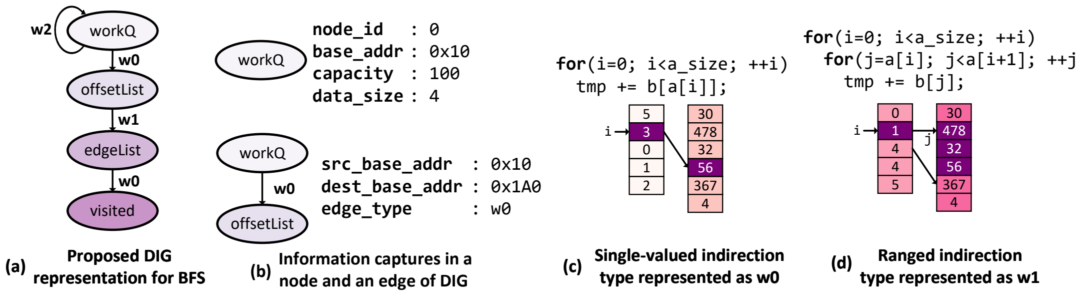


Figure 5. Proposed Data Indirection Graph (DIG) representation—(a) example representation for BFS, (b) data structure memory layout and algorithmic traversal information captured by a DIG node and a weighted DIG edge respectively; two unique data-dependent indirection patterns supported by Prodigy—(c) single-valued indirection, and (d) ranged indirection.

A. Data Indirection Graph (DIG): Compact Representation of Program Semantics

We make the key observation that *two specific* data-dependent indirect memory access patterns are used in a wide range of irregular workloads. Taking this as a foundation, we can construct combinations of these patterns that span sets of irregular memory accesses for different algorithms.

With this insight, we propose a graph representation, which we call a Data Indirection Graph (DIG), to capture the relationship between data structures for irregular algorithms. In a DIG, each node represents a data structure (*e.g.,* the visited list in BFS), and each directed weighted edge represents a data-dependent access. Fig. 5 shows an example DIG representation for the BFS algorithm. Nodes of the DIG, which store data structure information, have the following fields: `node_id`—a unique identifier to reference the data structure, and an address identifier—a method for identifying which part of the address space belongs to the data structure represented by the node. For example, the address identifier for an array are: `base_addr`—base address of the array, `capacity`—number of data elements in the array, and `data_size`—data size of each element of the array in bytes.

Edges of the DIG, which store the algorithmic traversal pattern between data structures have the following fields: `src_base_addr`—base address of the source data structure from which data are read to index into the destination data structure, `dest_base_addr`—base address of the data structure that is indexed into, and `edge_type`—data-dependent indirect access pattern from source node to destination node. As stated before, Prodigy supports two types of indirection patterns that are abstracted using edge weights of `w0` and `w1`. Fig. 5(c,d) show these two types of data-dependent indirection functions supported by our representation, *i.e.,* single-valued indirection (*e.g.,* indirection between edge list and visited list for BFS) and ranged indirection (*e.g.,* indirection between offset list and edge list in BFS). Additionally, we define a special edge called a *trigger edge* (`w2` in Fig. 5(a)), which is a self-edge to the data structure triggering prefetches. Trigger edge contains `node_base_addr`—data structure base address, and `edge_type`—details of prefetch sequence initialization (more details in Section IV). A trigger edge represents the control flow specifying the prefetch sequence to initialize.

```
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);
  […]
```

Figure 6. Annotated BFS source code to construct the DIG.

B. Construction and Communication of the DIG

This section discusses how to generate the DIG representation from software and communicate it to hardware. We first describe how a programmer can achieve this by manually inserting simple annotations to the application source code using our API calls. To reduce the burden on the programmer, we further propose a compiler analysis and code generation technique to automatically analyze the application source code, construct the DIG representation, and instrument the application binary using the proposed API calls.

1) Using Programmer Annotations: Assuming that the programmer is cognizant of the key data structures and traversal algorithms used in the application, they can add simple API calls in the application source code to construct the DIG representation. Fig. 6 presents these modifications for BFS, where three unique API calls are used to annotate the DIG. `registerNode()`—register a node of the DIG. This call writes a node's information into the prefetcher memory; the arguments to this call are the base address of this data structure, total number of elements, size of data elements, and the node ID. `registerTravEdge()`—register an edge of the DIG. This call writes edge information into the prefetcher memory; the arguments to this call are the addresses of the source and destination nodes, and the type of indirection (*i.e.,* `w0`/`w1` as shown in Fig. 5). `registerTrigEdge()`—register a trigger edge of the DIG. This call writes the base address of the trigger data structure into the prefetcher registers. The second argument (`w2`) holds information about the type of prefetch to be initiated (more details in Section IV-C).

(a)

```
1: void main()
2: {
3:   int * a = malloc(size);
4:   int * b = malloc(size);
5:   [...]
6:   kernel(a, b, dst);
7: }
```

(b)

```
1: void kernel(int* a,
        int* b, int* dst)
2: {
3:   for(int i = 0; i < size; ++i)
4:     *(dst+i) = b[a[i]];
5: }
```

Compiler analysis — Code generation

(c)

```
 1: define void @kernel(i64* %0, i64* %1, i64* %2) {
 2:   %10 = call i32 @registerTrigEdge(i64* %0, i32 5)
 3:   %11 = call i32 @registerTravEdge(i64* %0, i64* %1, i32 1)
 4:   [...]
 5:   ; loop
 6:   %16 = getelementptr inbounds i64, i64* %0, i64 %.01
 7:   %17 = load i64, i64* %16, align 4
 8:   %19 = getelementptr inbounds i64, i64* %1, i64 %17
 9:   %20 = load i64, i64* %19, align 4
10:   [...]
11: }
12:
13: define void @main() {
14:   %3 = call i8* @malloc(i64 4000)
15:   %4 = call i32 @registerNode(i64* %3, i32 1000, i32 4, i32 0)
16:   %7 = call i8* @malloc(i64 4000)
17:   %8 = call i32 @registerNode(i64* %7, i32 1000, i32 4, i32 1)
18:   [...]
19:   call void @kernel(i64* %3, i64* %7, i64* %10)
20: }
```

Figure 7. An example C program (a) and (b), translated into LLVM IR (c) and instrumented with our API calls to register DIG nodes and edges.

2) Using Compiler Analysis: Identifying indirections in non-trivial programs (*e.g.,* [16]) can be complicated for the programmer, often requiring in-depth application knowledge. Our compiler alleviates this manual work by automatically identifying these indirections and transforms the program by annotating it with prefetcher API calls. Our compiler analyzes the *application source code once* for annotation with a negligible cost compared to the graph reordering approaches [14], [92] that incur significant cost of profiling and re-organizing the *input data set*. Node and edge identification avoids complex interprocedural analysis by performing the resolution of their relationships during execution. Prefetching is only triggered for indirections whose edges consist of these resolved and registered nodes, as seen in Fig. 8(d). This section describes the operation of our LLVM-based compiler analyses and transformations.

First, our compiler analysis extracts information required for node registration from allocations. Apart from conventional defaults (*i.e.,* `malloc`), the user can specify custom allocators. The pseudocode for this procedure is presented in Fig. 8(a). Fig. 7(c) shows two node registrations, each using information from the immediately preceding `malloc` calls. Next, by tracking the use of these nodes, it extracts edge information and detects their associated indirection patterns. Fig. 7(b) contains a single-valued indirection in the form of a load to `b[a[i]]` (line 4), which corresponds to the LLVM IR in lines 6-9 of Fig. 7(c). As the base addresses of these two arrays form the edge between the nodes, our pass extracts them and uses them in the `registerEdge()` function along with the final argument that specifies the type of edge being registered—in this case, a single-valued indirection. Our code generation pass places the edge registration calls as soon as all the required arguments have been defined. In Fig. 7, the pointers to the arrays are passed into the kernel as arguments, allowing edges to be registered at the start of the function (lines 2-3). Ranged

```
for func in module:
  for inst in funct:
    if isInstanceOf(inst, AllocCall):
      alloc = AllocCall(inst)
      alloc_info = {alloc.total_size, alloc.num_elems,
      ↪  alloc.base_ptr}
      emit(<registerNode(alloc_info)>)
```

(a)

```
# identify address calculations
for func in module:
  for inst in func:
    if isInstanceOf(inst, AddrCalc):
      source_addresses.append(inst.addr)

# find edge
for source_addr in source_addresses:
  loads = getLoadsUsing(source_addr)
  for ld in loads:
    dependent_addr_instr = getAddrCalcsUsing(ld)
    for target_inst in dependent_addr_instr:
      if isUsedInLoad(target_inst.addr):
        emit(<registerTravEdge(source_addr,
        ↪  target_inst.addr)>)
```

(b)

```
# identify address calculations
# same as in single-valued indirection above

# find edge
for source_addr in source_addresses:
  addr_calc2 = findAddrCalcWithSameBasePtr(source_addr)
  if areUsedInBoundsCheck(source_addr,addr_calc2.addr):
    target_inst = findLoadUsingAddr(source_addr)
    emit(<registerTravEdge(source_addr,
    ↪  target_inst.addr)>)
```

(c)

```
def registerNode(base_ptr, num_elems, elem_size, node_id):
  # note: the node_table is depicted in Figure 9a
  node_table.insert({base_ptr, base_ptr + num_elems *
  ↪  elem_size, node_id})

def registerTravEdge(src_addr, target_addr, edge_type):
  # note: The edge_table is depicted in Figure 9c
  src_base_addr = scan_node_table(src_addr)
  target_base_addr = scan_node_table(target_addr)
  if src_base_addr and target_base_addr:
    edgeTable.insert({src_base_addr, target_base_addr,
    ↪  edge_type})

def registerTrigEdge(addr, edge_type):
  node_base_addr = scan_node_table(addr)
  if node_base_addr:
    edge_table.insert({node_base_addr, node_base_addr,
    ↪  edge_type})
```

(d)

Figure 8. Pseudocode of Prodigy's compiler analyses for (a) node identification, (b) single-valued indirection, (c) ranged indirection, and (d) runtime.

indirection can be identified similarly. For a ranged indirection from array `a` to `b` as shown in Fig. 5(d), we detect the array accesses (*i.e.,* `a[i]` and `a[i+1]`) that control loop bounds for accessing/indexing into another array `b`. The pseudocode for identifying single-valued and ranged indirections is presented in Fig. 8(b) and 8(c), respectively.

At the final stage, our analysis picks trigger edges using the set of traversal edges identified previously. If a node from that set does not have an incoming edge, then it has a trigger edge (*i.e.,* a self-edge to the trigger node). For example, the address calculations in lines 6 and 8 in Fig. 7(c) form a traversal edge. However, because the node with address generation in line 6 does not have any incoming edges, it is designated as a trigger edge, with its registration inserted in line 2.

The code generated by our compiler pass and the programmer

(a)

Node ID	Base Address	Bound Address	Data Size	Trigger
0	0x00010	0x0019C	4	true
1	0x001A0	0x00330	4	false
2	0x00334	0x00B00	4	false
3	0x00B04	0x00C90	4	false

(b)

Edge Index
0
1
2

(c)

Src Node Addr	Dest Node Addr	Edge Type
0x00010	0x001A0	0
0x001A0	0x00334	1
0x00334	0x00B04	0

(d)

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

Figure 9. Memory structures used in Prodigy—(a) node table, (b) edge index table, and (c) edge table for storing the DIG representation, (d) prefetch status handling register (PFHR) file tracking progress for live prefetch sequences and issuing non-blocking prefetches.

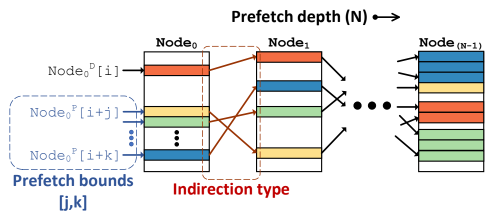


Figure 10. Prefetching algorithm initiates prefetch sequences between prefetch bounds j and k and advances a prefetch sequence using software-defined indirection types. The superscripts denote a demand (D) or a prefetch (P) access.

annotations use the same API, presented in Fig. 8(d), and can complement each other, thus improving the overall accuracy of our compiler. For example, the programmer can choose to manually annotate the relevant nodes, and rely on the compiler to identify edges.

3) Application Hardware Interface: A small SRAM-based memory unit is used on the hardware prefetcher that is memory mapped to hold the DIG. Once software generates the DIG using API calls presented above, these calls are translated into a set of store operations by a run-time library.

IV. Proposed Hardware Design

A. Memory Requirements for a DIG

Fig. 9(a-c) show three prefetcher-local memory structures to store a DIG representation. As described in Section III, the node table and the edge table store properties of DIG nodes and edges, respectively. The base address, number of elements, and data size of each node specified by software are converted into base and bound addresses by the runtime library, and then stored into the node table. Because the DIG captures program semantics from the source code, these tables store virtual addresses. Additionally, we use an edge index table to find outgoing edges from a DIG node, which mimics the software offset list in hardware. To perform prefetching, Prodigy state machine uses these structures to extract program's data structures and traversal information.

B. The Prefetch Status Handling Registers

A typical prefetch sequence for graph workloads can span four or more data structures. While the prefetcher is waiting to receive multiple outstanding data requests, it is important to track which responses belong to which issued requests. In addition, prefetch opportunities may be lost if the prefetcher is blocking, *i.e.,* waiting for a whole prefetch sequence to complete before accepting a new one. To address these challenges, we introduce a hardware structure called PreFetch status Handling Register (PFHR) file for Prodigy, which addresses both of these issues at once. While PFHRs are analogous to the Miss Status Handling Registers (MSHRs) in non-blocking caches, PFHRs have a unique design because they also have to track the status of long prefetch sequences in addition to making their host hardware structure non-blocking.

Fig. 9(d) shows the hardware structure for PFHR file, where each row has the following entries. `Free` indicates if a PFHR is free or occupied. `Node ID` denotes the DIG node ID of an outstanding prefetch request. `Prefetch trigger address` stores the *virtual address* from which the prefetch sequence is initiated. This is used to drop the prefetch sequence if the demand sequence advances close to the prefetch sequence. `Outstanding prefetch addresses` stores the cache line-aligned *physical addresses* of outstanding prefetch requests. Upon a prefetch fill, Prodigy performs a CAM look-up in this column to find the PFHR that is keeping track of that request. `Offset bitmap` stores a bitmap of outstanding prefetch byte-addresses in a cache line whose address is indicated in the previous entry.

C. Prefetching Algorithm

The prefetching algorithm has two phases: (a) prefetch sequence initialization and (b) prefetch sequence advance.

1) Prefetch Sequence Initialization Algorithm: This algorithm dictates actions to perform upon a prefetch trigger event. A prefetch trigger event occurs when Prodigy observes a demand load request to a data structure with a trigger edge. To dynamically adapt to changing machine states (*e.g.,* cache contents), Prodigy initializes *multiple* prefetch sequences at once and selectively drops some prefetch sequences.

The role of a trigger edge is to indicate the parameters to initialize prefetch sequence(s), which include the prefetch bounds and prefetch direction as shown in Fig. 10. The prefetch bounds represent a look-ahead distance for prefetching (*i.e.,* j) and the number of prefetch sequences to initialize (*i.e.,* $k - j + 1$). Additionally, the data structure traversal direction can also be defined, *i.e.,* ascending or descending order of their memory addresses. Intuitively, when the prefetch depth, *i.e.,* number of nodes on the DIG's critical path, is high, the time to traverse an entire path is long. Hence, a small look-ahead distance is effective to balance data processing and data fetch times. Similarly, for a short critical path, a large look-ahead distance is effective. This simple intuition is incorporated in a heuristic to determine the prefetch look-ahead distance, where the distance decreases with an increase in the prefetch depth of up to three. For algorithms traversing through four or more data structures, a look-ahead distance of one is used. In practice, we found there was little performance variation when the look-ahead distance is up to 4× smaller/greater than the ideal value.

Moreover, to adapt to dynamic data processing speed of the core, Prodigy uses a feedback from load requests to selectively drop prefetch sequences. As shown in Fig. 9(d), we store a trigger address in each PFHR entry to record the starting address of the prefetch sequence. When the core demands the trigger address of a live prefetch sequence, we drop the sequence because the prefetcher can only partially hide the memory latency. Instead, we choose to hide the full latency of future load operations by prefetching ahead. *This way, dropping of prefetch sequence(s) helps Prodigy to always run ahead of the core, and multiple prefetch sequence initialization ensures the liveliness of some prefetch sequence(s) even if few others are terminated.*

2) Prefetch Sequence Advance Algorithm: Upon servicing a prefetch, Prodigy reads its data to issue further prefetch requests using two types of indirection functions, *i.e.,* single-valued indirection and ranged indirection (see Section III-A).

Single-valued indirection is an indirection type that connects two arrays, where the source array stores indices/pointers to index into the destination array as shown in Fig. 5(c). This traversal function is common in irregular algorithms (*e.g.,* graph algorithms use vertex identifier to index into data storage (*e.g.,* visited list for BFS and vertex scores for PageRank)). Notably, pointers are a special class of this indirection type, where the address of the destination can be found by using the pointer itself. With node information stored in the DIG, the prefetcher can interpret the address as an index (or a pointer) and indexes into the next array as done in software using the base address and data size of the next DIG node.

Ranged indirection is an indirection type in which an array stores pairs of base and bound indices (or pointers) pointing to a section of another array which is accessed together as shown in Fig. 5(d). Fundamentally, this access pattern summarizes a streaming access through a portion of memory specified by this pair. For example, in CSR/CSC representations, ranged indirection is used in graph algorithms to find neighbors of a vertex using offset list and edge list.

D. Hardware Flow of Prodigy

Fig. 11 shows the operation of Prodigy and its interaction with the rest of the system. The figure shows that the graph data structures are populated in memory for the BFS algorithm on an example graph same as Fig. 3. For simplicity, we assume that a cache line size is a single data block and caches are not yet populated. Once the prefetcher is programmed, it snoops on load requests from the core to the L1D and waits for a demand request within the address ranges of the data structure with the trigger edge. Similar to the prefetching algorithm, Prodigy state machine has two phases for issuing prefetches: prefetch sequence initialization and advance.

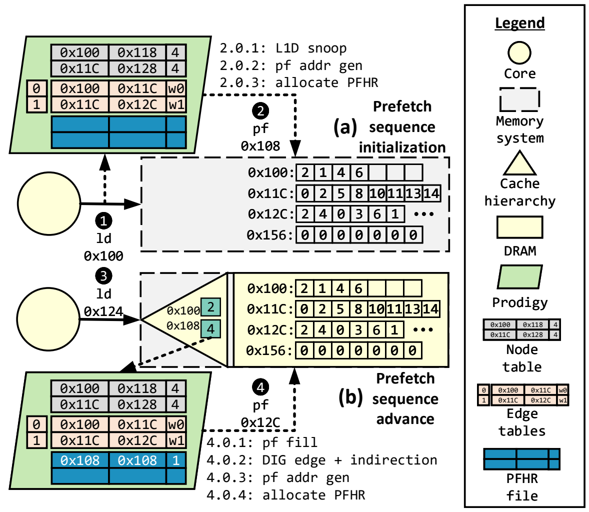


Figure 11. Prodigy operation: (a) prefetch sequence initialization, and (b) prefetch sequence advance.

Fig. 11(a) shows Prodigy's operation in the first phase. Upon observing a load request ❶ that falls in the trigger data structure (*i.e., workQueue*), a prefetch sequence is initialized. Based on the prefetch-lookahead distance of (let us assume) 2 communicated via a trigger edge as described in Section IV-C, Prodigy computes memory address `0x108` (*i.e.,* `0x100+2×4`) to prefetch. Lastly, this address is translated to a physical address using the TLB and issued for prefetching ❷. A new PFHR is allocated for tracking this prefetch request.

Fig. 11(b) shows the second prefetching phase, where demand and prefetch requests are serviced with their data resident in the cache. Upon receiving the demand request, the core traverses through other data structures ❸ `ld 0x124` (`0x11c+2×4`; using index of 2 and data size of 4). Note that further load requests do not trigger prefetch sequences until another access to *workQueue*. Upon prefetch fills, Prodigy finds the PFHR entry keeping track of this request using a CAM look-up. Once identified, a source DIG node corresponding to this prefetch fill, its outgoing edges, and data indirection type are found by indexing into the edge and edge index tables. Using the single-valued indirection `w0` and prefetched data, next prefetch address of `0x12C` is computed. Lastly, a prefetch request is sent ❹ by translating its address using the TLB and a new PFHR is allocated; this process repeats until a leaf DIG node is encountered. A new PFHR is only allocated for prefetch addresses belonging to non-leaf DIG nodes.

E. Prodigy in a Parallel Execution Setting

In a multi-core execution, a private instance of Prodigy is present on each core. Prodigy snoops on the L1D cache to trigger prefetch sequences. Prodigy supports trigger data structures that are contiguously partitioned across multiple threads in the virtual address space. Thus, Prodigy supports both statically-scheduled (OpenMP-static) and dynamically-scheduled or work stealing-based compilers (OpenMP-dynamic, CILK [36]). With this contiguous partitioning, Prodigy mostly prefetches the

TABLE I
BASELINE SYSTEM CONFIGURATION.

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

correct data for each core; this prevents any significant increase in NoC/coherence traffic. The only exception is present at the data structure boundaries, which are rarely accessed. Timeliness in presence of synchronization is maintained by selectively dropping prefetch sequences based on each core's execution pace.

F. OS Integration

Prodigy works best when the number of user threads does not exceed the core count. This allows the use of thread affinity to ensure only one user context is needed in the prefetcher. In the event that a thread which uses Prodigy is preempted by the kernel, the prefetching is paused upon thread descheduling. The data in Prodigy's prefetcher-local memory structures remains untouched. This cached data can be used to resume prefetching when the thread is rescheduled. In the rare event that another user thread is scheduled that requires the prefetcher, the context needs to be saved/restored from the prefetcher data structures.

G. Prefetch Throttling Mechanism

While Prodigy focuses on designing a novel prefetching mechanism, we do not implement a prefetch throttling mechanism because it is out of the scope of this paper. We envision Prodigy to be used alongside a prefetch throttling mechanism similar to [88] that can identify and prevent prefetch-induced cache pollution to further improve performance. We leave studying the best throttling techniques as future work.

V. METHODOLOGY

This section describes the simulation infrastructure, algorithms and data sets, and state-of-the-art prefetching systems.

A. Simulation Infrastructure

We use Sniper [20]—a Pin [62] based x86 multi-core simulator with an interval core simulation model. Sniper has been validated against several Intel micro-architectures [10], [20], [21]. We use CACTI [70] to obtain cache access times for different cache capacities. We use the McPAT [60] model built into Sniper to model energy consumption. We implement our compiler analysis techniques using LLVM passes [57]. We evaluate our approach by modeling a parallel shared memory system with 8 cores as described in Table I. We run our workloads end-to-end and report the performance numbers by ignoring initialization cost, *i.e.,* reading a graph from a file and populating data structures. We use the region-of-interest (ROI) utility from Sniper to only profile the core algorithm.

TABLE II
REAL-WORLD GRAPH DATA SETS USED FOR EVALUATION.

Graph	Number of vertices	Number of edges	Size (in MB)	$\times$ 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

B. Irregular Workloads

We use unmodified versions of the following workloads and run through our compiler pass for analysis.

Algorithms. We use five graph algorithms from the GAP benchmark suite (GAPBS) [16] for evaluation—Betweenness Centrality (`bc`), Breadth-First Search (`bfs`)[3], Connected Components (`cc`), PageRank (`pr`), and Single-Source Shortest Path (`sssp`). We also use Sparse Matrix-Vector multiplication (`spmv`) and Symmetric Gauss-Seidel smoother (`symgs`) from the HPCG benchmark suite [29] as representative sparse linear algebra applications. Additionally, we use Conjugate Gradient (`cg`) and Integer Sort (`is`) from the NAS parallel benchmark suite [12] as representative computational fluid dynamics applications. We choose these algorithms as they exhibit single-valued and/or ranged indirections.

Data sets. As inputs to the graph algorithms, we use real-world graph data sets from SNAP [59] and UF's sparse matrix collection [27] as shown in Table II. We selected these data sets as they represent real-world graph data and offer diversity in total size as well as number of vertices and edges. The primary reasons for avoiding the use of the graph generators `kron` and `urand` from GAPBS are (a) they are synthetic data sets, and (b) they are severely bound by synchronization overheads when evaluated on our simulation infrastructure. Unless shown individually, results for each graph algorithm is averaged over all data sets. For non-graph algorithms, we use input generators from benchmark suites; data set sizes for the linear algebra and fluid dynamics kernels are 2M$\times$2M, and 33M (for `is`) and 75k (for `cg`), respectively.

VI. RESULTS

A. Design Space Exploration

We perform design space exploration on Prodigy to understand the trade-off between performance and hardware complexity. Fig. 12 shows the effect of PFHR file size on the overall performance normalized to a baseline of 4 registers. The figure illustrates two key findings. **First**, there is up to 30% performance difference between the performance-optimal configuration and the baseline PFHR file size. The performance

[3]For a fair comparison with prior work, we only use a top-down implementation of the `bfs` algorithm.Prodigy can also adapt to direction-optimizing BFS by re-configuring the DIG during run-time.

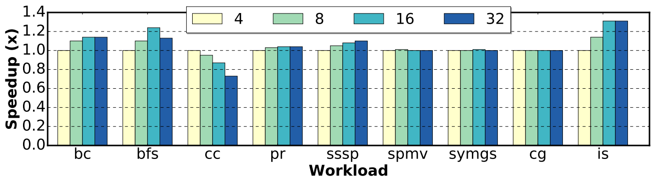


Figure 12. Design space exploration on the PFHR file size. Performance of each configuration is normalized to 4 entries.

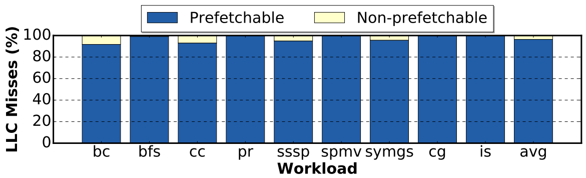


Figure 13. Classification of LLC miss addresses into potentially prefetchable and non-prefetchable addresses.

difference is attributed to structural hazards in the PFHR file—while issuing a prefetch, if the entire PFHR file is busy, the prefetch is dropped. We choose the size of PFHR file to be 16 for our design since it offers a reasonable trade-off between performance and storage area requirement. **Second**, increasing the number of PFHRs beyond 8 for `cc` hurts its performance since the benefits of timely prefetches are overshadowed by untimely prefetches that pollute the cache system. Dynamically adapting prefetch aggressiveness according to the usefulness of prefetched cache lines might help improve the performance of such workloads.

B. Prefetching Potential

To estimate the potential prefetch coverage of Prodigy, Fig. 13 evaluates the fraction of LLC misses, for a non-prefetching baseline, that Prodigy can prefetch. We evaluate this using DIG-annotated application binaries, disabling the prefetcher, and classifying LLC miss addresses based on whether they are within or outside the data structure address bounds annotated by the DIG. The figure shows that, on average, 96.4% of LLC misses can be prefetched. In other words, ideal prefetching and caching resources would convert an average of 96.4% of DRAM accesses into cache hits, which sets the upper bound for our evaluation.

C. Effect on Performance

Prodigy vs. no-prefetching: Fig. 14 shows the CPI stacks and speedups of Prodigy across all the workloads normalized to a non-prefetching baseline. For each workload, the first and second bars correspond to the CPIs of baseline and Prodigy, respectively. The figure shows the breakdown of execution time in terms of no-stalls and stalls because of DRAM and cache accesses, branch mispredictions, dependent instructions, and others. Prodigy achieves a significant average speedup of 2.6× compared to a non-prefetching baseline.

We see that Prodigy gains most of its performance by decreasing the DRAM stalls by an average of 80.3%. Notably,

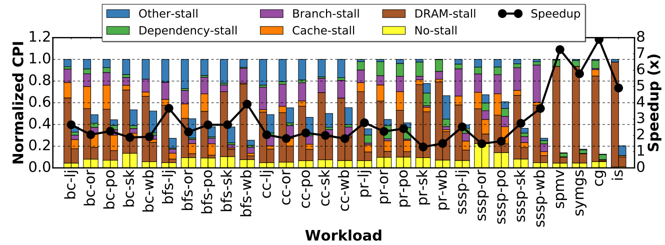


Figure 14. CPI stack comparison and speedup achieved by Prodigy against a non-prefetching baseline. Left bar: CPI stack of baseline; right bar: CPI stack of Prodigy normalized to baseline. Lower is better for CPI, higher for speedup.

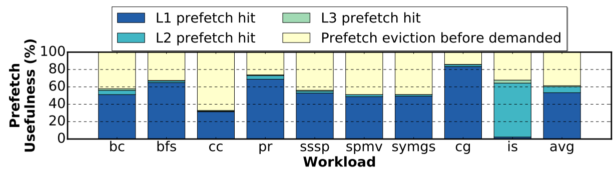


Figure 15. Location of prefetched data in the cache hierarchy when it is demanded. Blue is better.

the DRAM stall portion of the baseline non-graph workloads is 88.4% of the overall CPI, leading to substantial savings and speedups. Assuming that software communicates the correct workload semantics to the prefetcher, it mostly fetches useful data. The primary inefficiency stems from issuing untimely prefetches. We address this challenge by prefetching for the next few work queue items and dropping prefetch sequences after detecting that the core has caught up. This heuristic allows us to avoid cache pollution by modulating the number of requested cache blocks while also freeing PFHRs for more useful work if their prefetch sequences would only partially hide the memory latency. Note that the `pr` implementation uses both CSC and CSR graph data structures that achieves a similar speedup as other algorithms that only use CSR format. Furthermore, as a result of reduction in DRAM stalls, Prodigy slightly increases the cache stall portion of the CPI stack. This is due to converting DRAM accesses into cache hits that increases the aggregate time spent on cache accesses.

Additionally, mostly for graph workloads, Prodigy reduces the branch segment of the CPI stack by 65.3% on average as a side effect of reducing DRAM stalls. This is especially evident in `bfs`, `pr`, and `sssp` due to the prevalence of load data dependent branches. For example, in `bfs`, a vertex is only added onto the work queue after loading its visited list entry and verifying that it has not been traversed yet. This finding is consistent with prior work [89].

Prefetch Usefulness: Fig. 15 classifies the usefulness of prefetched data into four categories—demanded and resident in the L1/L2/L3 cache and evicted from the cache hierarchy without being demanded. The figure shows that data brought in by 32.9–85.8% of prefetch requests is demanded before it is evicted, which shows the accuracy of our prefetcher. On average, our prefetcher achieves an accuracy of 62.7%. Furthermore, most of these cache hits are found in the L1D

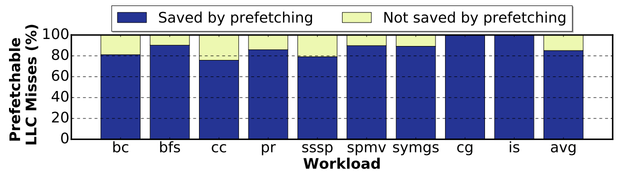


Figure 16. Percentage of prefetchable main memory accesses (as shown in Fig. 13) converted to cache hits. Blue is better.

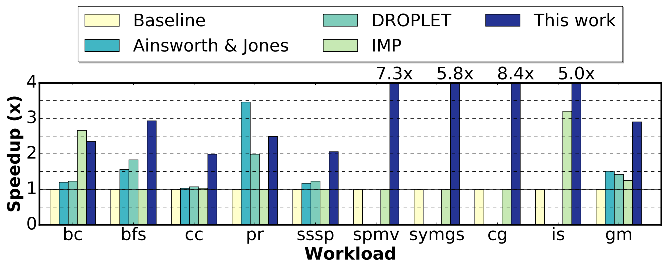


Figure 17. Performance comparison of a non-prefetching baseline, Ainsworth and Jones' prefetcher [6], DROPLET [15], IMP [99], and Prodigy (this work). Higher is better. Ainsworth & Jones and DROPLET are graph-specific approaches, and hence are omitted from non-graph workloads.

cache, which incurs the lowest latency of the load operations. Note that since Prodigy benefits from static analysis information provided by software, the fraction of evicted data can further be reduced by using an intelligent caching policy (*e.g.,* stream buffers or scratchpads [1]) since eviction is a consequence of imperfect timeliness. Fig. 16 shows the percentage of prefetchable LLC misses (blue portion of the bar in Fig. 13) that Prodigy converts into cache hits. On average, Prodigy converts 85.1% of prefetchable LLC misses to cache hits.

Significance of ranged indirection: For graph algorithms, ranged indirection is responsible for prefetching 35.4–75.9% (55.3% on average) of all data (not shown because of space limitation). The fraction of data prefetched using ranged indirection depends both on the position of indirection types in a prefetch sequence and the amount of data available to prefetch. For example, a major source of single-valued indirection in `bfs` is at a prefetch depth of four. At this depth, secondary effects, like squashing of prefetch sequences and PFHR unavailability, limit prefetching opportunities. Prior work [26], [79], [99] only prefetch single-valued indirection and fail to capture a significant prefetching opportunity.

Prodigy vs. hardware prefetchers: Next we compare the performance of Prodigy with the state-of-the-art hardware prefetchers including GHB-based G/DC data prefetcher [72], Ainsworth and Jones' prefetcher [6], DROPLET [15], and IMP [99]. Notably, the benefits of different prefetching solutions are highly sensitive to architectural parameters, graph traversal algorithm and design of their data structures, and input data sets. Hence, we present a comparison using the parameters from our simulation framework as well as a comparison with the best reported results on commonly evaluated algorithms from each prior work.

Prodigy outperforms the baseline and a GHB-based G/DC data prefetcher [72] (not shown because of space limitations) by 2.6× on average. GHB-based G/DC is known to predict inaccurate prefetch addresses for irregular memory accesses due to the lack of spatial locality, polluting the cache. Therefore, when Prodigy is enabled by software, other traditional prefetchers (*e.g.,* GHB, stride, stream) are disabled.

Fig. 17 shows the performance comparison of various prefetchers using our simulation framework. Prodigy outperforms Ainsworth and Jones' prefetcher[4] [5], [6] by 1.5×. We have verified with the authors [4] that our implementation and results are correct. The difference compared to [6] can be attributed to inaccurate prefetch timeliness. On average, 62.7% of Prodigy's prefetches are demanded by the core versus only 44.6% for [6]. Also, unlike Prodigy, initiating one prefetch sequence in [6] sometimes only partially hides the memory latency if the core catches up with the prefetcher. Furthermore, Prodigy is more flexible in that it can adapt with different combinations of data structures and indirection patterns, whereas Ainsworth and Jones' graph prefetcher aims to prefetch for BFS-like access patterns. While an extension of [6] is presented in [5], it incurs significant area overhead of 32KB of storage vs. 0.8KB for Prodigy.

Compared to DROPLET [15], Prodigy achieves a 1.6× speedup on average for two reasons. First, DROPLET only prefetches a subset of data structures, *i.e.,* edge list and visited list-like arrays exhibiting single-valued indirection, compared to Prodigy, which prefetches other graph data structures as well. Second, we notice that DROPLET MPP misses several prefetching opportunities because it can only trigger further prefetches from prefetch requests serviced from DRAM, while much of the prefetched data are present in the cache hierarchy.

Prodigy achieves an average speedup of 2.3× compared to IMP[5] [99], because IMP can only detect streaming accesses to data structures that perform `A[B[i]]` type prefetching and it only supports up to two levels of indirection. Extending both DROPLET and IMP to prefetch additional data structures would require significant effort because they do not support ranged indirection and DROPLET design is specific to a subset of graph data structures.

While Prodigy shows a significant speedup over prior work on our simulation environment, we could not reproduce similar results reported in the prior publications despite obtaining evaluation artifacts from the authors. We believe that this discrepancy is attributed to the difference in simulation environment, architecture parameters, and benchmark implementations. To offer better justice to prior work, we also compare Prodigy with the best reported speedups of hardware prefetchers from their original publications. Table III shows a comparison of best reported speedups over a non-prefetching baseline for optimal algorithm-data set combination for both Prodigy and prior work. The comparison shows that even compared to the best-

[4]We used open-sourced artifacts of for the evaluation of [6], and verified the presented results with the authors [4].

[5]We used the artifacts provided by the authors for evaluating IMP.

TABLE III
AVERAGE SPEEDUP COMPARISON OVER NO PREFETCHING.*

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×

*Best-performing input data sets used as reported in prior work.

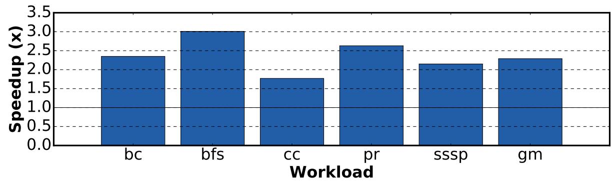


Figure 18. Speedup of Prodigy compared to a non-prefetching baseline on reordered graph data sets using HubSort [14].

reported speedups, Prodigy still outperforms the state-of-the-art hardware prefetchers.

Prodigy vs. software prefetching: We compare the performance of Prodigy with a software prefetching technique [8] for indirect memory accesses. To make our evaluation consistent with [8], we evaluated the performance of software prefetching on an Intel Broadwell microarchitecture and validated our results with authors of [3]. Our findings show that for pr, performing a pure software-based prefetching [8] achieves an average speedup of 7.6% compared to an average speedup of 2× for our approach (not shown due to space limitation). This is because Prodigy benefits from both static analysis information from software and dynamic run-time information from hardware to perform efficient prefetching. We do not report the results on other graph algorithms since we noticed that the compiler pass of [8] is not able to detect dynamically allocated array sizes, and conservatively avoids placing prefetch instructions to prevent faults [3].

Graph reordering: We also evaluate the performance benefits of Prodigy on reordered graphs using HubSort [14]. Fig. 18 presents the speedup of Prodigy compared to a non-prefetching baseline (both using graph reordering) for graph algorithms. The figure shows even after benefiting from added locality because of graph reordering, irregular memory accesses can still limit the performance, and Prodigy can further improve this performance by 2.3× on average.

D. Effect on Energy

Fig. 19 shows the breakdown of energy consumption for Prodigy normalized to the baseline. Prodigy reduces energy consumption across all categories with an average reduction of 1.6×. We primarily attribute the energy reduction to the static energy savings of the core, cache, and DRAM due to the reduced workload execution time. Accelerating long-latency memory operations also saves energy by reducing the number of instructions executed and memory accesses performed before recovering from mispredicted branches [89].

E. Overhead Analysis

Prodigy's hardware consists of a finite-state machine, whose area is dominated by the storage structures discussed in Section IV-A. These structures include DIG tables (*i.e.,* node table, edge table, and edge index table) and PFHRs. Although Prodigy reads data values for prefetching, this is done by snooping on the data response buses, rather than adding or sharing ports on the cache. This limits the performance impact and area overhead. Prodigy might increase the D-TLB contention, however, this is a known issue for prefetchers operating in the virtual address space.

We estimate the area overhead in terms of storage area requirements assuming 48-bit physical and 64-bit virtual address spaces. We calculate that the largest DIG used by our workloads has 11 nodes and 11 edges for bc. For a plausible extension to store larger DIGs, we conservatively assume 16-entry DIG tables. Moreover, based on Fig. 12, we use 16 PFHRs for our design. Using these parameters, we estimate the storage requirements of DIG tables and PFHRs to be 0.53KB and 0.26KB, respectively, totaling to just 0.8KB. Assuming this storage area to be dominant, we project our prefetcher to have a negligible area overhead of 0.004% compared to an entire CPU chip. Compared to Prodigy, other work has area overheads of 1.4× [99], 2× [6], 9.7× [15], and 40× [5].

In terms of the software overhead, adding one-time prefetch API calls slightly increases the size of program binaries. Because these calls are executed only once, they translate into a negligible dynamic instruction count increase. To add these API calls, our compiler analysis performs a linear scan of a program's static instructions. The average compilation time added to our benchmarks is less than one second.

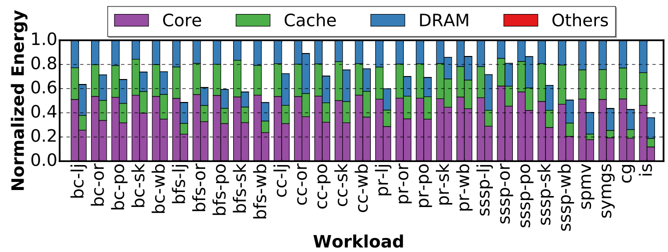


Figure 19. Normalized energy comparison of a non-prefetching baseline (first bar) and Prodigy (second bar). Lower is better.

F. Discussion on Scalability

Because of the irregular memory access patterns of evaluated workloads, cores are mostly stalled to receive responses from the memory system. Based on the baseline memory bandwidth utilization results and a bandwidth limit of 100GB/s, increasing the number of cores to around 40 will fully saturate the memory bandwidth, at which point, the benefits from prefetching will be limited. Our evaluation shows a more cost-effective design point where an 8-core system used with Prodigy can saturate the memory bandwidth while consuming 5× less transistor area and less static energy compared to a 40-core system without prefetching.

G. Limitations of Prodigy

A subset of irregular algorithms exhibiting single-valued/ranged indirection patterns also incorporate additional run-time information to issue load operations. For example, triangle counting algorithm in GAPBS [16] intelligently avoids redundant computation by examining only neighbors with higher vertex IDs than the source vertex (*i.e.,* branch-dependent loads). While Prodigy supports prefetching for indirect memory accesses, it does not account for this additional control-flow information for prefetching. Similar trends might be observed for ordered graph algorithms [28], [103] because node priority is not accounted for prefetching. In such cases, Prodigy might prefetch inaccurate vertices, and we envision using a mechanism that disables the prefetcher when it detects cache thrashing [88]. Additionally, the storage cost of hardware structures (*i.e.,* DIG tables and PFHR file) was chosen to fit the needs of the workloads evaluated in this paper. It is possible that other workloads with more DIG nodes/edges would require greater storage and PFHR resources. We leave the study of incorporating additional prefetching information and larger workload analysis for future work.

VII. Related Work

There is a rich body of work alleviating the memory access bottleneck for various workloads, especially through prefetching. This work employs a unique synergy of both hardware and software optimizations through the novel DIG representation. We divide the related work in different categories and discuss how our work is different.

Decouple access execute (DAE) architectures [18], [40], [48], [63], [84], [85], [90] use decoupled memory access and execute streams to reduce memory latency and communicate between them using architectural queues. While we use a separate prefetching unit for *accelerating memory accesses*, we still use a single thread with coupled access and execute streams with no additional requirement of queues for communication.

Helper threads [22], [24], [25], [47], [102] propose using a separate thread to speculatively prefetch data to reduce memory latency of the main thread. **Run-ahead execution** [30], [71] and some other architectures [38], [104] utilize additional or unused hardware resources to prefetch useful data for the main thread. Helper threads dedicate extra physical cores to perform prefetching that reduces compute throughput. Unlike Prodigy, runahead execution has to re-execute instructions after long-latency load-instructions.

More recently, several graph algorithm-based **hardware prefetchers** [5], [6], [15] have been proposed that assume graph data structure knowledge at hardware and prefetch for accesses falling in these data structures. Accelerating irregular workloads using hardware prefetchers [37], [54]–[56], [73], [77], [95], [99] has been long studied that cover other types of data structures and memory access patterns containing linked lists, binary trees, hash joins in application domains such as geometric and scientific computations, high-performance computing, and databases. Furthermore, several **temporal prefetchers** [46], [93], [95], [96] and **non-temporal prefetchers** [13], [17], [52], [53], [64], [82], [86] are also investigated for these workloads. These approaches however, when applied in the graph processing context, can either prefetch for a subset of data structures or incur high complexity and cost for generality. Given our compact DIG representation, our approach benefits covering all the data structures having data-dependent indirect accesses at a negligible hardware cost.

A class of prefetchers [11], [23], [26], [31], [43], [50], [79], [98] focuses on **linked data structure** traversals using pointers. They have limited applicability for graph algorithms, mainly because of the prevalence of ranged indirection as shown in the Section VI-C. Prodigy on the other hand, can cover all types of indirection present in graph algorithms.

Software prefetching [8], [19], [51], [61], [66], [91] is another technique to reduce the memory latency of both regular and irregular workloads where data structures are known at compile-time. However, software prefetching could significantly increase the size of the application binary and workloads with dynamically initialized and sized data structures are difficult to prefetch purely in software. Additionally, **direct memory access** (DMA) engines are used to move data around without explicit CPU instructions. Prodigy that reacts to hardware events is orthogonal to a DMA engine, which is primarily software controlled and used for peripheral devices.

Several **domain-specific architectures** [1], [2], [41], [67]–[69], [75], [83], [87], [97], [100], [101] have been proposed for accelerating graph processing applications. These architectures are orthogonal to our software-aided hardware prefetching work for CPUs; they either work as stand-alone accelerators, as near/in-memory processing engines, or as scheduling/intelligent caching aid to the processor core. Many of these architectures use some form of hardware prefetching support, and our low-cost prefetcher can be integrated within these architectures to further enhance their performance.

Prefetch throttling mechanisms [32], [33], [44], [45], [53], [58], [74], [78], [80], [81], [88], [94] use dynamic information such as prefetch coverage/accuracy, cache pollution, and/or bandwidth utilization to monitor the aggressiveness of prefetches. These mechanisms can be applied to our approach to reduce prefetch-induced cache pollution.

VIII. Conclusion

This paper presented Prodigy, a hardware-software co-design approach to improve the memory latency of data-indirect irregular workloads. We proposed a compact representation, called the *Data Indirection Graph (DIG)*, that efficiently abstracts an irregular algorithm's data structure layout and traversal patterns. This representation is constructed using static compiler analysis and code generation techniques, and communicated to the hardware. A programmable hardware prefetcher uses this information to cater its prefetches to irregular algorithms' memory access patterns. This approach benefits from (a) static program analysis from software to capture the irregular nature of memory accesses, and (b) dynamic run-time information from hardware to make adaptive prefetching decisions. We showed that our system is versatile and works for different sparse data representations. We evaluated the benefits of our system using a variety of irregular algorithms on real-world large-scale data sets and showed a 2.6× average performance improvement, 1.6× energy savings, and a negligible storage cost of 0.8KB.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This material is also based upon work supported by the National Science Foundation (NSF) under Grant No. NSF-XPS-1628991 and under Graduate Research Fellowship Grant No. NSF-DGE-1256260. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the NSF.

References

[1] A. Addisie *et al.*, "Heterogeneous memory subsystem for natural graph analytics," in *IISWC*, Sep. 2018, pp. 134–145.

[2] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, June 2015, pp. 105–117.

[3] S. Ainsworth, Private communication to verify Ainsworth and Jones, CGO 2017 results, 2019.

[4] S. Ainsworth, Private communication to verify Ainsworth and Jones, ICS 2016/ASPLOS 2018 results, 2020.

[5] S. Ainsworth and T. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *ASPLOS*, New York, NY, USA, 2018, pp. 578–592.

[6] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *ICS*, New York, NY, USA, 2016, pp. 39:1–39:11.

[7] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *CGO*, Feb 2017, pp. 305–317.

[8] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *CGO*, Piscataway, NJ, USA, 2017, pp. 305–317.

[9] A. Airola *et al.*, "All-paths graph kernel for protein-protein interaction extraction with evaluation of cross-corpus learning," *BMC bioinformatics*, vol. 9, no. 11, p. S2, 2008.

[10] A. Akram and L. Sawalha, "x86 computer architecture simulators: A comparative study," in *ICCD*, Oct 2016, pp. 638–645.

[11] M. Annavaram *et al.*, "Data prefetching by dependence graph precomputation," in *ISCA*, June 2001, pp. 52–61.

[12] D. Bailey *et al.*, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, Sep. 1991.

[13] M. Bakhshalipour *et al.*, "Bingo spatial data prefetcher," in *HPCA*, Feb 2019, pp. 399–411.

[14] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *IISWC*, 2018, pp. 203–214.

[15] A. Basak *et al.*, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *HPCA*, Feb 2019, pp. 373–386.

[16] S. Beamer *et al.*, "The GAP Benchmark Suite," in *arXiv:1508.03619 [cs.DC]*, 2015.

[17] R. Bera *et al.*, "Dspatch: Dual spatial pattern prefetcher," in *MICRO*, New York, NY, USA, 2019, pp. 531–544.

[18] P. L. Bird *et al.*, "The effectiveness of decoupling," in *ICS*, New York, NY, USA, 1993, pp. 47–56.

[19] D. Callahan *et al.*, "Software prefetching," *SIGPLAN Not.*, vol. 26, no. 4, pp. 40–52, Apr. 1991.

[20] T. E. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, Nov 2011, pp. 1–12.

[21] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[22] Chi-Keung Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *ISCA*, June 2001, pp. 40–51.

[23] S. Choi *et al.*, "A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching," *ACM Trans. Comput. Syst.*, vol. 22, no. 2, pp. 214–280, May 2004.

[24] J. D. Collins *et al.*, "Dynamic speculative precomputation," in *MICRO*, Dec 2001, pp. 306–317.

[25] J. D. Collins *et al.*, "Speculative precomputation: long-range prefetching of delinquent loads," in *ISCA*, June 2001, pp. 14–25.

[26] R. Cooksey *et al.*, "A stateless, content-directed data prefetching mechanism," *SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 279–290, Oct. 2002.

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[28] L. Dhulipala *et al.*, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *SPAA*. New York, NY, USA: Association for Computing Machinery, 2017, p. 293–304.

[29] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, p. 150, 2013.

[30] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *ICS*, New York, NY, USA, 1997, pp. 68–75.

[31] E. Ebrahimi *et al.*, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *HPCA*, Feb 2009, pp. 7–17.

[32] E. Ebrahimi *et al.*, "Prefetch-aware shared-resource management for multi-core systems," in *ISCA*, June 2011, pp. 141–152.

[33] E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multi-core systems," in *MICRO*, New York, USA, 2009, pp. 316–326.

[34] V. M. Eguiluz *et al.*, "Scale-free brain functional networks," *Physical review letters*, vol. 94, no. 1, p. 018102, 2005.

[35] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*. IEEE, 2011, pp. 365–376.

[36] M. Frigo *et al.*, "The implementation of the cilk-5 multithreaded language," in *PLDI*, New York, NY, USA, 1998, p. 212–223.

[37] A. Fuchs *et al.*, "Loop-aware memory prefetching using code block working sets," in *MICRO*, Washington, DC, USA, 2014, pp. 533–544.

[38] A. Garg and M. C. Huang, "A performance-correctness explicitly-decoupled architecture," in *MICRO*, Nov 2008, pp. 306–317.

[39] A. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," Microsoft Research, Tech. Rep. MSR-TR-2004-24, July 2004.

[40] J. R. Goodman *et al.*, "Pipe: A vlsi decoupled architecture," *SIGARCH Computer Architecture News*, vol. 13, no. 3, pp. 20–27, Jun. 1985.

[41] T. J. Ham *et al.*, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, Oct 2016, pp. 1–13.

[42] S. Han *et al.*, "Eie: Efficient inference engine on compressed deep neural network," in *ISCA*. IEEE Press, 2016, p. 243–254.

[43] C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 4, pp. 448–463, Apr. 2005.

[44] Y. Ishii *et al.*, "Access map pattern matching for data cache prefetch," in *ICS*, New York, NY, USA, 2009, pp. 499–500.

[45] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *ISCA*, June 2018, pp. 110–123.

[46] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *MICRO*, New York, NY, USA, 2013, p. 247–259.

[47] Jiwei Lu *et al.*, "Dynamic helper threaded prefetching on the sun ultrasparc/spl reg/ cmp processor," in *MICRO*, Nov 2005, pp. 12–104.

[48] L. K. John *et al.*, "Program balance and its impact on high performance risc architectures," in *HPCA*, Jan 1995, pp. 370–379.

[49] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *SIGARCH Computer Architecture News*, vol. 25, no. 2, pp. 252–263, May 1997.
[50] M. Karlsson *et al.*, "A prefetching technique for irregular accesses to linked data structures," in *HPCA*. IEEE, 2000, pp. 206–217.
[51] M. Khan and E. Hagersten, "Resource conscious prefetching for irregular applications in multicores," in *SAMOS*, July 2014, pp. 34–43.
[52] J. Kim *et al.*, "Path confidence based lookahead prefetching," in *MICRO*, Oct 2016, pp. 1–12.
[53] J. Kim *et al.*, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," *SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 737–749, Apr. 2017.
[54] O. Kocberber *et al.*, "Meet the walkers accelerating index traversals for in-memory databases," in *MICRO*, Dec 2013, pp. 468–479.
[55] N. Kohout *et al.*, "Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes," in *PACT*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 268–279.
[56] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *HPCA*, Feb 2014, pp. 614–625.
[57] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.
[58] C. J. Lee *et al.*, "Prefetch-aware dram controllers," in *MICRO*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 200–209.
[59] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[60] S. Li *et al.*, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, Dec 2009, pp. 469–480.
[61] M. H. Lipasti *et al.*, "Spaid: software prefetching in pointer- and call-intensive environments," in *MICRO*, Nov 1995, pp. 231–236.
[62] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, New York, NY, USA, 2005, pp. 190–200.
[63] W. Mangione-Smith *et al.*, "The effects of memory latency and fine-grain parallelism on astronautics zs-1 performance," in *Twenty-Third Annual Hawaii International Conference on System Sciences*, vol. 1, Jan 1990, pp. 288–296 vol.1.
[64] P. Michaud, "Best-offset hardware prefetching," in *HPCA*, March 2016, pp. 469–480.
[65] A. Mislove *et al.*, "Measurement and analysis of online social networks," in *IMC*, New York, NY, USA, 2007, pp. 29–42.
[66] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 12, no. 2, pp. 87–106, Jun. 1991.
[67] A. Mukkara *et al.*, "Cache-Guided Scheduling: Exploiting Caches to Maximize Locality in Graph Processing," in *AGP*, June 2017.
[68] A. Mukkara *et al.*, "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," in *MICRO-51*, October 2018.
[69] A. Mukkara *et al.*, "PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates," in *MICRO-52*, October 2019.
[70] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to understand large caches," in *HP laboratories*, 2009.
[71] O. Mutlu *et al.*, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *HPCA*, Feb 2003, pp. 129–140.
[72] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA*, Feb 2004, pp. 96–96.
[73] K. Nilakant *et al.*, "Prefedge: Ssd prefetcher for large-scale graph traversal," in *SYSTOR*. New York, NY, USA: ACM, 2014, pp. 4:1–4:12.
[74] A. V. Nori *et al.*, "Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies," in *ISCA*, June 2018, pp. 96–109.
[75] M. M. Ozdal *et al.*, "Energy efficient architecture for graph analytics accelerators," *SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, Jun. 2016.
[76] L. Page *et al.*, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.
[77] L. Peled *et al.*, "Semantic locality and context-based prefetching using reinforcement learning," in *ISCA*, June 2015, pp. 285–297.
[78] S. H. Pugsley *et al.*, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *HPCA*, Feb 2014, pp. 626–637.
[79] A. Roth *et al.*, "Dependence based prefetching for linked data structures," *SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 115–126, Oct. 1998.
[80] V. Seshadri *et al.*, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *PACT*, Sep. 2012, pp. 355–366.
[81] V. Seshadri *et al.*, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *TACO*, vol. 11, no. 4, pp. 51:1–51:22, Jan. 2015.
[82] M. Shevgoor *et al.*, "Efficiently prefetching complex address patterns," in *MICRO*, Dec 2015, pp. 141–152.
[83] S. G. Singapura *et al.*, "Oscar: Optimizing scratchpad reuse for graph processing," in *HPEC*, Sep. 2017, pp. 1–7.
[84] Smith *et al.*, "A simulation study of decoupled architecture computers," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 692–702, Aug 1986.
[85] J. E. Smith, "Decoupled access/execute computer architectures," *SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, Apr. 1982.
[86] S. Somogyi *et al.*, "Spatial memory streaming," in *ISCA*, June 2006, pp. 252–263.
[87] L. Song *et al.*, "GraphR: Accelerating Graph Processing Using ReRAM," in *HPCA*, Feb 2018, pp. 531–543.
[88] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA*, Feb 2007, pp. 63–74.
[89] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *MICRO*, Dec 1998, pp. 148–159.
[90] N. Topham *et al.*, "Compiling and optimizing for decoupled architectures," in *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Dec 1995, pp. 40–40.
[91] S. P. Vander Wiel and D. J. Lilja, "A compiler-assisted data prefetch controller," in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*, Oct 1999, pp. 372–377.
[92] H. Wei *et al.*, "Speedup graph processing by graph ordering," in *SIGMOD*, New York, NY, USA, 2016, p. 1813–1828.
[93] T. F. Wenisch *et al.*, "Practical off-chip meta-data for temporal memory streaming," in *HPCA*, Feb 2009, pp. 79–90.
[94] C. Wu *et al.*, "PACMan: Prefetch-Aware Cache Management for high performance caching," in *MICRO*, Dec 2011, pp. 442–453.
[95] H. Wu *et al.*, "Efficient metadata management for irregular data prefetching," in *ISCA*, New York, NY, USA, 2019, pp. 449–461.
[96] H. Wu *et al.*, "Temporal prefetching without the off-chip metadata," in *MICRO*, New York, NY, USA, 2019, pp. 996–1008.
[97] M. Yan *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *MICRO*. New York, NY, USA: ACM, 2019, pp. 615–628.
[98] C.-L. Yang and A. R. Lebeck, "A programmable memory hierarchy for prefetching linked data structures," in *ISHPC*. London, UK, UK: Springer-Verlag, 2002, pp. 160–174.
[99] X. Yu *et al.*, "IMP: Indirect memory prefetcher," in *MICRO*, Dec 2015, pp. 178–190.
[100] D. Zhang *et al.*, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *ASPLOS*, New York, NY, USA, 2018, pp. 593–607.
[101] M. Zhang *et al.*, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *HPCA*, Feb 2018, pp. 544–557.
[102] W. Zhang *et al.*, "Accelerating and adapting precomputation threads for effcient prefetching," in *HPCA*, Washington, DC, USA, 2007, pp. 85–95.
[103] Y. Zhang *et al.*, "Optimizing ordered graph algorithms with graphit," in *CGO*, New York, NY, USA, 2020, p. 158–170.
[104] H. Zhou, "Dual-core execution: Building a highly scalable single-thread instruction window," in *PACT*, Washington, DC, USA, 2005, pp. 231–242.