Optimizing Emerging Graph Applications Using Hardware-Software Co-Design

by

Nishil Talati

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Associate Professor Ronald G. Dreslinski, Co-Chair
Professor Trevor N. Mudge, Co-Chair
Professor Saman Amarasinghe, MIT
Associate Professor Hun-Seok Kim
Associate Professor Danai Koutra

Nishil Talati
talatin@umich.edu
ORCID iD: 0000-0002-2457-4119

DEDICATION

Dedicated to my wonderful partner Shruti Nagaraja, and our parents Hena Talati, Rakesh Talati, Malini Nagaraja, and D. A. Nagaraja.

ACKNOWLEDGMENTS

I would like to start by thanking my advisors Prof. Ronald Dreslinski and Prof. Trevor Mudge for their constant support throughout my PhD. They have not only offered me an excellent technical advice, but showed me how to become a mature researcher, and a good citizen of the research community. I am thankful to my advisors who offered me freedom and independence to choose my own research direction throughout my PhD. This independence made me grow as a researcher. More importantly, I got to learn from them about how to become a responsible academic researcher in so many ways that are hard to list here. So, thank you both Ron and Trev for taking me as a PhD student and shaping me into who I am today.

In addition to my advisors, I have also had an opportunity to collaborate and learn from Prof. Danai Koutra, Prof. Saman Amarasinghe, Prof. Hun-Seok Kim, Prof. David Blaauw, Prof. Alex Bronstein, Prof. Reetuperna Das, Prof. Scott Mahlke, Prof. Todd Austin, and Prof. Michael O'Boyle. Additionally, I am also deeply thankful to my Master's thesis advisor Prof. Shahar Kvatinsky and mentor Mr. Ronny Ronen, and my undergraduate research mentor Prof. Pravin Mane who got me excited about research and pursuing a PhD degree. Their invaluable support also help me grow as a researcher in my early research career.

I would like to acknowledge and thank the funding agencies that made my research possible. These agencies include Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory (AFRL), and United States—Israel Binational Science Foundation (BSF).

I am extremely grateful to my colleagues, fellow students, with whom I have enjoyed conducting research, including reading new papers, brainstorming ideas, developing simulators, and writing/presenting our research outcomes. Specifically, I would like to acknowledge Haojie Ye, Yuhan Chen, Kuan-Yu Chen, Sanketh Vedula, Di Jin, Armand Behroozi, Leul Belayneh, Tarunesh Verma, Yichen Yang, Ajay Brahmakshatriya, Daniel Liu, Yichao Yuan, Agreen Ahmadi, Brandon Nguyen, Kuba Kaszyk, Chris Vasiladiotis, and many more. I have had an excellent time working with and learning from these individuals, so thank you all.

In addition to the individuals who worked with me on research topics, I am also extremely thankful to several other senior PhD students who served me as mentors to help me navigate my academic life. Specifically, I am thankful to Arun Subramaniyan, Abraham Addisie, Vaibhav Gogate, Zelalem Aweke, Ofir Weisse, Subhankar Pal, Aporva Amarnath, Vidushi Goyal, Tanvir

Ahmed Khan, and Akshitha Sriraman. All of them have had a positive influence in shaping my PhD journey.

For making my PhD journey as enjoyable as possible, I would like to thank my friends Renuka Kumar, Tarunesh Verma, Vidushi Goyal, Armand Behroozi, Kimiya Shehzadi, Leul Belayneh, Heewoo Kim, Sanjay Singapuram, Vikalp Aggarwal, Anish Shah, Marina Minkin, Stephan van Schaik, and several others from Ann Arbor. Outside of Ann Arbor too, I have had several friends from back in my high school, undergrad, and master's degrees who have positively contributed to my well-being, and I am forever thankful to have such excellent friends.

Finally, but most importantly, I would like to express my deepest gratitude to my family. First, I would like to thank my parents Hena and Rakesh Talati for raising me to become the person I am today. My parents recognized my passion in Science and Technology very early on in my life and made me believe that anything is possible with hard and honest work. My parents offered me a healthy environment to best follow my passion. But most importantly, they taught me that being an honest and responsible citizen of the world, no matter the cost, is the single most important thing, which I will remember for the rest of my life. Without their support and nurturing, I would be living a very different life right now.

I am extremely thankful to my partner Shruti Nagaraja who has supported me both personally and professionally. She has always put my interests and ambitions before her professional goals. Without her sacrifices and support, it'd have taken me much longer to finish my PhD degree. Despite the long working hours before the deadlines, and never-ending debugging sessions, she has always been understanding of my obligations. Truly, I could not have asked for a better partner, so thank you Shruti for always being there for me. I am also grateful to Caliber, our German Shepherd pup, who filled our lives with joy, and made my PhD journey so much more enjoyable. Finally, I would like to express my gratitude to our parents Malini and D. A. Nagaraja, and brother Shashank Nagaraja for constantly providing me support throughout my PhD journey.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

A graph is a ubiquitous data structure that models entities and their interactions through the collections of nodes and edges. It is widely employed in several important application domains ranging from social media, navigation tools, search engines, physics simulations, and biology. Despite its prevalence, the performance of graph algorithms on commercial platforms is limited. This is mainly due to the irregular memory accesses and convoluted control flow instructions used in graph algorithms while accessing large volumes of graph data (with billions of nodes/edges). Therefore, there is a pressing need for optimizing the performance of graph workloads.

In this thesis, I present a systematic optimization study of a variety of graph workloads running on both static and dynamic graphs. At a high level, I first analyze the unique challenges and execution bottlenecks of the state-of-the-art graph software frameworks running on commercial hardware platforms. I then use the insights obtained from this analysis to propose design optimizations catered to the unique workload characteristics of a diversity of graph workloads.

Specifically, first, I propose Prodigy—a hardware-software co-design solution to improve the performance of traditional graph processing algorithms (*e.g.,* PageRank and SSSP) on multi-core CPUs. Second, I present an in-depth study of random walk–based graph learning algorithms on temporal graphs (a type of dynamic graph). Specifically, this study delivers high-performance, open-source CPU and GPU implementations of important graph learning applications, conducts a detailed performance analysis, and makes recommendations for future optimizations. Third, I showcase NDMiner—a domain-specialized Near Data Processing (NDP) architecture that significantly improves the performance of Graph Pattern Mining (GPM) workloads. Last, I present Mint—a novel hardware accelerator architecture and an accompanying programming model for efficiently mining motifs in temporal graphs.

CHAPTER 1

Introduction

1.1 Prevalence of Graph-based Applications

A graph data structure effectively models the interactions between entities through nodes and edges. As an intuitive example, user interactions on social media networks today are represented in terms of graphs, where users represent nodes, and user interactions represent edges. Twitter models connections, tweets/retweets, likes, comments, and messages in the form of graph, where users are nodes and their interactions are edges [161]. In addition to social media networks, graphs are used to model road networks [140], communication networks [120, 136, 138], and citation networks [76, 137]. Beyond Computer Science, graphs model protein and drug interactions [53, 247] computational chemistry [63], and high-energy physics [49].

A variety of graph algorithms extract useful information out of graph data structures. This includes finding possible connections between nodes, identifying important graph nodes, establishing distance between a pair of nodes, detecting community of nodes, and mining uniquely shaped subgraphs. These graph algorithms enable a wide range of applications. For example, search engines such as Google heavily rely on analyzing the interactions between web pages through a webgraph to recommend user query results [187]. Social media companies such as Twitter analyze interactions between online content and users to offer improved recommendations [127, 169]. Navigation tools, for instance Google Maps, examine road networks to recommend shortest path from a source location to a destination [81]. Moreover, graph algorithms are used in several other real-world

applications such as bioinformatics [47], cyber-security [69, 203], spam detection [134], monitoring electrical power grids [125], and machine learning [31, 214, 259]. In sum, graph-based applications are prevalent.

1.2 Motivation

Due to their prevalence, speeding up graph workloads results in a significant positive impact in improving the quality of day-to-day services. However, an efficient execution of graph workloads face two major challenges on modern hardware platforms: (a) convoluted algorithmic traversal over graph data structures, and (b) large volumes of today's real-world graph datasets.

Because real-world graphs are sparse in nature (*i.e.,* no interactions exist between most pair of nodes), sparse representations are often used to store graphs in memory to conserve space by storing only non-zero values. Compressed Sparse Row (CSR) is one such space-efficient technique for representing 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. While CSR representation saves space, it comes at a cost of more complicated traversal algorithms, where edge list can only be accessed after reading data from the offset list. Therefore, graph algorithms frequently use **data-dependent irregular memory accesses**.

Furthermore, graph algorithms often use memory access-dependent control flow instructions. For example, the Breadth-First Search (BFS) graph algorithm uses a per-vertex flag to indicate whether a node is previously visited or not. This translates into a **data-dependent branch instruction**. Because the outcomes of these branch instructions depend on the data stored in memory that change with the progression of the algorithm, these branch instructions are extremely challenging to correctly predict by the modern branch predictors.

To make things worse, modern graph datasets have billions to trillions of nodes/edges. The sheer volume of graph datasets does not allow the graph information to be stored on capacity-limited

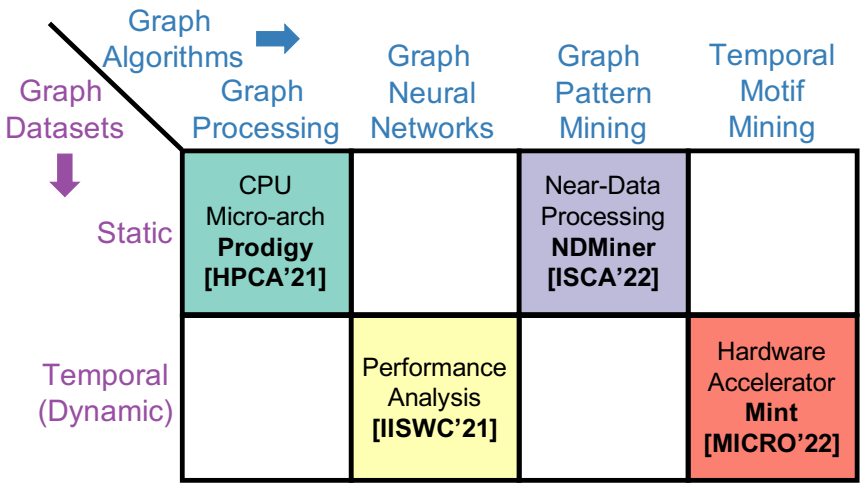


Figure 1.1: Contributions presented in this dissertation.

CPU/GPU caches, leading to frequent main memory/DRAM accesses. DRAM accesses are costly from both performance and energy perspectives [197].

To summarize, the large volumes of real-world graph data coupled with convoluted memory and control flow behavior of graph algorithms render optimization techniques (*e.g.,* on-chip caching and prefetching) on traditional hardware platforms ineffective. This is because commercial hardware platforms are designed to optimize dense computations with regular memory accesses. As a result, the performance of large-scale graph workloads is severely limited on commercial hardware platforms [11, 24, 252]. Therefore, there is an urgent need for optimizing the performance of graph workloads that can benefit a wide range of applications discussed above.

1.3 Dissertation Contributions and Organization

The goal of this dissertation is to address the performance limitations of commercial hardware platforms, and propose hardware-software co-designed solutions to enable high-performance graph workload execution. Figure 1.1 shows the contributions of this dissertation. At a high level, these contributions span a variety of graph algorithms, *i.e.,* traditional graph processing (*e.g.,* PageRank and SSSP), Graph Neural Networks (GNN, *e.g.,* link prediction and node classification), Graph Pattern Mining (GPM, *e.g.,* triangle counting), and temporal motif mining (*e.g.,* cycle detection),

running on both static and temporal graph datasets. More specifically, the individual contributions of each of these works are detailed below.

- **Prodigy (Chapter 3)**. Irregular workloads (including traditional graph 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 work presents 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 traditional 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×.

- **Analyzing temporal graph learning (Chapter 4)**. Machine learning on graph data has gained significant interest because of its applicability to various domains ranging from product recommendations to drug discovery. While there is a rapid growth in the algorithmic community, the computer architecture community has so far focused on a subset of graph learning algorithms including Graph Convolution Network (GCN), and a few others. In this work, we study another, more scalable, graph learning algorithm based on *random walks*, which operates on dynamic input graphs and has attracted less attention in the architecture community compared to GCN. We propose high-performance CPU and GPU implementations of two important graph learning tasks, that cover a broad class of applications, using random walks on continuous-time dynamic graphs: link prediction and node classification. We show that the resulting workload exhibits distinct characteristics, measured in terms of irregularity, core and memory utilization, and cache hit rates, compared to graph traversals, deep learning, and GCN. We further conduct an in-depth performance analysis focused on both algorithm and hardware to guide future software optimization and architecture exploration. The algorithm-focused study presents a rich trade-off space between algorithmic performance and runtime complexity to identify optimization opportunities. We find an optimal hyperparameter setting that strikes balance in this trade-off space. Using this setting, we also perform a detailed microarchitectural characterization to analyze hardware behavior of these applications and uncover execution bottlenecks, which include high cache misses and dependency-related stalls. The outcome of our study includes recommendations for further performance optimization, and open-source implementations for future investigation.

- **NDMiner (Chapter 5)**. Graph Pattern Mining (GPM) algorithms mine structural patterns in graphs. The performance of GPM workloads is bottlenecked by control flow and memory stalls. This is because of data-dependent branches used in set intersection and difference operations that dominate the execution time.

This work first conducts a systematic GPM workload analysis and uncovers four new observations to inform the optimization effort. First, GPM workloads mostly fetch inputs of costly set operations from different memory banks. Second, to avoid redundant computation, modern GPM workloads employ symmetry breaking that discards several data reads, resulting in cache pollution and wasted DRAM bandwidth. Third, sparse pattern mining algorithms perform redundant memory reads and computations. Fourth, GPM workloads do not fully utilize the in-DRAM data parallelism.

Based on these observations, this work presents NDMiner, a Near Data Processing (NDP) architecture that improves the performance of GPM workloads. To reduce in-memory data transfer of fetching data from different memory banks, NDMiner integrates compute units to offload set operations in the buffer chip of DRAM. To alleviate the wasted memory bandwidth caused by symmetry breaking, NDMiner integrates a *load elision unit* in hardware that detects the satisfiability of symmetry breaking constraints and terminates unnecessary loads. To optimize the performance of sparse pattern mining, NDMiner employs *compiler optimizations* and maps reduced reads and composite computation to NDP hardware that improves algorithmic efficiency of sparse GPM. Finally, NDMiner proposes a new *graph remapping* scheme in memory and a *hardware-based set operation reordering* technique to best optimize bank, rank, and channel-level parallelism in DRAM. To orchestrate NDP computation, this work presents design modifications at the host ISA, compiler, and memory controller. We compare the performance of NDMiner with state-of-the-art software and hardware baselines using a mix of dense and sparse GPM algorithms. Our evaluation shows that NDMiner significantly outperforms software and hardware baselines by 6.4× and 2.5×, on average, while incurring a negligible area overhead on CPU and DRAM.

- **Mint (Chapter 6)**. A variety of complex systems, including social and communication networks, financial markets, biology, and neuroscience are modeled using temporal graphs that contain a set of nodes and directed timestamped edges. Temporal motifs in temporal graphs are generalized from subgraph patterns in static graphs in that they also account for

edge ordering and time duration, in addition to the graph structure. Mining temporal motifs is a fundamental problem used in several application domains. However, existing software frameworks offer sub-optimal performance due to high algorithmic complexity and irregular memory accesses of temporal motif mining.

This work presents Mint—a novel accelerator architecture and a programming model for mining temporal motifs efficiently. We first divide this workload into three fundamental tasks: search, book-keeping, and backtracking. Based on this, we propose a task–centric programming model that enables decoupled, asynchronous execution. This model unlocks massive opportunities for parallelism, and allows storing task context information on-chip. To best utilize the proposed programming model, we design a domain-specific hardware accelerator using its data path and memory subsystem design to cater to the unique workload characteristics of temporal motif mining. To further improve performance, we propose a novel optimization called search index memoization that significantly reduces memory traffic. We comprehensively compare the performance of Mint with state-of-the-art temporal motif mining software frameworks (both approximate and exact) running on both CPU and GPU, and show 9×–2576× benefit in performance.

1.4 Impact Statement

The current and future impact of this dissertation work is summarized below.

- **Prodigy.** This work opened new doors in defining novel hardware-software interfaces that has and will inspire significant additional follow-up works across the community. This work showed how to enable system-wide optimization effort based on a customized hardware-software contract such as Data Indirection Graph (DIG) representation that can be widely adopted in several academic and industrial products. Prodigy was recognized as the best paper at a top architecture conference – the 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2021).

- **Benchmarking of temporal graph learning.** This work is a result of a unique collaboration between computer architecture, software design, and data mining research groups. While Graph Convolutional Network (GCN)–type algorithms are well-studied and optimized in the hardware community, this work introduces a more scalable graph learning algorithm to the architecture community, *i.e.,* random walks. This work presents both high-performance open-source implementations of key graph learning tasks on both CPU and GPU, and conducts a detailed performance analysis study for finding further optimization opportunities. The open-sourced tool-chain is currently used, and continue to be used, in the community to learn further optimizations (*e.g.,* sparsification of graph neural networks).

- **NDMiner.** While most prior works either focused fully on designing a custom hardware accelerator architectures or generic near-data processing systems to accelerate the workload of graph pattern mining, NDMiner showed how to combine the unique benefits of both near-data processing and domain specialization. Specifically, NDMiner finds new unique insights about the workload characteristics of graph pattern mining, and employs these findings to further optimize the near-data processing architecture. This architecture aims an extremely low-cost area overhead to the commercial DRAM DIMMs. This design philosophy can be adopted widely in the community to optimize various types of workloads.

- **Mint.** This is the first work that characterizes bottlenecks and optimizes the execution of temporal motif mining. Mint is a three-part design that introduces a new programming model, a hardware accelerator architecture, and a domain-specific optimization. Mint divides the workload in terms of tasks, and proposes a task-centric asynchronous programming model to enable massive opportunities of parallelism and improved hardware utilization. To best utilize this programming model, a Mint presents a versatile accelerator architecture to mine any arbitrary temporal motif, and search index memoization optimization to significantly reduce the memory traffic.

CHAPTER 2

Background: Graph Data Structures and Algorithms

Graphs represent various types of relationships. For example, a professional network graph like LinkedIn represents different professionals as nodes and their interactions (*e.g.,* collaboration between a pair of individuals) as edges. There are various types of representing graphs. From a temporal standpoint, a graph can be either static or dynamic. Dynamic graphs capture time-evolving relationships among its vertices by adding, deleting, or changing their nodes and edges. These dynamic graphs can be made *static* by rendering two nodes connected if any interaction took place between a pair of nodes. In this chapter, I discuss both static and dynamic graph datasets, their in-memory representations, and unique graph traversal algorithms.

2.1 Static Graph

Static graphs can be represented in terms of an adjacency matrix, where each dimension of this matrix is equal to the number of vertices in a graph. A non-zero value at row i and column j would represent an edge between nodes i and j. This non-zero value can either be an identity (in the case of unweighted graph) or a weight value for weighted graphs. Conversely, a value of zero at row k and l means an absence of edge between nodes k and l. Because most nodes in real-world graphs are not connected with most other nodes, a typical adjacency matrix of a graph is extremely sparse in nature. In other words, most values of this matrix store zeros, leading to storage inefficiency.

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:

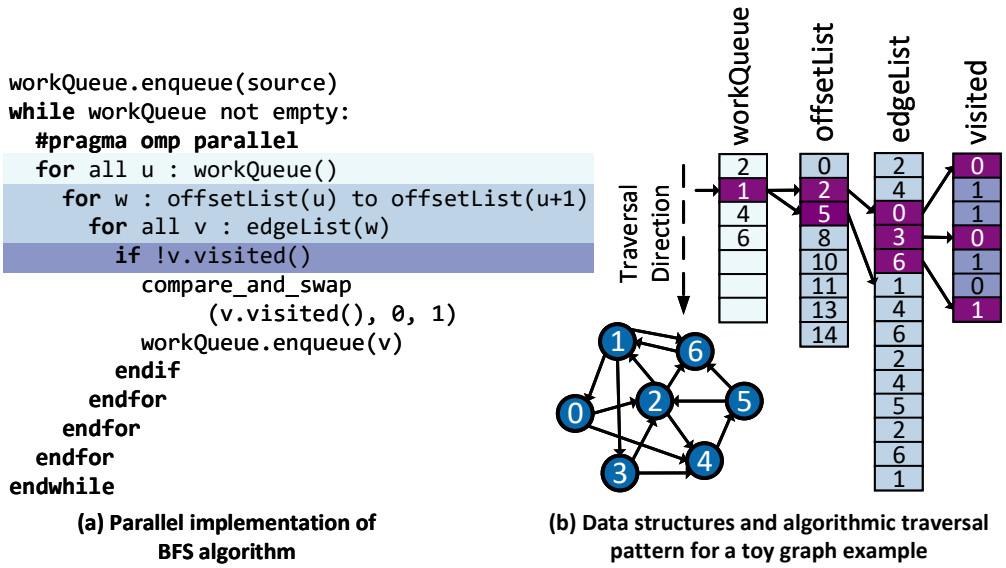


Figure 2.1: BFS algorithm: (a) pseudo-code for a parallel implementation of BFS, and (b) a toy example of BFS traversal on a graph stored in a compressed sparse row (CSR) format.

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. 2.1(b).

2.2 Temporal Graph

A temporal graph is a type of dynamic graph that represents network interaction in terms of timestamped edges between a pair of nodes. A *temporal edge* is defined to be a timestamped directed edge between an ordered pair of nodes. A *temporal graph* is then defined as a collection of temporal edges. Formally, a temporal graph G is a collection of tuples $G = \{(u_i, v_i, t_i)\}_{i=1}^{m}$, where u_i and v_i are source and destination nodes of the edge (u_i, v_i), and $t_i \in \mathbb{R}^+$ is a timestamp of the edge. I assume that the timestamps of the edges in the temporal graph G are unique[1]. Strictly speaking, G is a *multi-digraph* as (a) the edges are directed, (b) there might be many edges between a pair of nodes, each one with a different timestamp.

Depending on the traversal algorithm, a temporal graph can be stored in many different ways

[1]This assumption comes without loss of generality [156, 192].

in memory. This includes extending the CSR structure to store timestamped edge list in place of edge list or storing a chronologically ordered edge list and additional index array to this edge list indicating incoming/outgoing edge IDs from each node. More details on the exact representation are accompanied with the algorithmic/implementation details of specific problems.

2.3 Traditional Graph Processing Algorithms

In this section, I use Breadth-First Search (BFS) as a representative traditional graph processing algorithm and discuss its data structures and algorithmic traversal in detail. In addition to BFS, traditional graph processing algorithms include PageRank, Single-Source Shortest Path (SSSP), Betweenness Centrality (BC), Connected Components (CC), *etc.* 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.

Fig. 2.1(a) describes the traversal pattern of the BFS algorithm. I 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. 2.1(b).

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

Algorithm 1 Pseudocode for Triangle Counting (TC)

```
1:  procedure GPM_TC(G, P)                                  ▷ G: graph, P: pattern (triangle in this case)
2:      num_trialges = 0;
3:      for u ∈ V do                                        ▷ V: Vertex set of G, {u}: single-vertex embedding
4:          N_u = G.out_neighbors(u);                       ▷ Neighborhood expansion
5:          for v ∈ N_u do                                  ▷ {u, v}: two-vertex embedding
6:              if v ≥ u then                               ▷ Neighborhood filtration for symmetry breaking
7:                  break;
8:              N_v = G.out_neighbors(v);                   ▷ Neighborhood expansion
9:              N_uv = INTERSECTION(N_u, N_v);              ▷ Set intersection
10:             for w ∈ N_uv do                             ▷ {u, v, w}: three-vertex embedding
11:                 if w ≥ v then                           ▷ Intersection filtration for symmetry breaking
12:                     break;
13:                 num_triangles++;
14:     return num_triangles;
15:
16: procedure INTERSECTION(SetA, SetB)                      ▷ Set intersection procedure
17:     intersection_result = [];
18:     while i < SetA.size() and j < SetB.size() do
19:         if SetA[i] < SetB[j] then                       ▷ Data-dependent control flow
20:             i++;
21:         else if SetA[i] > SetB[j] then                  ▷ Data-dependent control flow
22:             j++;
23:         else                                            ▷ SetA[i] = SetB[j]
24:             intersection_result.insert(SetA[i]);
25:             i++; j++;
26:     return intersection_result;
```

2.4 Graph Pattern Mining (GPM) Algorithms

The problem of GPM finds all *unique* subgraphs (also known as *embeddings*) in an input static graph that are *isomorphic* to a given input pattern. A pattern is isomorphic to a subgraph if there exists a one-to-one mapping of all the vertices and edges between the pattern and a subgraph. Permuting vertices and edges of a given subgraph generates equivalent subgraphs, also called *automorphic* embeddings.

GPM algorithm. It uses a search tree to enumerate embeddings in an input graph G matching a user-defined pattern P. From all single-vertex subgraphs, the tree visits one node/edge at a time to expand the embedding in each level. The isomorphism test is performed after all the embeddings reach a desired tree depth (*i.e.,* size of the embedding), where the number of vertices in expanded subgraphs matches the number of vertices in P. Following the terminology in Peregrine [104], GPM algorithms can be broadly classified in two categories: (a) pattern-oblivious, and (b) pattern-aware. Peregrine concludes that pattern-aware GPM algorithms outperform their pattern-oblivious counterparts by eliminating redundant computations. Therefore, I use pattern-aware algorithms.

Algorithm 1 shows the pseudo-code of triangle counting. Starting from single-vertex embeddings shown in line 3, the algorithm expands them to two-vertex embeddings (line 5) by finding their outgoing neighbors (line 4). These embeddings are further expanded by finding common neighbors amongst its vertices. The *intersection* (line 9) of vertex neighborhood sets is employed to find common neighbors. With this expansion, embeddings isomorphic to a desired pattern (triangle) are found. Notably, the pattern-aware GPM algorithms only find embeddings isomorphic to P. In other words, the isomorphism test is encoded into the algorithms, precluding the necessity for explicit isomorphism tests after search tree expansion.

GPM algorithm optimizations. Pattern-specific GPM algorithms enable several performance optimizations. I briefly discuss (a) optimal schedule, and (b) symmetry breaking restrictions optimizations used in this work, and refer the reader to prior works [41, 42, 104, 105, 159, 160, 232] for other optimizations. The *schedule* of a GPM algorithm determines the order at which each vertex of a pattern is searched. While searching for patterns, *restrictions* are applied to vertex IDs to avoid redundant computation. This is also known as symmetry breaking/search tree pruning as it avoids expanding unnecessary tree branches that cannot lead to P.

GraphPi [232] shows that there is a large design space to find the optimal schedule and restrictions that can affect performance by up to three order of magnitude. This is because of the schedule and restrictions define the size and pruning level of the search tree that lead to significant performance differences in large-scale graphs. Lines 6 and 11 show instances of *filteration operation* applied to triangle counting for search tree pruning. Because a triangle is a symmetric pattern, the order at which the vertices are searched makes no difference, leading to only one schedule. However, large asymmetric patterns can benefit significantly from schedule optimizations. My work adopts optimal schedule and restrictions from GraphPi.

2.5 Temporal Random Walk Algorithm and Graph Representation Learning

In this section, I discuss a set of algorithms that perform link prediction and node classification based on temporal random walk on the graph. Specifically, the workload first generates the temporally-valid walks to characterize the structure of the subgraph centering around each node, and then leverages word2vec to encode it into the low-dimensional Euclidean space as the node embeddings (§2.5.1). Then, depending on specific downstream tasks, the workload feeds the derived node embeddings into the neural network architectures, and trains the model to minimize the training loss (§2.5.2). Finally, training is performed. The notations used in this section are defined in §4.2.

2.5.1 Temporal Random Walk based Representation

2.5.1.1 Temporal Random Walk

I follow an earlier algorithmic work, CTDNE [179], to deploy the workload. Specifically, for a node v in graph G, our workload leverages a set of temporally-valid walks originating from v as the characteristic features to derive the embeddings. As mentioned in Definition 4.2.2, the temporally-valid walks reflect the reachability of nodes following the graph structure over time, which further reflects how a node v dynamically interacts with its neighbors in the graph.

I leverage temporal random walk to collect the neighborhood information for each node $v \in G$. In typical temporal random walks, the nodes along the walks are chosen randomly without a specific destination as long as the associated timestamps are increasing. The transitional probability $p(v|u)$ is denoted as $p(v|u) = \frac{1}{|\mathcal{N}_u|}$, where $\mathcal{N}_u$ denotes the set of nodes that are reachable from u following the connected edges. Thus, as long as $\mathcal{N}_u$ for $u \in \mathcal{V}$ is computed efficiently, the temporal walks can be collected efficiently. I detail the implementation of $\mathcal{N}_u$ in §4.4.1. As an example shown in Fig. 2.2, the random walker currently reaches node v following the edge with timestamp 1. The next node it reaches would be either node x or y with equal probability 0.5.

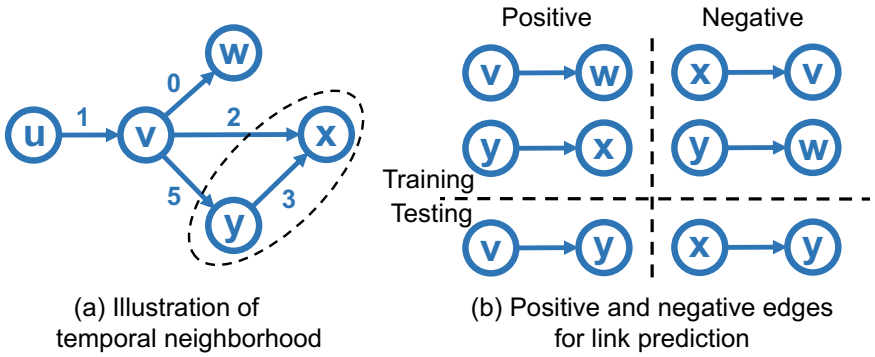


Figure 2.2: Illustration of temporal neighborhood and positive/negative edges. At timestamp 1, the random walker reaches node v, then the set of nodes $\{x, y\}$ forms the temporal neighbors of node v.

While a typical transitional probability marks an efficient way to gather temporal walks, it fails to incorporate the temporal continuity. Again, in the example shown in Fig. 2.2, an edge from node v to x appears immediately after the edge from the source node u to v. Compared to node y that appears later in time, node x is more correlated with v.

In order to capture this notion of temporal continuity in the graph dynamics, I follow Jin *et al.* [107] to model the transition probability using the softmax function:

$$Pr[v|u] = \frac{\exp\left(-\tau(u,v)/r\right)}{\sum_{i \in \mathcal{N}_u} \exp\left(-\tau(u,i)/r\right)}, \tag{2.1}$$

where $\tau(u, v)$ denotes the timestamp associated with the edge u, v in the graph, and r is the normalization term that denotes the total range of timespan.

With the transitional probability, our workload performs $|\mathcal{W}|$ walks with lengths L per node, and collect them as the features to describe each individual node in the graph $\mathcal{W}_u = \{v_1, v_2, \cdots, v_L\}$. Next, I describe the derivation of node embeddings based on these walks.

2.5.1.2 Node Embedding

Given the set of temporal walks as features per node, our workload then leverages the skip-gram model [85, 179] to learn the node embeddings, where the objective function is

$$\max_f \log Pr(\mathcal{W}_u | f(u)), \tag{2.2}$$

where $f(u)$ denotes the embedding for node u to optimize. To solve Equation (2.2), I assume conditional independence between nodes in $\mathcal{W}_u$, generating a relaxed objective

$$Pr(\mathcal{W}_u|f(u)) = \prod_{v \in \mathcal{W}_u} Pr(v|f(u)), \tag{2.3}$$

where Pr denotes the softmax function (Eq. (2.1)). As the output, our workload generates the embedding function $f = G \rightarrow \mathbb{R}^d$ for each node $u \in G$. For our implementation I leverage the word2vec [167] framework.

2.5.2 Downstream Tasks

Given the d-dimensional embedding vector per node, our workload leverages the feed forward neural network architecture (FNN) to perform two representative downstream tasks: link prediction and (multi-class) node classification. The parameters of FNN are updated in the training set S^{tr} and tested on the testing set S^{te}. The optimizers used for both tasks are Stochastic Gradient Descent (SGD).

Depending on the tasks, the specific network architecture and loss function adopted in our workload is given as follows.

Link Prediction. The goal of link prediction is to correctly predict the existence of edges that occur later in time based on the initial graph temporal connectivity. Our workload casts link prediction as a classification task, so that the trained FNN can distinguish edges in temporal graph G (positive edges) from the non-existing ones (negative edges). An example is shown in Fig. 2.2(b), where the goal is to predict the recent edge $e_{(v,y)}$ in the toy graph. Our workload randomly samples two early edges as the positive samples with the same number of negative edges to train the neural network. In the testing stage, the same amount of negative samples are generated as well. The embedding for edge $e_{(u,v)}$ is derived by concatenating the embedding of the source and destination nodes, *i.e.*, $f(e_{(u,v)}) = [f(u), f(v)]$ following [35].

In this task, I deploy the 2-layer FNN, where the output layer generates the probability of

classification. I use a binary cross-entropy loss function in the training stage $L = -\sum_{k=1}^{2} p_k \log q_k$, where p_k is the binary target ($\{0, 1\}$) and q_k is the output probability of the neural net, *i.e.*, $q = \text{FNN}_{\text{LP}}(f(e_+, f(e_-)))$.

Node Classification. Multi-class classification is another widely studied task, where the goal is to classify the multi-class labels of nodes in the graph. In our workload, I cast the multi-class classification task by feeding the node-wise embeddings as well as their labels to a 3-layer neural network. The output layer has $|\mathcal{C}|$ neurons, each of which indicates the probability of the input node belonging to the class $c \in \mathcal{C}$. The loss function used is negative log likelihood loss $L = -\log(q_c)$, where q_c is the output probability of a node belonging to the ground-truth class c, *i.e.*, $q = \text{FNN}_{\text{NC}}(f(u), l(u)))$, where $l(u)$ denotes the label for node u.

2.6 Temporal Motif Mining Algorithm

This section details the problem definition, and data structures and the state-of-the-art algorithm used for temporal motif mining.

2.6.1 Problem Definition

A *δ-temporal motif*[3] is defined as a sequence of l edges, $M = \{(u_i, v_i, t_i)\}_{i=1}^{l}$, that are time-ordered and occur within a δ duration, *i.e.,* $t_1 < t_2 < \ldots < t_l$ and $t_l - t_1 \leq \delta$. The problem of *temporal motif mining* is to mine occurrences of the δ-temporal motif M within a larger temporal graph G. In simple words, a δ-temporal motif is an occurrence of the sequence of edges in the graph G such that the first and last edges of this sequence occur *at most* δ time apart. It differs from the task of static motif mining in two ways: (1) in static motif mining, we are not interested in the *sequence* in which the motif's edges occur within G; (2) static motifs do not impose any constraints on the edge properties. Temporal motif mining may be interpreted as identifying subgraph isomorphisms with sequential and δ–constraints over edges.

[3] I sometimes refer to *δ-temporal motif* simply as *temporal motif* for brevity.

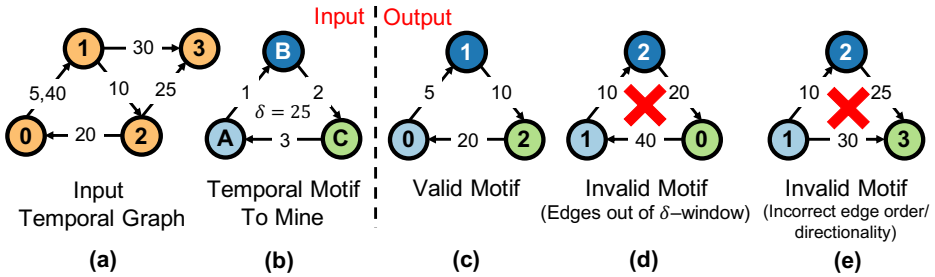


Figure 2.3: Example of $\delta-$temporal motif mining task. Depicted in (a) is the input graph, and (b) is the $\delta-$temporal motif. (c) presents a valid candidate for $\delta-$temporal motif in the input graph, (d, e) are invalid motifs due to violation of $\delta-$constraint and edge ordering, respectively.

As a simple example, consider mining of a three-cycle *δ-temporal motif* in an input temporal graph as shown in Fig. 2.3. Fig. 2.3(c) shows a valid motif because the edges between nodes 0, 1, and 2 in this motif follow edge ordering and occur within $\delta = 25$. On the other hand, Fig. 2.3(d,e) show invalid motifs either due to the δ-constraint violation or an incorrect edge order. In a static setting, however, all three motifs are valid as it does not account for edge ordering and timestamps.

2.6.2 Algorithmic Behavior

This section introduces the temporal motif mining algorithm proposed by Mackey *et al.* [156].

Data structures. The primary data structure used in this algorithm is a *temporal edge list*, stored in an array of structures. Each member of this array contains source and destination node IDs and a timestamp. Temporal edges in this array are sorted based on their timestamps. Additionally, the graph structure is stored in a compressed format to simplify retrieving incoming and outgoing edges from each node. This structure stores *indices* of temporal edges in the temporal edge list (instead of storing the temporal edges [251]). In addition to the graph structure, this algorithm stores key book-keeping information. This includes mappings between motif and graph nodes ($m2gMap[]$, $g2mMap[]$). A stack ($eStack$) of mined edge indices is used for Depth-First Search (DFS) traversal.

Algorithm. Mackey *et al.* [156] present a pattern-agnostic temporal motif mining algorithm that uses search tree exploration. Each node of the search tree matches an edge in a motif to an edge

Algorithm 2 Generic temporal motif mining algorithm [156]

procedure TEMPORALMOTIFMINING($G,\ M,\ \delta$)
  Input: temporal graph (V_G, E_G), motif (V_M, E_M), time limit δ.
  Output: temporal motifs
  // Initialize data structures: edge mapping, counters
  Initialize: $m2gMap[u] = -1\ \forall u \in V_M;\ g2mMap[u] = -1\ \forall u \in V_G$
    $eCount[u] = 0\ \forall u \in V_G, eStack = [], e_M = -1, e_G = -1, t' \leftarrow \infty$
  while true **do** ▷ Loop until all motifs found
    e_G = FINDNEXTMATCHINGEDGE() ▷ Search: find a graph edge to match
    if e_G is valid **then**
      UPDATEDATASTRUCTURES() ▷ Book-keeping: update data struct
    $e_G\ +=\ 1$
    while $e_G > |E_G|$ **or** $time(e_G) > t'$ **do** ▷ Backtrack: void previous mapping
      if $eStack$ is not empty **then**
        $e_G = eStack.pop() + 1$
        if $eStack$ is empty **then** $t' \leftarrow \infty$
        $eCount[u_G] -= 1,\ eCount[v_G] -= 1$ ▷ Reduce mapped edge cnt
        if $eCount[u_G] == 0$ **then** ▷ No edges of u_G mapped
          $u_M \leftarrow g2hMap[u_G]$
          $g2hMap[u_G] = -1,\ h2gMap[u_M] = -1$ ▷ Free u_G, u_M
        if $eCount[v_G] == 0$ **then** ▷ No edges of v_G mapped
          $v_M \leftarrow g2hMap[v_G]$
          $g2hMap[v_G] = -1,\ h2gMap[v_M] = -1$ ▷ Free v_G, v_M
      else
        return results

procedure FINDNEXTMATCHINGEDGE( ) ▷ Find a new mapping
  $(u_M, v_M) = E_M[e_M]$
  $(u_G, v_G) = m2gMap[u_M], m2gMap[v_M]$
  // Gather candidate edges to match with the next motif edge
  if $u_G \geq 0$ and $v_G \geq 0$ **then** ▷ Both u_G, v_G mapped to motif nodes
    $S \leftarrow \{e \in \mathcal{N}_{out}(u_G)/\mathcal{N}_{in}(v_G) : t_e > time(e_G)\}$ ▷ Irregular access + filter
  else if $u_G > 0$ **then** ▷ Only u_G mapped to a motif node
    $S \leftarrow \{e \in \mathcal{N}_{out}(u_G) : t_e > time(e_G)\}$ ▷ Irregular access + filter
  else if $v_G > 0$ **then** ▷ Only v_G mapped to a motif node
    $S \leftarrow \{e \in \mathcal{N}_{in}(v_G) : t_e > time(e_G)\}$ ▷ Irregular access + filter
  else ▷ Both u_G, v_G not mapped
    $S \leftarrow \{e \in E_G : t_e > time(e_G)\}$ ▷ Search space is an entire edge list
  // Return the first valid candidate edge that satisfies temporal constraints
  for each edge e in S **do**
    if e is not mapped and $time(e) < t'$ **then**
      return e

procedure UPDATEDATASTRUCTURES( ) ▷ Add a new mapping
  if $e_M == |E_M| - 1$ **then** ▷ Entire motif found
    Create a motif H from edges in $eStack$; add H to results.
  else ▷ Partial motif found
    $(u_G, v_G) \leftarrow E_G[e_G],\ (u_M, v_M) \leftarrow E_M[e_M]$
    $m2gMap[u_M] = u_G,\ m2gMap[v_M] = v_G$ ▷ Map motif node to graph node
    $g2mMap[u_G] = u_M,\ g2mMap[v_G] = v_M$ ▷ Map graph node to motif node
    $eCount[u_G] += 1,\ eCount[v_G] += 1$ ▷ Increment mapped edge cnt
    if $eStack$ is empty **then** ▷ e_G is the first matched edge
      $t' \leftarrow time(e_G) + \delta$ ▷ Upper bound on the motif's end time
    $eStack.push(e_G); e_M\ +=\ 1$

in the graph. Starting from the first edge (root node of the search tree), the algorithm iterates over edges of the temporal motif in a chronological order to find one match at a time, following a DFS tree traversal. Upon matching each edge, book-keeping information is updated.

Algorithm 2 presents this in detail. The outer while loop iterates over edges in an input motif. For

each edge in a motif, the FINDNEXTMATCHINGEDGE() function tries to match a corresponding edge in a graph. By $\mathcal{N}_{out}(u)$ and $\mathcal{N}_{in}(u)$, I denote the list of outgoing and incoming edges of a node u, respectively. If a valid match is found, book-keeping structures are updated. Otherwise, a backtracking procedure voids previous matches using a stack following a DFS traversal order, and this process is repeated until all motifs are found. This algorithm performs most of its work in finding an edge to map. As shown in the algorithm, this procedure also takes into account whether or not either source and/or destination node of the motif edge have been mapped, and edge orderings to reduce the search space.

CHAPTER 3

Improving The Memory Latency of Data-Indirect Irregular Workloads

This is a collaborative work with K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. Magnus Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, R. Dreslinski.

Sparse irregular algorithms are widely deployed in several application domains including social networks [169, 187], online navigation systems [81], machine learning [89], and genomics [14, 67]. 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 [20, 109, 116, 171, 271], are inadequate.

There is a class of prefetchers [17, 48, 54, 66, 98, 112, 216, 279] 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.

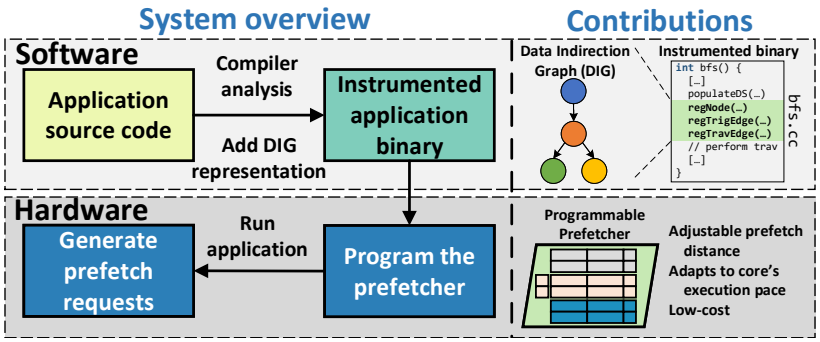


Figure 3.1: Overview of our design and contributions. Prodigy software efficiently communicates key data structures and algorithmic traversal patterns, encoded in the proposed compact representation called the Data Indirection Graph (DIG), to the hardware for informed prefetching.

Hardware prefetchers rely on capturing memory access patterns using explicit programmer support [10, 11], learning techniques [198], and intelligent hardware structures [282]. Limitations of these approaches include their limited applicability to a subset of data structures and indirect memory access patterns [11, 24, 282] or high complexity and hardware cost to support generalization [10, 198]. While software prefetching [12] can exploit static semantic view of algorithms, it lacks dynamic run-time information and struggles to maintain prefetch timeliness.

In this work, 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. 3.1 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.

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 [26] with five real-world large-scale data sets from [57, 139], two sparse linear algebra algorithms from the HPCG benchmark suite [61], and two computational fluid dynamics algorithms from the NAS

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

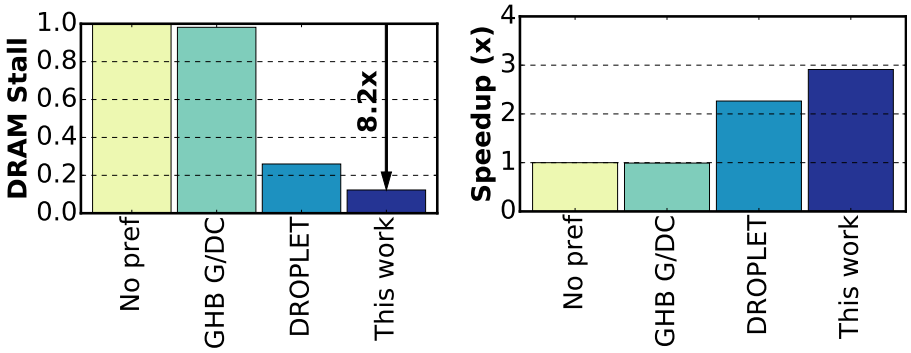


Figure 3.2: Reduction in memory stalls and speedup of different approaches normalized to a non-prefetching baseline for the PageRank algorithm on the `livejournal` data set.

parallel benchmark suite [19]. 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 [282], Ainsworth and Jones' [10, 11], DROPLET [24], and software prefetching [13].

Fig. 3.2 presents a highlight of performance benefits of Prodigy on the PageRank algorithm running on the `livejournal` data set [139]. 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 [24] which only prefetches a subset of data structures. §3.5 presents further comparisons with [10–12, 282]. 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 [282], Ainsworth and Jones' prefetcher [11], and DROPLET [24] 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 [70], state-of-the-art hardware frequently employs specialized accelerators for key applications. With

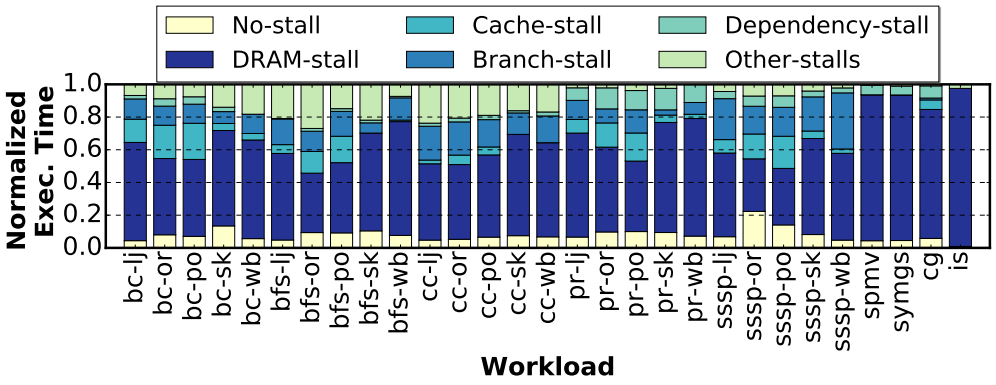


Figure 3.3: Normalized execution time of irregular workloads, without prefetching, broken down into: no-stall, and stalls due to DRAM, cache, branch mispredictions, data dependencies, and others. *The goal of this work is to reduce the DRAM stalls (dark blue portion of the bar).*

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 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.

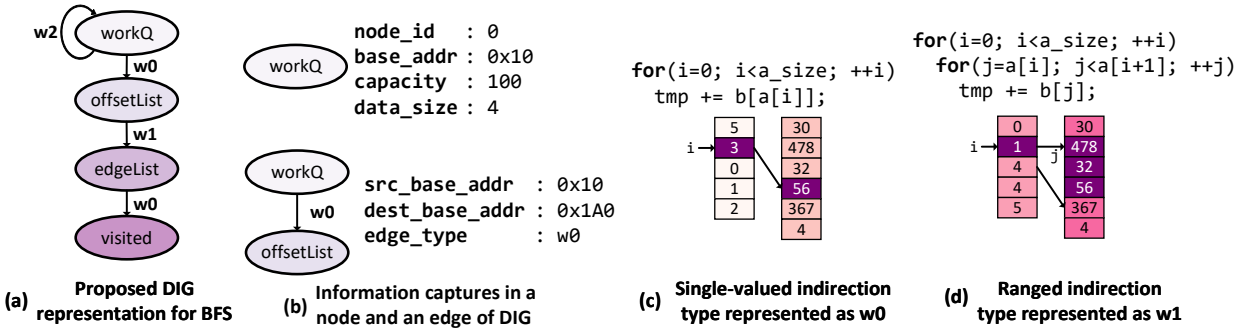


Figure 3.4: Proposed Data Indirection Graph (DIG) representation—(a) example representation for BFS, (b) data structure memory layout and algorithmic traversal information captured by a DIG node and a weighted DIG edge respectively; two unique data-dependent indirection patterns supported by Prodigy—(c) single-valued indirection, and (d) ranged indirection.

3.1 Motivation

Based on the background present in §2.1 and §2.3, 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 [242], 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. 3.3 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 §3.4. 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.

3.2 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.

3.2.1 Data Indirection Graph (DIG): A 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. 3.4 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. 3.4(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. 3.4(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 §3.3). A trigger edge represents the control flow specifying the prefetch sequence to initialize.

3.2.2 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.

3.2.2.1 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. 3.5 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,

```
1: int BFS(FILE* inputGraph, vtxID source)
2: {
3:   Graph g = readGraph(inputGraph);
4:   queue<vtxID> workQueue(g.numNodes());
5:   vtxID** offsetList = (vtxID**) malloc(g.numNodes()+1);
6:   vtxID*  edgeList   = (vtxID*)  malloc(g.numEdges());
7:   vtxID*  visited    = (vtxID*)  malloc(g.numNodes());
8:   populateDataStructures(g, offsetList, edgeList, visited);
9:   registerNode(&workQueue,  g.numNodes(),   4,  0);
10:  registerNode(offsetList,  g.numNodes()+1, 4,  1);
11:  registerNode(edgeList,    g.numEdges(),   4,  2);
12:  registerNode(visited,     g.numNodes(),   4,  3);
13:  registerTravEdge(&workQueue, offsetList,  w0);
14:  registerTravEdge(offsetList, edgeList,    w1);
15:  registerTravEdge(edgeList,   visited,     w0);
16:  registerTrigEdge(&workQueue, w2);
17:  workQueue.enqueue(source);
18:  […]
```

Figure 3.5: Annotated BFS source code to construct the DIG.

and the type of indirection (*i.e.,* `w0`/`w1` as shown in Fig. 3.4). `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 §3.3.3).

3.2.2.2 Using Compiler Analysis

Identifying indirections in non-trivial programs (*e.g.,* [26]) 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 [21, 267] 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. 3.7(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

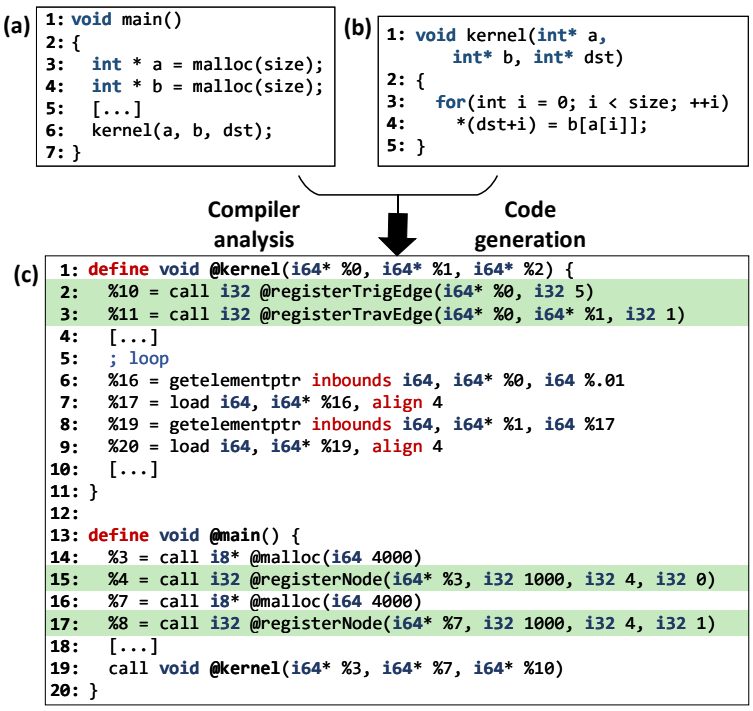


Figure 3.6: An example C program (a) and (b), translated into LLVM IR (c) and instrumented with our API calls to register DIG nodes and edges.

pseudocode for this procedure is presented in Fig. 3.7(a). Fig. 3.6(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. 3.6(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. 3.6(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. 3.6, 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 indirection can be identified similarly. For a ranged indirection from array `a` to `b` as shown in Fig. 3.4(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. 3.7(b,c), respectively.

```
for func in module:
   for inst in func:
      if isInstanceOf(inst, AllocCall):
         alloc = AllocCall(inst)
         alloc_info = {alloc.total_size, alloc_num_elements,
                                    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(ins.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 2.9a
   node_table.insert({base_ptr, base_ptr + num_elems *
                      elem_size, node_id})

def registerTravEdge(src_ptr, target_addr, edge_type):
   # note: the edge_table is depicted in Figure 2.9c
   src_base_addr = scan_node_table(src_addr_
   target_base_addr = scan_node_table(target_addr)
   if src_base_addr and target_base_addr:
      edge_table.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 3.7: Pseudocode of Prodigy's compiler analyses for (a) node identification, (b) single-valued indirection, (c) ranged indirection, and (d) runtime.

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

(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 3.8: 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.

self-edge to the trigger node). For example, the address calculations in lines 6 and 8 in Fig. 3.6(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 annotations use the same API, presented in Fig. 3.7(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.

3.2.2.3 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.

3.3 Proposed Hardware Design

3.3.1 Memory Requirements for a DIG

Fig. 3.8(a-c) show three prefetcher-local memory structures to store a DIG representation. As described in §3.2, 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.

3.3.2 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. 3.8(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`

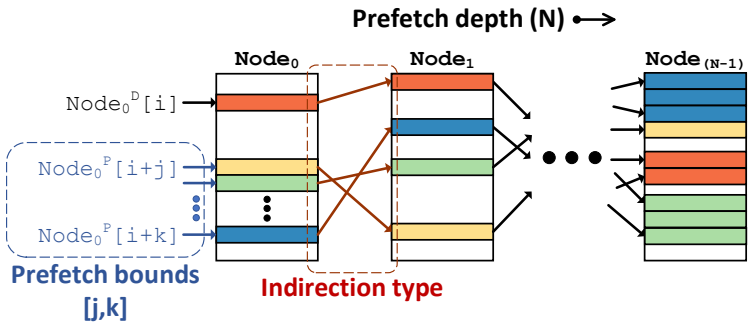


Figure 3.9: Prefetching algorithm initiates prefetch sequences between prefetch bounds j and k and advances a prefetch sequence using software-defined indirection types. The superscripts denote a demand (D) or a prefetch (P) access.

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.

3.3.3 Prefetching Algorithm

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

3.3.3.1 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. 3.9. 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. 3.8(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.*

3.3.3.2 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 §3.2.1).

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. 3.4(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. 3.4(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.

3.3.4 Hardware Flow of Prodigy

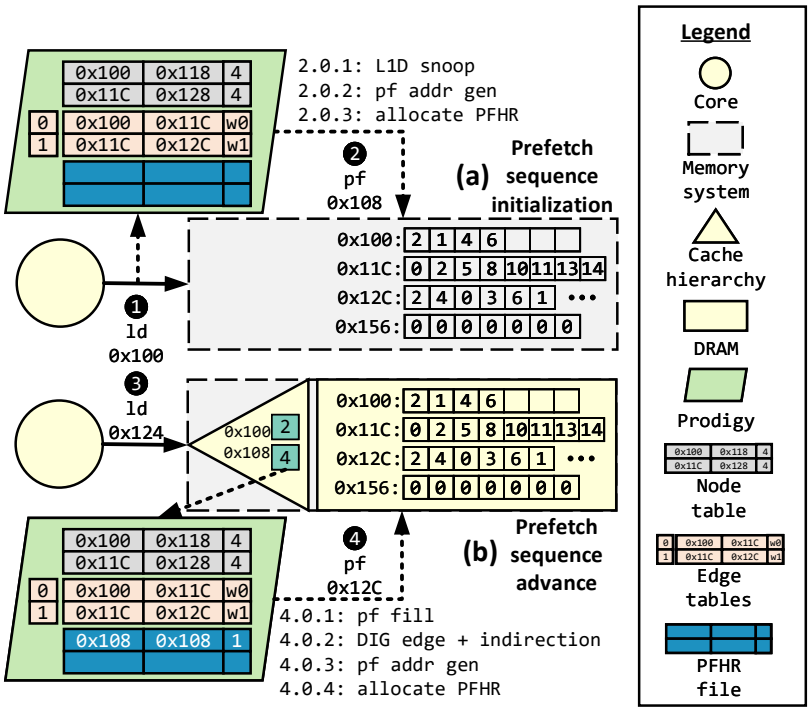


Figure 3.10: Prodigy operation: (a) prefetch sequence initialization, and (b) prefetch sequence advance.

Fig. 3.10 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. 2.1. 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.

Fig. 3.10(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 §3.3.3, 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. 3.10(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.

3.3.5 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 [73]). With this contiguous partitioning, Prodigy mostly prefetches the 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.

3.3.6 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.

3.3.7 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 work. We envision Prodigy to be used alongside a prefetch throttling mechanism similar to [241] that can identify and prevent prefetch-induced cache pollution to further improve performance. We leave studying the best throttling techniques as future work.

3.4 Methodology

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

3.4.1 Simulation Infrastructure

We use Sniper [36]—a Pin [154] based x86 multi-core simulator with an interval core simulation model. Sniper has been validated against several Intel micro-architectures [15, 36, 37]. We use

Table 3.1: 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

CACTI [175] to obtain cache access times for different cache capacities. We use the McPAT [146] model built into Sniper to model energy consumption. We implement our compiler analysis techniques using LLVM passes [130]. We evaluate our approach by modeling a parallel shared memory system with 8 cores as described in Table 3.1. 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.

3.4.2 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) [26] for evaluation—Betweenness Centrality (`bc`), Breadth-First Search (`bfs`)[2], 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 [61] as representative sparse linear algebra applications. Additionally, we use Conjugate Gradient (`cg`) and Integer Sort (`is`) from the NAS parallel benchmark suite [19] as representative computational fluid dynamics applications. We choose these algorithms as they exhibit single-valued

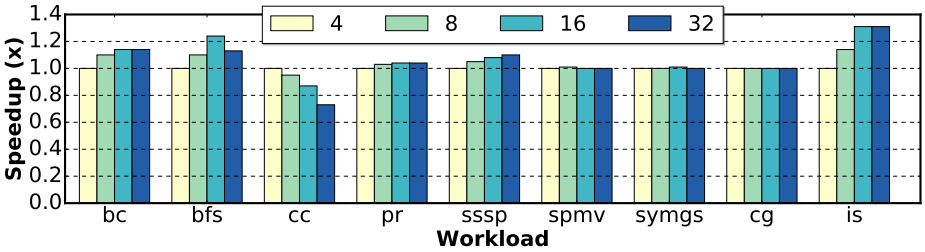


Figure 3.11: Design space exploration on the PFHR file size. Performance of each configuration is normalized to 4 entries.

and/or ranged indirections.

Table 3.2: Real-world graph data sets used for evaluation.

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

Data sets. As inputs to the graph algorithms, we use real-world graph data sets from SNAP [139] and UF's sparse matrix collection [57] as shown in Table 3.2. 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×2M, and 33M (for `is`) and 75k (for `cg`) , respectively.

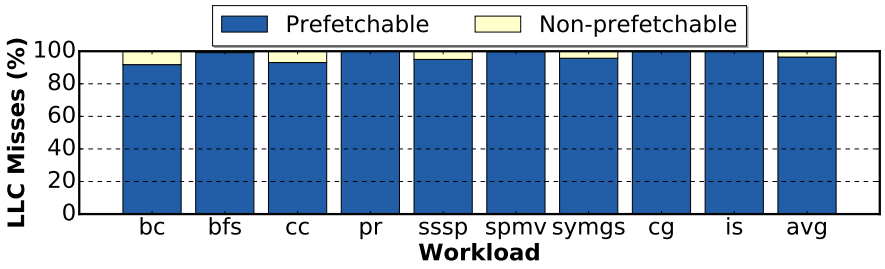


Figure 3.12: Classification of LLC miss addresses into potentially prefetchable and non-prefetchable addresses.

3.5 Results

3.5.1 Design Space Exploration

We perform design space exploration on Prodigy to understand the trade-off between performance and hardware complexity. Fig. 3.11 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 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.

3.5.2 Prefetching Potential

To estimate the potential prefetch coverage of Prodigy, Fig. 3.12 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

[2]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.

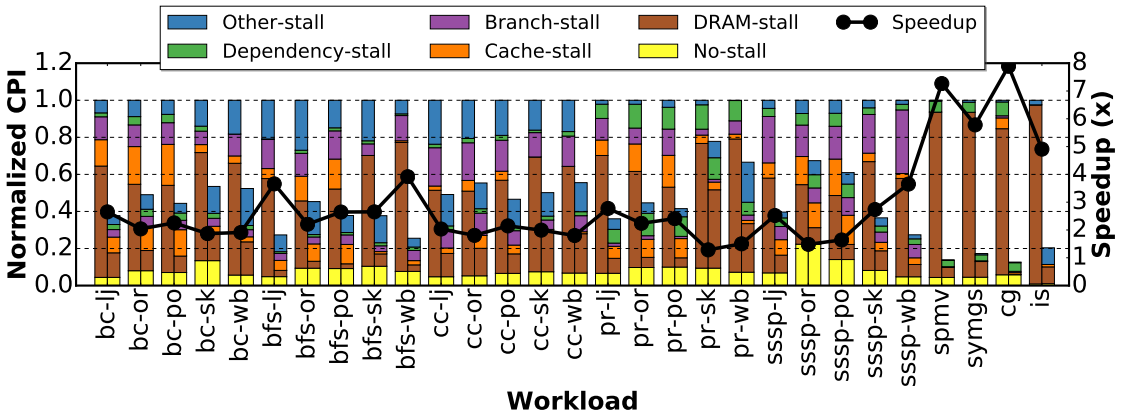


Figure 3.13: CPI stack comparison and speedup achieved by Prodigy against a non-prefetching baseline. Left bar: CPI stack of baseline; right bar: CPI stack of Prodigy normalized to baseline. Lower is better for CPI, higher for speedup.

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.

3.5.3 Effect on Performance

Prodigy vs. no-prefetching: Fig. 3.13 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, 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

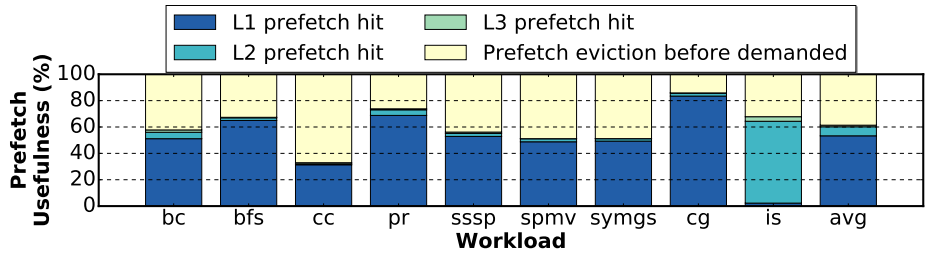


Figure 3.14: Location of prefetched data in the cache hierarchy when it is demanded. Blue is better.

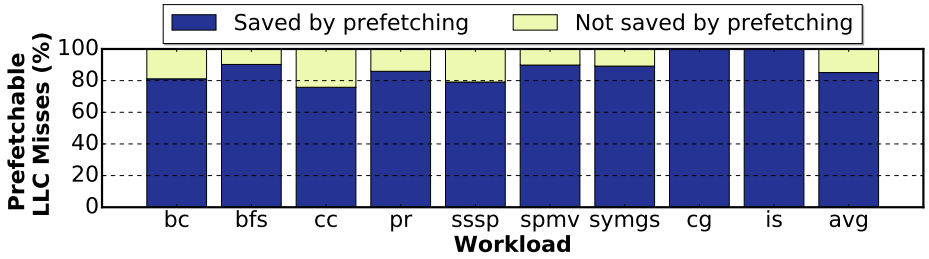


Figure 3.15: Percentage of prefetchable main memory accesses (as shown in Fig. 3.12) converted to cache hits. Blue is better.

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 [242].

Prefetch Usefulness: Fig. 3.14 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

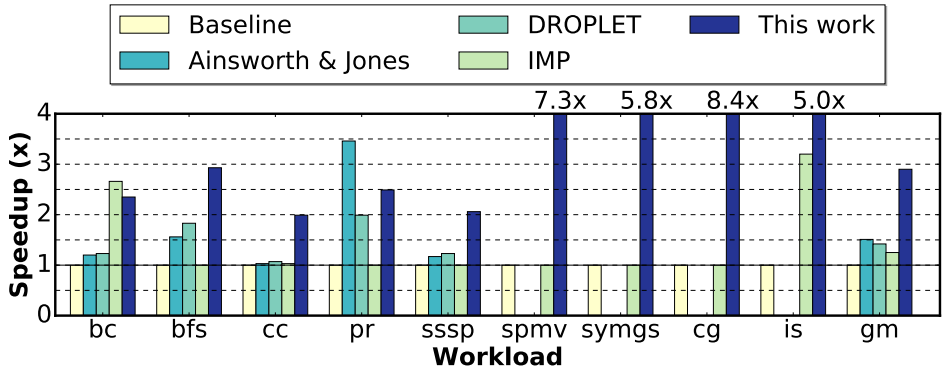


Figure 3.16: Performance comparison of a non-prefetching baseline, Ainsworth and Jones' prefetcher [11], DROPLET [24], IMP [282], and Prodigy (this work). Higher is better. Ainsworth & Jones and DROPLET are graph-specific approaches, and hence are omitted from non-graph workloads.

an accuracy of 62.7%. Furthermore, most of these cache hits are found in the L1D 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 [3]) since eviction is a consequence of imperfect timeliness. Fig. 3.15 shows the percentage of prefetchable LLC misses (blue portion of the bar in Fig. 3.12) 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 [54, 216, 282] 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 [177], Ainsworth and Jones' prefetcher [11], DROPLET [24], and IMP [282]. 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 [177] (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. 3.16 shows the performance comparison of various prefetchers using our simulation framework. Prodigy outperforms Ainsworth and Jones' prefetcher[3] [10, 11] by 1.5×. We have verified with the authors [9] that our implementation and results are correct. The difference compared to [11] 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 [11]. Also, unlike Prodigy, initiating one prefetch sequence in [11] 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 [11] is presented in [10], it incurs significant area overhead of 32KB of storage vs. 0.8KB for Prodigy.

Compared to DROPLET [24], 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[4] [282], because IMP can only

[3]We used open-sourced artifacts of for the evaluation of [11], and verified the presented results with the authors [9].
[4]We used the artifacts provided by the authors for evaluating IMP.

Table 3.3: Average speedup comparison over no prefetching.*

Common algorithms	Prior work		Prodigy
bc,bfs,bc,pr	Ainsworth & Jones [11]	2.4×	2.8×
bc,bfs,bc,pr,sssp	DROPLET [24]	1.9×	2.9×
bfs,pr,spmv,symgs	IMP [282]	1.8×	4.6×

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

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 3.3 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-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 [13] for indirect memory accesses. To make our evaluation consistent with [13], we evaluated the performance of software prefetching on an Intel Broadwell microarchitecture and validated our results with authors of [8]. Our findings show that for pr, performing a pure software-based prefetching [13] 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 [13] is not able to detect dynamically allocated array

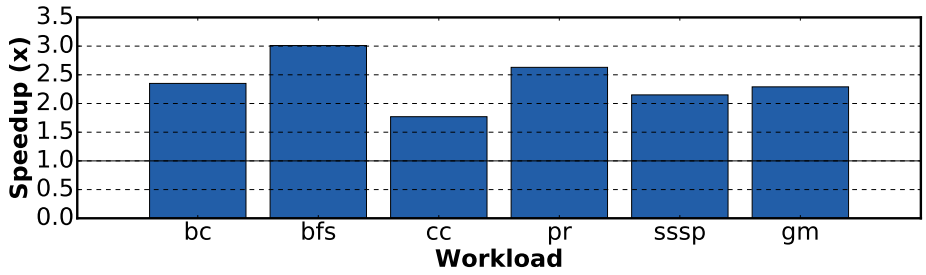


Figure 3.17: Speedup of Prodigy compared to a non-prefetching baseline on reordered graph data sets using HubSort [21].

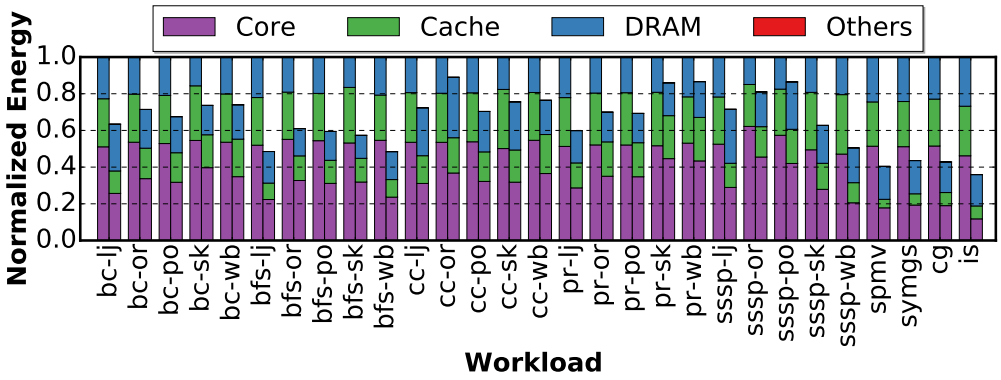


Figure 3.18: Normalized energy comparison of a non-prefetching baseline (first bar) and Prodigy (second bar). Lower is better.

sizes, and conservatively avoids placing prefetch instructions to prevent faults [8].

Graph reordering: We also evaluate the performance benefits of Prodigy on reordered graphs using HubSort [21]. Fig. 3.17 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.

3.5.4 Effect on Energy

Fig. 3.18 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 [242].

3.5.5 Overhead Analysis

Prodigy's hardware consists of a finite-state machine, whose area is dominated by the storage structures discussed in §3.3.1. 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. 3.11, 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× [282], 2× [11], 9.7× [24], and 40× [10].

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.

3.5.6 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.

3.5.7 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 [26] 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 [59, 290] 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 [241]. 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 work. 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.

3.6 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 [30, 82, 108, 157, 237, 238, 257] 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 [46, 51, 52, 153, 289] propose using a separate thread to speculatively prefetch data to reduce memory latency of the main thread. **Run-ahead execution** [62, 176] and some other architectures [75, 296] 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** [10, 11, 24] 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 [74, 121, 122, 129, 181, 198, 271, 282] 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** [103, 268, 270, 271] and **non-temporal prefetchers** [20, 28, 116, 117, 164, 231, 239] 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 [17, 48, 54, 66, 98, 112, 216, 279] 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 §3.5.3. Prodigy on the other hand, can cover all types of indirection present in graph algorithms.

Software prefetching [13, 34, 114, 150, 171, 261] 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** [3, 7, 87, 172–174, 186, 236, 240, 278, 286, 288] 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 [64, 65, 100, 102, 117, 132, 182, 206, 226, 227, 241, 269] 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.

3.7 Chapter Conclusion

This work 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.

CHAPTER 4

Understanding The Random Walk-Based Temporal Graph Learning

This is a collaborative work with D. Jin, H. Ye, A. Brahmakshatriya, S. Amarasinghe, T. Mudge, D. Koutra, R. Dreslinski.

A graph[1] is a ubiquitous data structure that models entities and their interactions through the collections of nodes and edges. It is widely employed in many domains ranging from social media [18] to bioinformatics [106, 158]. More recently, the process of learning representation of graph structured data, *i.e., graph representation learning*, has gained significant popularity in the algorithmic community [88, 119, 179, 200, 262]. This is due to its superiority on multiple machine learning tasks in domains ranging from social science [170, 281], computer vision [151], physics, chemistry, and biology [49, 63, 213, 244]. Following this algorithmic evolution, several works in the architecture community have analyzed its workload characteristics [23, 275, 295], and built domain-specific hardware [77, 143, 276] for acceleration.

The scope of these works, however, has so far been limited to (a) *static* input graphs [96], and (b) a *subset* of graph learning algorithms including Graph Convolution Network (GCN) [119], and a few others [88, 273]. Nonetheless, most real-word graphs are dynamic in nature, *i.e.,* naturally evolving over time by adding, deleting, or changing their nodes and edges. Modeling these dynamic graphs as static would inevitably incur information loss and performance deterioration of downstream predictive tasks. Moreover, while GCN has shown state-of-the-art algorithmic

[1] In this work, we use the term "graph" and "network" interchangeably.

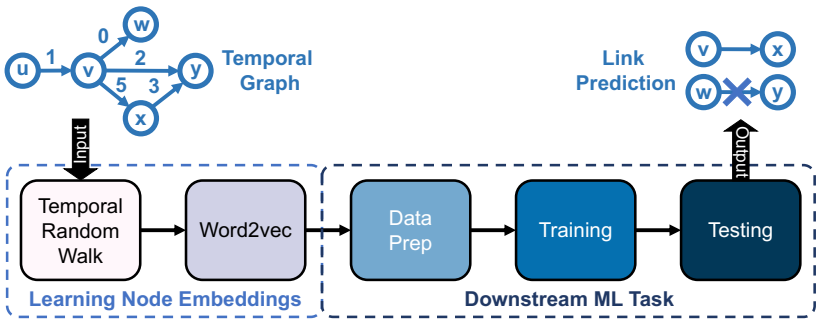


Figure 4.1: A high-level overview of our modeled pipeline that takes a temporal graph as an input and learns the network dynamics to encode each node into a low-dimension embedding space by using *temporal random walk* and *word2vec*. These embeddings are then fed into a downstream machine learning task such as link prediction or node classification.

performance on various prediction tasks [96], it mostly works on static graphs and cannot model the graph dynamics such as the sequential interactions between nodes and temporal dependency between graph snapshots. Besides, high computation and memory complexity of GCN makes it difficult to scale to large-scale graphs [96].

In this work, we investigate the behavior of a fundamentally new class of graph learning algorithms for temporal graphs based on random walks, namely, temporal random walk [179]. Temporal graphs are a category of dynamically evolving networks with timestamp information associated with each network interaction (*i.e.,* temporal edge). Informally, a temporal walk is defined as a sequence of temporally-valid edges $\{(u, v_1, t_1), (v_1, v_2, t_2), \cdots, (v_{i-1}, v_i, t_i)\}$, where $t_{i-1} \leq t_i$. As an example, for the temporal graph shown in Fig. 4.1, the walk $\{u, v, x\}$ is temporally-valid as it naturally indicates how the node u interacts with its neighbors with respect to time, while $\{u, v, w\}$ is invalid. Temporal random walk is an important algorithm that underlies a wide range of applications on graphs such as information cascading [141], user behavior modeling [107]. It is also the foundation of many follow-up research in the field of machine learning and representation learning [193, 218, 258]. However, temporal random walk has gained relatively less popularity in the architecture community so far. Additionally, this conceptually straightforward algorithm could effectively model the temporally-valid node interactions while being more scalable [200] to handle large-scale graphs. Furthermore, we show that a workload resulting from temporal random walks exhibits distinct characteristics compared to traditional graph processing and GCN algorithms (see

§4.3.2).

Fig. 4.1 shows an overview of a canonical pipeline based on a prior algorithmic work [179]. We model high-performance implementations of two variants of this pipeline for both the CPU and GPU-based computing. The front-end of the pipeline employs temporally-valid random walks and word2vec, a technique from Natural Language Processing (NLP), to map nodes into a low-dimension embedding space. This process translates the similarity between nodes in the original network into closeness in the embedding space. Then, these node embeddings are fed into downstream machine learning tasks. Specifically, this part models the two most widely-known graph learning tasks, used in several applications, as follows.

- **Link prediction.** This task predicts the presence/absence of an edge between a given pair of nodes. A concrete application of this task is product recommendation from the online sales websites such as Amazon.

- **Node classification.** This task assigns labels to nodes. Its concrete application is identifying the professional role of a user in social networks such as LinkedIn.

Based on this pipeline, we perform detailed two-step performance characterization: (a) algorithm-focused, and (b) hardware-focused. This reveals a rich design space and performance acceleration opportunities as listed below.

(a) Accuracy-complexity trade-off. While high prediction accuracy is desirable, it *does not* always come with high cost. We use three hyperparameters to show this: (a) number of random walks per node, (b) random walk length, and (c) embedding space dimensionality. While increasing these values monotonically increases workload memory consumption and execution time, their benefit in accuracy are limited. While prior works [85, 179, 200] often over-provision these values, we find optimal parameters balancing accuracy and complexity.

(b.1) Instruction diversity. By analyzing dynamic instruction types of individual kernels, we find the dominance of both memory and compute instructions, indicating the necessity to optimize both types of operations. This is particularly interesting for temporal random walk that executes more

compute operations than traditional graph processing.

(b.2) Thread scalability. Despite irregularity, individual workload kernels can scale well using work stealing.

(b.3) Time Breakdown and CPU versus GPU. Classifier training dominates the execution time of end-to-end workload; accelerating training will yield high workload speedup. A cross-platform workload comparison reveals that the GPU outperforms CPU at large graph sizes.

(b.4) Execution Bottlenecks. GPU workload characterization reveals that individual kernels exhibit diversity of bottlenecks including cache misses, and compute and memory dependency.

Using these insights, we discuss strategies to optimize this workload for future exploration using: algorithm, ML framework, GEMM library, compiler, and hardware.

This is the first work introducing the random walk-based learning pipeline on dynamic graphs for computer architecture research. In summary, we make the following contributions:

- High-performance CPU and GPU implementations of random walk-based temporal graph learning tasks.
- A detailed algorithic workload characterization presenting a rich accuracy-complexity trade-off space.
- An in-depth hardware-focused performance characterization uncovering future optimization opportunities.
- Open-source benchmark implementations and datasets for the benefit of the broader research community at
  https://github.com/talnish/iiswc21_rwalk.

4.1 Related Work

4.1.1 Graph Representation Learning

Recently, graph representation learning or node embedding has attracted massive research attention from both academia and industry due to its success in downstream tasks like link prediction and node classification. Inspired by the notion of word proximity from NLP, early research in graph learning focused mainly on leveraging the node proximity in a graph, such as DeepWalk [200] and node2vec [85]. These works either leverage first or second-order node proximity [254], or higher-order (> 2) [35] to construct the global node representations. Additionally, there are works based on graph structural properties. For example, struc2vec [213] defines similarity in terms of degree sequences in node-centric subgraphs, and role2vec [6] inductively learns structural similarity by introducing attributed random walk atop relational operators. Furthermore, other works attempt to incorporate external node features with the graph structures [88, 119, 262]. For instance, Graph Neural Network (GNN) [220, 297] and its variants propose to aggregate node features in its dependent contexts with arbitrary depth via propagation/diffusion. Representative works include GCN [119], GraphSAGE [88], and GAT [262].

4.1.2 Temporal Network Modeling

Temporal network modeling has been widely studied in dynamic network analysis [4, 92]. Most existing works in the field of machine learning and representation learning empirically process the temporal graph as a sequence of snapshots [83, 144, 218]. While the sequential order of the snapshots models the evolution of temporal dynamics, each individual snapshot is static and analyzed without the temporal information. Streaming graph models can be seen as an extreme case of the snapshot model, where the most recent snapshot is a dynamically changing graph in real time [5, 131]. Another direction that is orthogonal to snapshot-based methods is based on sequential interactions between node pairs in the graph. In this work, we follow an earlier algorithmic work CTDNE [179], which proposes the notion of temporal walks and leverages it to learn embeddings directly from the

stream of timestamped edges at the finest temporal granularity. Other works [126, 300] propose to model the sequential interaction as the point-process to predict the occurrence of link over time.

4.1.3 Software Frameworks

Several software frameworks have been proposed to understand performance implications of different graph learning algorithms [72, 155, 266, 285, 298]. However, these frameworks mostly model GCN algorithm and a few others [119]. This work, on the other hand, models random walk-based graph learning. Additionally, there has been tremendous efforts for developing high-performance implementations for traditional bulk-synchronous graph applications on shared memory systems [1, 78, 84, 93, 199, 223, 233, 234, 245, 246, 263, 287, 292]. These frameworks implement abstractions for programming graph applications as a library of high-level primitives or a new programming language and compilers [188, 291, 293]. They also combine optimizations with different iteration orders, data structures, direction-optimization [291] *etc.* to improve performance across different graph inputs and applications.

4.1.4 Hardware Proposals

Several prior works accelerate similar algorithms using novel hardware designs. In the context of our work, similar algorithms include graph traversals, traditional deep learning, and graph neural networks. A subset of prior works focus on optimizing graph algorithms on the CPU using techniques such as hardware prefetching [11, 24, 252]. Other works optimize graph algorithms on GPUs [94, 115, 221]. Additionally, several accelerators have also been proposed to accelerate graph traversals [87, 186, 208, 277]. Both traditional deep learning and graph neural networks have been extensively optimized using hardware accelerators [44, 77, 110, 143, 148, 276, 284]. However, random walk based graph learning is not well studied in the context of hardware accelerators. In §4.3.2, we show that random walk-based graph learning exhibits significantly different nature in terms of its characteristics compared to aforementioned well-studies application domains, motivating the need for our study.

Symbol	Definition
$G(\mathcal{V}, \mathcal{E})$	a directed temporal network with $\lvert\mathcal{V}\rvert$ nodes and $\lvert\mathcal{E}\rvert$ edges
$G_t(\mathcal{V}_t, \mathcal{E}_t)$	a snapshot of the temporal network G at time t with $\lvert\mathcal{V}_t\rvert$ nodes and $\lvert\mathcal{E}_t\rvert$ temporal edges
$\mathbf{A}$, $\mathbf{A}_t$	adjacency matrix for graph G and G_t, respectively
$w_{u,v}$	a temporal walk reaching out from u to v
f	arbitrary base embedding method
d	dimensionality of the embedding
$\mathbf{Z}$	$\lvert\mathcal{V}\rvert \times d$ embedding matrix

Table 4.1: Summary of notation.

4.2 Preliminaries

This section provides the definitions of notions used in this work. The related symbols are listed in Table 4.1.

Definition 4.2.1 (Temporal Graph) *A temporal graph G consists of a set of nodes $\mathcal{V}$ and a set of temporal edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathbb{R}^+$, where $t \in \mathbb{R}^+$ represents the timestamp of an edge $(u, v, t) \in \mathcal{E}$.*

At a high level, a collection of temporal edges $\{(u, v, t)\}$ forms a time-evolving network structure. For example, the time-evolving email exchange network is constituted by individual contacts from user u to v at time t. Comparing with static networks, the edge timestamps endorse in-depth analysis of the network dynamics over time. A fundamental data structure defined in temporal networks is a set of temporal walks, *i.e.,* a sequence of walks with respect to time [107, 179].

Definition 4.2.2 (Temporal Walk) *A temporal walk w from u to v in the network $G(\mathcal{V}, \mathcal{E})$ is defined as a sequence of connected edges $w_{u,v} = \{(u, u_1, t_1), (u_1, u_2, t_2), \cdots (u_k, v, t_k)\}$ where $t_i < t_{i+1}$ for $i = 1, 2, \cdots, k$.*

A temporal walk indicates the reachability from the source to destination node in a time-increasing order, which encapsulates detailed information about network dynamics as well as node characteristics. In the email exchange network example, temporal walks denote the paths of a user reaching out to another. These walks reflect how people get to know each other and further expand their social networks over time. In this process, detailed user activities such as reply, forward, *etc.* are critical to user profiling and behavioral analysis.

In order to mathematically characterize such node properties in the graph, the notion of graph representation learning has been proposed and widely applied in practice. The high-level idea is to map the nodes from the graph space to a low-dimensional distance space (*e.g.*, 128-d Euclidean space) such that the computational complexity is reduced while the similarity between nodes is preserved. As a result, the low-dimensional representation can be applied to various machine learning tasks such as link prediction, clustering, and node classification. The formal definition of graph representation learning is given as follows.

Definition 4.2.3 (Graph Representation Learning) *Given a graph $G(\mathcal{V}, \mathcal{E})$, graph representation learning aims to learn a function $f : G(\mathcal{V}, \mathcal{E}) \rightarrow \mathbb{R}^d$ that maps nodes from the graph to a low-dimensional space such that $d \ll |\mathcal{V}|$ and $d \ll |\mathcal{E}|$ while preserving the notion of similarity between nodes.*

Depending on specific approaches, the notion of similarity can be defined as the proximity between nodes. Intuitively, a node is more similar to its 1-hop neighbors than its 2-hop neighbors and other distant nodes. Thus, nodes that share common neighbors are embedded closely. On the other hand, node similarity can be measured through the functionality or structural role of a node in terms of its connection to its neighbors. For example, the centers of two star-like subgraphs are structurally similar to each other because they both are at the center and thus behave like "hubs" that bridge other nodes. In this work, we address the first type of node similarity in graphs through temporal proximity.

4.3 Motivation

Following the background discussion in §2.5, we briefly show how this workload is different from other standard benchmarks (specifically GCN) to motivate our study.

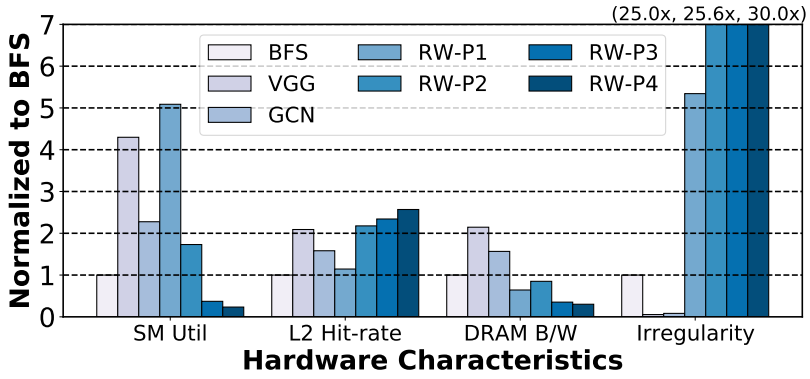


Figure 4.2: Hardware metric comparison of purely graph traversal (BFS), deep learning inference (VGG), graph convolution network inference (GCN), and modelled pipeline: RW-P1 (random walk), RW-P2 (word2vec), RW-P3 (training), and RW-P4 (testing) The figure showcases unique behavior of modeled application compared to other well-studied benchmarks.

4.3.1 GCN versus Random Walk-based Graph Learning

In comparison with GCN that performs spectral convolutional operation over a node's neighbors up to a pre-defined number of hops, temporally-valid random walk captures the sequential interactions with respect to time. As a basic way to explore the spatial property on temporal graphs, the presented algorithm exploits global graph property that is beyond the local node-centric subgraphs. Therefore, it is more powerful in predictive tasks such as link prediction. Furthermore, the presented algorithm works on feature-less graphs and uses a single-integer vertex-identifier as a feature, whereas GCN requires vertex-wise long feature vectors. Interestingly, there is connection between GCN and random walk, for example, [145] shows that random walk can be used to supplement GCN to improve performance on static graphs. However, the difference in these patterns result in different workload characterization and performance optimization strategies on temporal graphs.

4.3.2 Why Study this Workload?

Fig. 4.2 compares the hardware characteristics of a traditional graph traversal (BFS), deep learning inference (VGG), graph convolution network inference (GCN), and different workload phases of random walk based graph learning application (RW-P[1:4]) on a GPU. The figure shows GPU core utilization (SM Util), L2 cache hit rate, DRAM bandwidth utilization, load imbalance, and a

measure of irregularity (ratio of number of replayed to issued instructions) [33] normalized to BFS. The datasets used for these tasks are the following: BFS—a synthetic graph using `graphgen` utility from Rodinia [38] with 16M nodes and 117M edges, VGG—ImageNet [58], GCN—Reddit [88], and this work—a synthetic Erdős-Renyi graph with 10M nodes and 200M edges.

The figure clearly shows that random walk based graph learning pipeline yields unique characteristics compared to other applications, which warrants its further investigation. Specifically, the amount of irregularity (measured using a ratio of the number of replayed to issued GPU instructions) is high, which can be because of long-latency load instructions and/or load/branch divergence. These characteristics further results in low SM and DRAM bandwidth utilization.

4.4 Benchmark Implementation

This section presents implementation details of modeled graph learning applications for both CPU and GPU. At a high-level, this follows the flow presented in Fig. 4.1. We first present the temporal random walk algorithm and a modified version of word2vec that outperforms its open-source counterparts. Then, we briefly discuss the data preparation and classifier steps.

4.4.1 Temporal Random Walk

This is the first step of modeled pipeline that takes a temporal graph G as an input, and outputs temporally-valid random walks starting from each node in the graph. We build this kernel by extending a high-performance graph processing framework — the GAP benchmark suite (GAPBS) [26]. We use the weighted graph structure `WGraph` for storing a temporal network, which stores graph edges as an array of structures (*i.e.,* destination and weight). The weight field is re-purposed to store timestamps with appropriate changes in the data type. Furthermore, we add support to preserve multiple edges between the same source and destination vertices. This is important to preserve multiple temporally-distant interactions between the same set of nodes.

This algorithm is shown in Algorithm 3. Its time complexity is $\mathcal{O}(KN|\mathcal{V}|M)$, where K is the

Algorithm 3 Pseudocode for temporal random walk

```
1: Input: Graph G in CSR format, temporal walk length N, Number of walks per vertex K
2: Output: Temporal walk output matrix of dimensions |G.V| × K × N, W
3: W ← new matrix[|G.V|][K][N]
4: for w : 0 → K do
5:     par_for v : 0 → |G.V| do
6:         currVertex ← v
7:         currTime ← 0
8:         for i : 0 → N do
9:             if G.neighbors(currVertex) == 0 then
10:                break
11:            currVertex, currTime ← G.sampleLatent(currVertex, currTime)
12:            W[v][w][i] ← currVertex
13:    end par_for
```

number of random walks per node, N is the length of each random walk, $|\mathcal{V}|$ is the total number of vertices in the graph, and M is the max degree of all the vertices in the graph. The factor of M comes from the call to the `G.sampleLatent` function (line 11) that iterates through all the neighbors of the vertex and compares each edge against the timestamp. With any value of `currVertex`, this would have to process edges equal to the maximum degree in the graph. There are three nested loops: 1) the outer loop to iterate over the walk number per node when performing multiple random walks per node (line 4); 2) the middle loop to iterate over all the vertices in the graph (line 5); and 3) the inner loop to iterate over an individual step of a walk (line 8). In our implementation, we parallelize the middle loop that iterates over all vertices, based on an empirical finding that it offers optimal performance compared to alternative settings.

4.4.2 Word2vec

This algorithm takes a series of temporally-valid random walks as an input and outputs node embeddings. For the CPU, we adopt an open-source implementation [166]. However, we find that the available GPU implementations [180, 235] have sub-optimal performance when applied to the graph learning problem. This is because of their parallelism model. These implementations parallelize word embedding updates *within* each sentence, and processes different sentences sequentially. While this might be optimal in NLP with long sentences, it leads to poor parallelism in the graph learning context. This is because, as shown in Fig. 4.3, the random walk lengths (*i.e.,* the number of walks that complete for a given length given the timestamp constraints) are centered around 1 to 5.

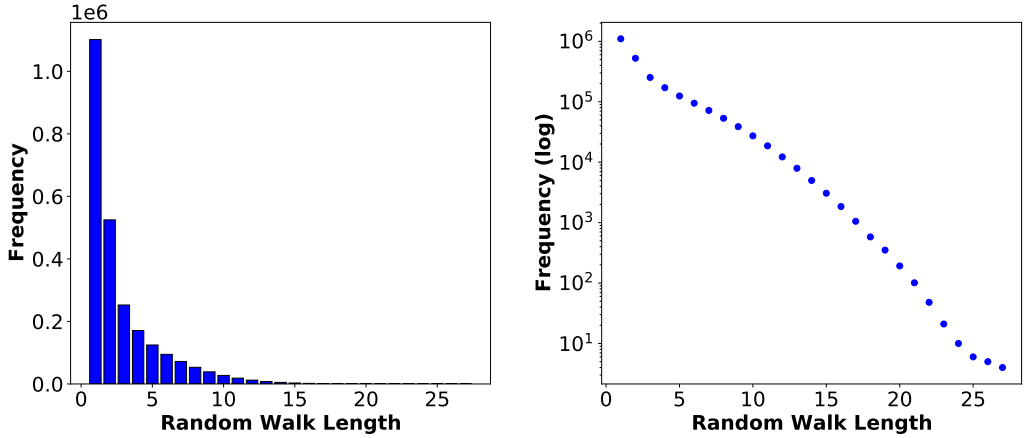


Figure 4.3: The power-law distribution of temporal random walk lengths on wiki-talk dataset (in linear and log scales). Most walks are of short lengths, and the frequency of longer walk length decreases exponentially. Other datasets also show similar patterns.

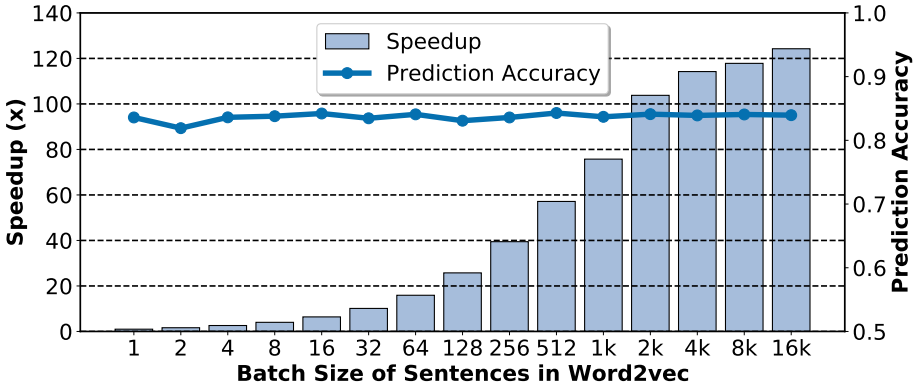


Figure 4.4: Sensitivity of word2vec phase speedup and end-to-end link prediction accuracy for different batched sentence sizes on a GPU using wiki-talk dataset. Compared to a baseline open-source implementation [180, 235], our batch implementation gains 124.2× speedup without a loss in accuracy at a batch size of 16k sentences.

As the walk length is analogous to sentence length, the word2vec input constitutes a large number of *short* sentences. This causes the GPU resources to be under-utilized and launches a large number of GPU kernels, one launch for each sentence.

To improve this implementation, we propose the following optimizations. First, we batch multiple sentences together, and process sentences within a batch in parallel. This adds a new possibility to read from a stale word embedding model, potentially reducing accuracy, as we process multiple word embedding updates concurrently. However, because the model update is a sparse operation [200], concurrently updating word embedding model does not result in an accuracy loss.

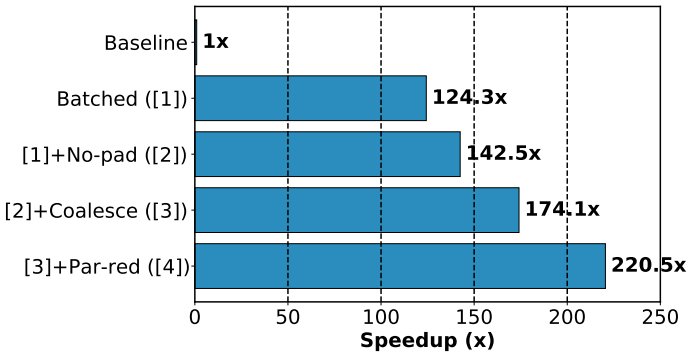


Figure 4.5: Speedup of the word2vec phase on a GPU for different optimizations. Compared to baseline, batched sentences (Batched), no cache line padding (No-pad), memory operation coalescing (Coalesce), and parallel reduction (Par-red) result in an end-to-end speedup of 220.5× on wiki-talk dataset.

On the flip side, this technique greatly improves the GPU core utilization. Empirically, Fig. 4.4 shows that the batch size of 16k achieves a *124.2× speedup* over no batching *without accuracy loss*. The speedup is attributed to (a) improved GPU core utilization, (b) CPU-GPU data transfer cost amortization over long computation, and (c) reduced kernel launch overhead.

Second, a prior implementation [180] uses cache line padding to address false sharing at the private L1 caches. This heavily under-utilizes cache lines as our embedding space dimension is small (*i.e.,* 8 as shown in §4.6.1). To optimize cache line utilization, we remove the cache line padding (No-pad) and add support to bypass the L1 cache. Third, we assign multiple GPU threads to process each embedding dimension in a coalesced manner (Coalesced), and use parallel reduction for accumulation (Par-red). With a small embedding dimension, we also eliminate all the `__syncthreads()`, and rely on the in-warp synchronization. Fig. 4.5 shows the benefit of each of these optimizations, leading up to an end-to-end speedup of 220.5× on the wiki-talk dataset without accuracy loss.

4.4.3 Data Preparation

Inputs to this step include the node embeddings from word2vec, and a temporal edge list/a labeled node list for link prediction/node classification. This step outputs datasets for training (S^{tr}),

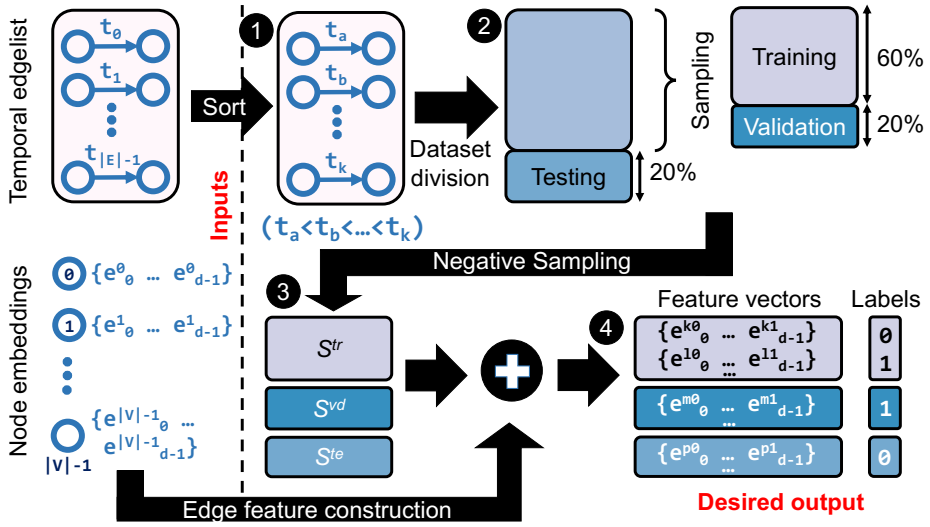


Figure 4.6: Data preparation step for link prediction.

validation (S^{vd}), and testing (S^{te}).

Fig. 4.6 shows the data preparation algorithm for link prediction. First, the input edges are sorted by their timestamps (❶) and then 20% of the edges are chosen for testing from the end of this list. The intuition behind sorting the edges is to train the classifier on the past edges and test it on the future edges. Excluding the testing edges, 60% and 20% of the total edges are randomly sampled for training and validation (❷), respectively. Because these edges exist in the original input network, they form *positive edge sets* with a label 1. Negative sampling (❸) is used to construct negative edges with a label 0. This is done by altering one/both vertex IDs of positive edges so that the resulting edge is absent in the input graph. After constructing these sets, edge features are computed by concatenating node embeddings as described in §2.5.2 (❹). A similar mechanism is employed for node classification, where labeled dataset precludes the need for negative sampling.

4.4.4 Classifier

Data obtained in the previous stage is fed into the classifier, which goes through training and testing phases. We use an FNN-based classifier as discussed in §2.5.2.

4.5 Experimental Methodology

This section details our experimental methodology. Specifically, we talk about our hardware platforms, software toolchain, and input graph datasets used for evaluation.

4.5.1 Hardware Platforms

We characterize modeled applications on two platforms — CPU and GPU. We use a dual-socket server with two AMD EPYC^{TM} 7742 CPUs with 128 physical cores (256 SMT threads). The aggregate Last Level Cache (LLC) size is 2×256MB. The size of main memory is 512GB. Additionally, we use a discrete NVIDIA GPU with Ampere architecture.

4.5.2 Software Toolchain

We model our applications in `C++` and compile them using the `g++ v7.5` compiler with `-O3` optimization level for the CPU. We compile CUDA programs using `nvcc v11.2` with `-O3` and `-arch=sm_80` flags. For hardware profiling, we use manual instrumentation and MICA Pintool [95] for the CPU, and NVIDIA Nsight Compute [183] for the GPU. We use dynamically scheduled OpenMP threads for CPU parallelism. The downstream ML task is implemented using the `PyTorch-C++` API [207].

Task	Dataset Name	#Nodes	#Temporal Edges	Description
Link prediction	ia-email [50, 215]	87,274	1,148,072	Enron email network from Jan. 1998 until Feb. 2004
Link prediction	wiki-talk [135, 139, 192]	1,140,149	7,833,140	User editing network of Wikipedia Talk pages
Link prediction	stackoverflow [139, 192]	6,024,271	63,497,050	Stack exchange interaction network on Stack Overflow
Node classification	dblp5 [272]	6,606	42,815	Co-author network from DBLP from 5 research areas
Node classification	dblp3 [272]	4,257	23,540	Co-author network from DBLP from 3 research areas
Node classification	brain [205, 272]	5,000	1,955,488	Connectivity network of tidy cubes of brain tissues

Table 4.2: Real-world temporal networks used for algorithmic evaluation.

4.5.3 Input Datasets

We use both real-world and synthetic graphs for evaluation. Because the publicly available real-world temporal datasets are limited in size, we use them for algorithmic evaluation. Table 4.2 shows the list of these datasets and their properties. For hardware study, we use large-scale synthetic graph datasets generated using Python-based `networkx` library. Specifically, we generate Erdős-Renyi random graphs, with varying sizes and degrees, with synthetic timestamps.

4.6 Results and Analysis

Presented analysis is divided into two parts: (a) algorithm-focused study, and (b) hardware-focused study. The former presents the trade-off between prediction accuracy and runtime performance. The latter focuses on understanding the workload characteristics to find performance optimization opportunities.

4.6.1 Algorithmic Analysis

We study the effect of three important algorithmic parameters: number of random walks per node, walk length, and embedding space dimension. As shown in §4.4.1, runtime complexity of the random walk algorithm is proportional to the number of random walks per node and walk length. Additionally, the runtime complexities of word2vec and classifier training/testing are dependent on the embedding space dimension as it decides the feature vector length. Therefore, increasing these parameter values will increase the execution times of different kernels. Fig. 4.7(a) empirically confirms this finding by showing the increase in random walk execution time when increasing in the number of walks per node for the stackoverflow dataset. A similar trend is observed for random walk length and embedding space dimension. In general, we find that the performance on link prediction tasks is better than node classification. This is because that temporal random walk exploits global graph property that is beyond the local node-centric subgraphs. As the task of node classification requires detailed information centric to specific nodes, temporal random walk is

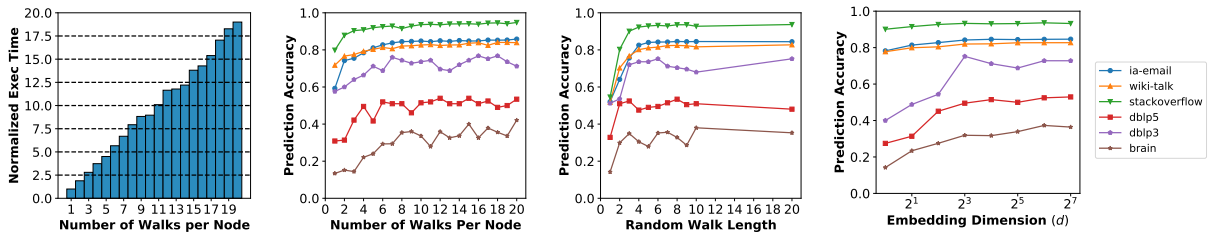


Figure 4.7: Accuracy-complexity trade-off. (a) Normalized execution time of the random walk kernel for different number of walks per node, and (b-d) Accuracy of link prediction and node classification with respect to different parameter values.

not the optimal algorithm for it. Thus, as link prediction requires more global information about the graph connectivity, the performance is better. Next, we present the parameter sensitivity on prediction accuracy of downstream tasks.

Number of Random Walks Per Node. As shown in prior works [85, 179, 200], the network is best sampled by performing multiple random walks from the same node. This is because one walk can only sample a vertex neighborhood via one of its neighbors. Performing multiple walks from a node can potentially sample a *wider* vertex neighborhood, enriching the amount of information used for downstream learning tasks. Fig. 4.7(b) shows the effect of performing multiple random walks from a node on the prediction accuracy of link prediction and node classification. The figure confirms that more walks from the same node increases the prediction accuracy. Interestingly, this improvement saturates after 8-10 walks. This is because of the power-law nature of real-world graphs, *i.e.,* most nodes have few neighbors. In a majority of sparsely connected nodes, performing 8-10 walks are enough to cover most neighbors. Beyond this, there is limited value by performing more walks.

Random Walk Length. Length of the random walks indicates the distance of sampled neighbor from the source. For example, a random walk of length 5 will sample a 5-hop neighbor from a source vertex. While multiple random walks per node sample *wide* neighborhoods, larger random walk length indicates the sampled neighborhood *depth*. Intuitively, larger the length of random walk, deeper the network can be sampled. Fig. 4.7(c) shows an increase in prediction accuracy with an increase in the random walk length. This trend, however, saturates after a walk length of 4-6,

which can be described using an earlier finding. Fig. 4.3 shows that the frequency of random walks decreases with increased walk length. This translates into marginal information gain with large walk lengths and saturation in prediction accuracy.

Embedding Space Dimension. At a high level, a graph learning task maps each node to an embedding space, where the dimension of the embedding space defines complexity of interactions that can be modeled. While prior algorithmic works [85, 179, 200] use a fixed dimension size (d) of 128, we analyze how this affects end-to-end accuracy. Fig. 4.7(d) shows the effect of changing d on the prediction accuracy. Increasing d from 1 to 8 results in gain in prediction accuracy as higher dimensions can model more complex network interactions. Interestingly, we find that an embedding space of dimension 8 is enough to make meaningful network predictions.

To summarize, there exists a rich trade-off space between algorithmic performance and runtime complexity. *While increasing the value of aforementioned hyperparameters will monotonically increase the execution time of different kernels, their effect on prediction accuracy is limited.* Based on our empirical findings, we find the optimal values of number of random walks per node, random walk length, and embedding space dimension to be 10, 6, and 8, respectively.

4.6.2 Hardware Analysis

Next, we perform a detailed hardware analysis based on the optimal parameter values found above. Using real-world and synthetic graph datasets, we study the instruction diversity, scalability, time breakdown, and execution bottlenecks.

Instruction Diversity. Instruction diversity characterization helps understanding the operation types present in a workload, which can be used to make design decisions building specialized hardware. Fig. 4.8 shows the breakdown of dynamic instruction types of individual kernels on a CPU for the link prediction task on ia-email dataset. This is divided in terms of memory, branch, compute (both arithmetic and floating point), and others. The others category includes instructions for stack usage, bitwise shifts, string operations, SIMD, *etc.*

The figure shows that both compute (36.6% on average) and memory (30.4% on average)

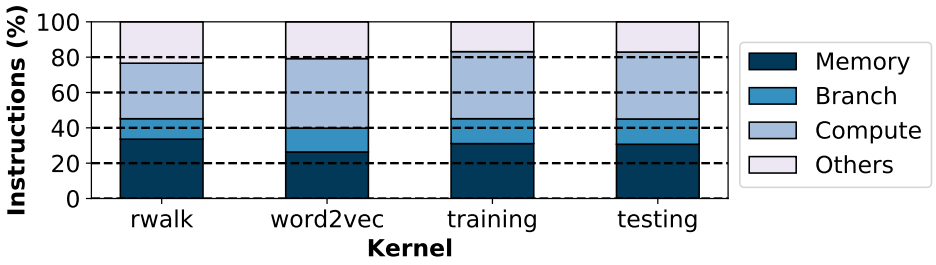


Figure 4.8: Dynamic instruction breakdown of different kernels involved in link prediction for ia-email dataset. The figure shows that all kernels have a high number of both compute and memory instructions.

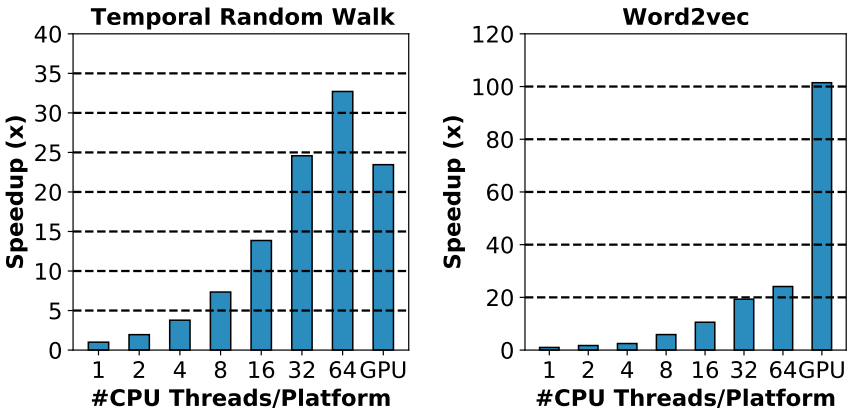


Figure 4.9: CPU thread scaling analysis and its comparison with GPU implementation for temporal random walk and word2vec kernels on the stackoverflow dataset. The speedups are normalized to a single-thread implementation. The figure shows reasonable scaling trend.

operations are dominant in all kernels. Word2vec and classifier training/testing phases use neural network-type computation, hence, this breakdown is not surprising. However, a similar count of compute and memory instructions for random walk is surprising as graph traversals are known to have a low memory-to-compute operation ratio. This distribution is attributed to the compute-intensive operations used in selecting a neighbor to walk as shown in Eq. (2.1). As a takeaway, system designers should target both compute and memory operations for optimizing all workload kernels.

Scaling Analysis. Fig. 4.9 shows the thread scaling behavior of temporal random walk and word2vec kernels for stackoverflow. Additionally, it shows GPU performance normalized to a serial CPU implementation. Using more than 64 threads does not improve performance further as the thread creation/logic logic dominates the computation cost. We do not show the scaling

	rwalk		word2vec		training/epoch		testing	
$\|\mathcal{V}\|$,$\|\mathcal{E}\|$	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
10k,50k	0.0	1.2	0.3	0.4	0.3	0.7	0.1	0.0
10k,100k	0.0	1.2	0.3	0.3	0.6	0.8	0.4	0.1
10k,200k	0.0	1.2	0.3	0.4	0.8	1.2	0.2	0.2
100k,500k	0.1	1.2	0.7	0.4	2.4	2.3	0.5	0.5
100k,1M	0.1	1.2	0.6	0.4	4.1	3.4	1.2	0.8
100k,2M	0.1	1.1	0.6	0.4	5.1	8.0	2.2	1.6
1M,5M	0.9	1.4	2.7	1.3	13.7	15.5	6.0	4.2
1M,10M	1.2	1.4	3.4	1.4	32.5	28.3	7.8	7.0
1M,20M	1.8	1.4	3.2	1.6	62.2	58.6	20.7	14.7
10M,50M	12.2	4.0	25.4	20.0	147.7	147.1	56.3	44.2
10M,100M	14.2	4.0	27.3	22.1	315.8	303.9	133.0	87.8
10M,200M	18.7	4.2	36.8	27.4	695.2	668.5	233.1	206.9

Table 4.3: Execution times of workload phases in seconds for both CPU and GPU implementations. Cell colored in green indicates a faster implementation between CPU and GPU.

of classifier training/testing as its Pytorch-based implementation does not offer an explicit thread-scaling control[2].

The figure shows that both kernels show a reasonable thread scaling trend despite irregularity. For the random walk kernel, the amount of work per thread is dependent on the outgoing degree and timestamp distribution, which leads to heavy load imbalance in a naïve implementation. To alleviate this problem, we employ work stealing using dynamically scheduled OpenMP threads. The GPU performs similar to 32 CPU threads. This is because of the CPU-GPU data transfer time, and workload irregularity leading to branch divergence and non-coalesced memory accesses. On the other hand, the GPU implementation of word2vec performs much better than CPU, despite the data transfer cost and irregularity. This is because of the proposed optimizations discussed in §4.4.2.

Execution Time Breakdown. Using synthetic Erdős-Renyi graphs of varying sizes and degrees, Table 4.3 shows the execution time breakdown of end-to-end workload. The training and testing times are reported for link prediction classifier. A similar trend follows for node classification. Note that Table 4.3 shows per-epoch training time; the actual number of training epochs is dependent on other hyperparameter values (*e.g.,* batch size, learning rate, and rate decay).

There are two main insights here. *First, the training time dominate an end-to-end execution time of the workload.* The motivation of examining end-to-end workload time breakdown is that in a real-world deployment, the graph evolves over time. With this evolution, an entire pipeline needs

[2]PyTorch API uses workers for parallel data-loading, which spawns multiple processes replicating the memory space.

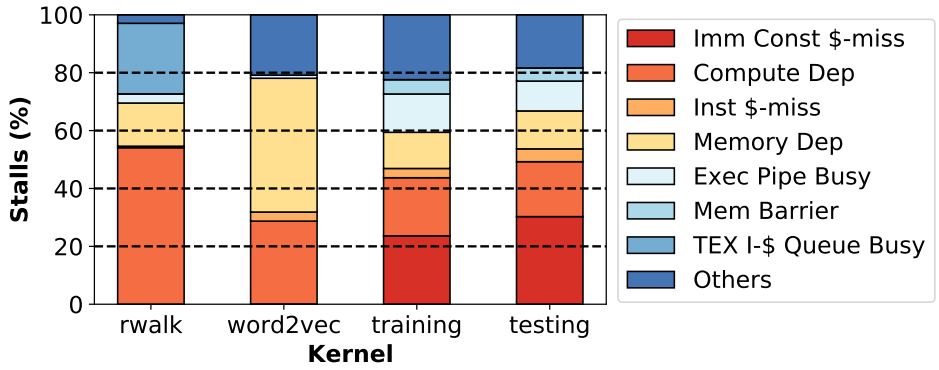


Figure 4.10: Characterization of stalls in different kernels on a GPU. There is a diversity of stalls observed across kernels; most stalls are caused by immediate constant cache (IMC) misses, and compute and memory dependencies.

to run to account for new nodes/connections. This study shows that optimizing classifier training would yield maximum benefits in reducing the end-to-end workload time. *Second, the execution times of classifier training/testing increase monotonically with the graph size.* To understand this performance further, we compare the *testing time per instruction* for modeled pipeline and VGG. This comparison finds that per-instruction execution time of random walk-based training is *37.4*× *slower* than VGG. We believe this is because of discrepancy in the matrix sizes. For example, the largest layer size in VGG is *3136*× larger than the largest layer in the studied pipeline limiting its potential for parallelism. Both applications are modeled using PyTorch, which internally calls GEMM kernels. While the performance of GEMM kernels are highly optimized for popular network sizes (*e.g.,* VGG), our study shows that there is a significant room for improvement for other network sizes.

Cross-platform Performance Comparison. Table 4.3 also compares the CPU and GPU performance. *GPU implementations outperform its CPU counterpart at large graph sizes.* This is not surprising because CPU-GPU data transfer time dominates computation time with small graphs. With large graph sizes, this time is amortized over longer, more efficient GPU computation, making it faster than the CPU. Additionally, the workload irregularity hurts GPU performance causing divergent thread pools and non-coalesced load operations.

Execution Bottlenecks. Finally, we perform a detailed microarchitectural analysis to charac-

terize stall cycles of different kernels. We perform this analysis on a large synthetic graph with 10M nodes and 200M edges. We use the GPU for this analysis because of its superior performance. Fig. 4.10 shows the characterization of stalls in terms of (from top to bottom on the legend): 1) immediate constant cache (IMC) misses, 2) compute dependencies (unresolved register dependencies because of long fixed-latency compute instructions), 3) instruction cache misses, 4) scoreboard dependencies on L1TEX operation, 5) execution pipe and MIO (memory I/O) instruction queue busy, 6) memory/CTA (cooperative thread array) barrier, 7) L1TEX instruction queue busy, and 8) others.

We observe two primary insights. *First, each kernel exhibits unique hardware characteristics and stall cycles.* For example, major causes of stalls in the random walk, word2vec, and classifier training/testing are compute dependencies (*i.e.,* 54.1%), memory dependencies (*i.e.,* 46.2%), and IMC cache misses (*i.e.,* 23.6/30.6%), respectively. As a result, no one optimization strategy can significantly speed up all workload phases, and kernel-wise investigation is necessary.

Second, on average, 65.5% of stall cycles across kernels are caused by IMC cache misses, and memory and compute dependencies. For the *random walk kernel*, the TEX I-cache queuing delay and compute dependencies cause the majority of the stall cycles. TEX I-cache stall is caused by the frequent control flow divergence as a result of the workload imbalance in sampling vertex neighborhoods. This sampling involves several long fixed-latency compute instructions (see Eq. (2.1)), causing compute dependencies. The memory dependency stall is relatively low because a large portion of the work performed for a single vertex exhibits spatial locality. The *word2vec kernel* is mostly bounded by a significant portion of memory dependencies. This is because this kernel fetches and updates the model weights by sliding through a vertex window. The vertex window being updated is dependent on the random walk result, which contains a random set of vertex IDs, generating irregular memory accesses. The *training and testing phases* show a similar stall distribution, which is attributed to small dimensions of our kernels [184], launching a small number of warps. This is further corroborated by the SM utilization for training/testing classifier being less than 10%. Therefore, loading immediate data has low reuse, causing high stall rates.

4.7 Discussion

This section discusses the employment of this framework to conduct optimization studies and incorporate new tasks.

4.7.1 Optimization Opportunities

For algorithm designers. In this work, we leverage the forward neural network for learning (§2.5.2) as a basic model for the workload analysis. It can be easily replaced by more advanced neural network architectures such as ResNet [90] or DenseNet [97]. Empirically, we observe at least $\sim 2\%$ accuracy improvement for link prediction using ResNet, and we leave the detailed investigation for future work.

For PyTorch framework designers. As briefly discussed in §4.6.2, the PyTorch framework uses multi-processing to employ multiple data loading workers. This significantly increases the memory consumption of the workload and hurts scalability. Multi-threading support with optimized memory usage will significantly improve the classifier performance.

For GEMM library designers. As shown in §4.6.2, training time per instruction of the modeled pipeline is 37.4$\times$ slower than VGG. This is owing to the differences in matrix sizes, and low-level demand-based math library optimization model. Optimizing the GEMM kernel performance for matrix sizes used in our pipeline can improve the performance of classifier training/testing by one-to-two orders of magnitude.

For compiler and hardware designers. Based on the execution stall characterization shown in §4.6.2, compiler optimization techniques such as operator fusion, loop interchange, and data structure changes can alleviate kernel launch and data transfer overheads. Additionally, compiler-based blocking, graph partitioning, and tiling [32] can improve memory performance. Furthermore, employing domain-specific hardware acceleration can significantly optimize this workload. The word2vec and classifier phases are similar to traditional deep-learning pipelines, hence, mapping them to an already existing accelerator [110] would be sufficient. However, the random walk kernel

```
1.  #include </* std header files */>
2.  #include <rwalk.h>
3.  #include <word2vec.h>
4.  #include <data_preproc.h>
5.  #include <model.h>
6.  #include <classifier.h>
7.
8.  int main( args ) {
9.    // Call graph reading API
10.   compute_rwalk( ... );
11.   word2vec( ... );
12.   data_preproc( ... ); // Implement data_preproc.h
13.   model_train( ... );  // Modify model.h, classifier.h
14.   model_test( ... );   // Modify model.h, classifier.h
15.   // Memory cleanup
16.   return 0;
17. }
```

Figure 4.11: Sample source code for incorporating new tasks.

exhibits significantly different characteristics and bottlenecks than traditional graph traversals (*i.e.,* presence of complicated compute primitives as shown in Eq. 2.1). This calls for exploring a novel accelerator design for the random walk kernel. This design must focus on optimizing both the compute pipeline for long-latency arithmetic and floating point operations, and the memory system to speed up data-dependent loads for traversing sparse graph data structures (*e.g.,* [252]).

4.7.2 Incorporating New Tasks

While this work presents two important graph learning tasks used in several application domains, our framework can be easily extended to realize other tasks. For example, if a user wants to implement link property prediction (*i.e.,* predicting edge labels), Fig. 4.11 shows the modification of main source file that calls different pipeline stages. A user can re-purpose random walk and word2vec implementations by simply calling functions shown in lines 11 and 12. As the step of preparing classifier data is unique to each task, a user has to implement an appropriate data preparation step. Finally, a classifier containing neural network model, training, and testing loops can be incorporated by modifying already implemented modules in our framework.

4.8 Chapter Conclusion

This work presented high-performance implementations of two important graph learning tasks on continuous-time dynamic networks, optimized individually to run both on the CPU and GPU. We used a scalable *random walk*-based algorithm for learning node embeddings of a graph. Based on these implementations, we conducted an in-depth performance analysis from both algorithmic and hardware fronts. The algorithm-focused study presented a rich trade-off space between prediction accuracy and runtime complexity. The hardware-focused investigation analyzed different phases of the application to find their instruction type diversity, thread scalability, execution time breakdown, and execution bottlenecks. Based on these insights, we made recommendations to further optimize the workload performance for designers of algorithms, ML frameworks, GEMM library, compiler, and hardware. The proposed implementations will be open-sourced to the broader research community to encourage further investigation.

CHAPTER 5

Accelerating Graph Pattern Mining

This is a collaborative work with H. Ye, Y. Yang, L. Belayneh, K-Y Chen, D. Blaauw, T. Mudge, R. Dreslinski.

Graph Pattern Mining (GPM) algorithms are used in numerous applications, including bioinformatics [47], cyber-security [69, 203], social network analysis [255, 260], and spam detection [134]. Despite their prevalence, GPM workloads are severely stalled on modern hardware platforms [29, 40, 280]. A majority of this performance slowdown is attributed to the irregular memory and complex data-dependent branch instructions used in set intersection and difference operations that dominate GPM workload execution times.

Prior hardware works have addressed the inefficiencies of GPM workloads either by proposing domain-specific accelerators [40, 280] or Near Data Processing (NDP) [29]. These works, however, can be significantly improved. While accelerators like FlexMiner [40] employ application-specific control and data paths, the general-purpose nature of their memory subsystems suffer from unnecessary data movement caused by GPM algorithms. On the other hand, SISA [29] optimizes GPM software by using a set-centric ISA and improved intersection algorithm. SISA, however, maps GPM computation to generic NDP architectures, *e.g.,* Ambit [225], without specialization. Therefore, GPM performance can be further improved by employing domain-specific techniques to design NDP architectures. To best design a domain-specific NDP solution, it is important to first understand the unique characteristics of GPM workloads.

To this end, we conduct a systematic characterization of GPM workloads to understand their

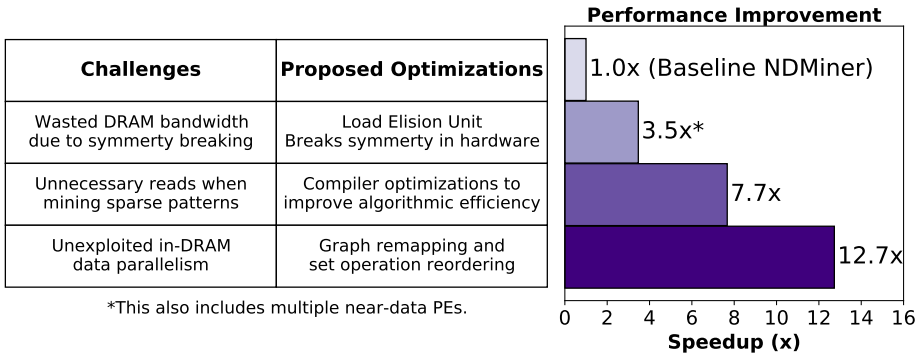


Figure 5.1: NDMiner optimizations and corresponding performance improvements inspired by the challenges of accelerating GPM workloads. Optimizations are cumulative as the bars move down.

sources of inefficiencies. This leads to four unique takeaways. First, because of the irregular graph data layout in memory, GPM workloads read data from different DRAM banks to compute set operations. Second, the symmetry breaking optimization used in modern GPM workloads discards most vertices fetched from memory in each iteration, resulting in cache pollution and wasted DRAM bandwidth. Third, sparse pattern mining algorithms perform several redundant reads and computations, leading to low algorithmic efficiency. Fourth, the size-limited memory controller queue does not allow GPM workloads to fully utilize internal DRAM data parallelism.

In this work, we present NDMiner—an NDP architecture to accelerate GPM workloads. In addition to tapping the abundant in-memory data bandwidth, the goal of this design is also to exploit presented domain-specific insights for optimization. NDMiner integrates low-cost compute units within a DIMM-based DRAM technology to effectively execute costly set operations in GPM. To support NDP operations, we also present a hardware-software interface that (a) extends the host ISA to include NDP instructions, (b) transforms GPM source code to use these NDP instructions, and (c) extends the memory controller design to orchestrate in-DRAM compute.

We further optimize NDMiner using domain-specialization as shown in Fig. 5.1. First, NDMiner integrates a new **load elision unit** in hardware to alleviate the DRAM bandwidth wastage due to symmetry breaking. This unit terminates unnecessary loads by breaking symmetry in hardware. Second, NDMiner employs **compiler optimizations** to improve the algorithmic efficiency of sparse pattern mining algorithms. This avoids redundant data loads and compute operations by fusing

multiple loops into composite set operations and hoisting loop invariant computations out of the loops. We also present how to map these computations to NDP hardware. Third, NDMiner reorders set operations at runtime to exploit internal data parallelism in DRAM. To make this reordering possible at low-cost, we first propose a novel **graph data remapping** scheme in DRAM. Based on this remapping, we design a new **vertex ID–based reordering** hardware that examines a large window (*e.g.,* 1024 entries) of set operations and reorders them to insert requests into a size-limited memory controller. The goal of this reordering is to exploit bank, rank, and channel-level parallelism in DRAM.

We rigorously evaluate NDMiner using seven GPM algorithms that mine cliques, user-defined subgraphs, and motifs on five real-world graphs. The input patterns contain a mix of both sparse and dense patterns. We first evaluate the effectiveness of various design optimizations by comparing NDMiner configurations with a baseline NDP architecture that integrates one set operation unit per channel. As shown in Fig. 5.1, proposed optimizations significantly improve the performance of this baseline design by 12.7× and reduces energy consumption by 5.1×, on average (more results in §5.7). We also compare NDMiner with the state-of-the-art GPM software (*i.e.,* GraphPi [232] and Pangolin [42]) and hardware (*i.e.,* FlexMiner [40]). We show that, on average, NDMiner significantly outperforms software and hardware baselines by 6.4× and 2.5×. Post-synthesis estimation of proposed circuits shows that NDMiner achieves these improvements at a negligible area cost.

In summary, we make the following contributions.

- A detailed analysis of GPM workloads uncovering new opportunities for performance optimization.
- *Load elision unit:* a novel design that breaks symmetry in hardware to avoid unnecessary loads.
- *Compiler optimizations:* a collection of software techniques and corresponding hardware mapping to reduce redundant loads and computations in sparse GPM.

- *Graph remapping and set operation reordering:* novel techniques to reorder computation in GPM to exploit internal data parallelism in DRAM.
- *NDMiner:* an end-to-end system that combines aforementioned optimizations that significantly improves the performance of the state-of-the-art GPM hardware accelerator by 2.5×, on average, at negligible silicon cost.

5.1 Near Data Processing Background

Near Data Processing (NDP)[1] improves the performance of memory bound workloads by reducing the amount of costly off-chip data transfers and exposing high internal memory bandwidth to compute units. The early efforts in this direction date back to the '90s [68, 80, 185, 194, 195] that integrate logic units in DRAM. More recent NDP architectures include computing in DRAM [7, 27, 55, 113, 133, 288, 299] and emerging memory technologies [45, 147, 228, 240, 249].

NDP proposals can be broadly classified into three categories based on the proximity of compute units from data. This classification is crucial to determining design choices while designing novel NDP architectures. Approaches similar to MAGIC [249] process data within a memory mat/subarray without reading them out. Such proposals enjoy high internal data bandwidth if the operands are aligned in two memory rows/columns. Other approaches process data at local/global row buffer (*e.g.,* a recent industrial proposal from Samsung [133]). While these proposals do not require the operands to be aligned within memory rows, they can be best utilized when the operands are present in the same bank. Although it is possible to move data internally within the memory from one bank to another using RowClone [224], frequent data movement can limit the benefit of near data processing. Lastly, other proposals place computation within the buffer chip or logic layers of the memory (*e.g.,* RecNMP [113] for DIMM, Teserract [7] for HMC). These approaches can avail data from different banks, however, their bandwidth is limited by the data acquisition bandwidth at the buffer chip or the TSVs in 3D DRAM. In sum, **where** to place compute units within memory

[1]Without losing generality, we refer to computing in/near memory approaches to Near Data Processing (NDP).

depends on the workload characteristics.

5.2 Finding Optimization Opportunities For GPM

This section presents unique GPM workload characteristics to motivate NDMiner design. We divide these findings into well-known GPM characteristics and new findings based on our profiling results.

5.2.1 Well-Known GPM Characteristics

Prior optimization works [29, 40, 210, 280] find several unique characteristics of GPM workloads. We list these well-known characteristics below.

Takeaway 1. Set intersection and difference operations dominate the end-to-end execution times of GPM workloads.

Takeaway 2. GPM workloads use simple arithmetic compute instructions (*e.g.,* shape count increments) that do not contribute to stall cycles.

Takeaway 3. The irregular memory accesses and their dependent control flow operations are the major sources of bottlenecks in GPM workloads.

Takeaway 4. GPM algorithms mostly use read-only data structures offering the opportunity for massive parallelism without needing synchronization.

5.2.2 Novel GPM Characteristics

In addition to validating well-known characteristics of GPM workloads, this work finds the following novel characteristics that we employ for NDMiner hardware design.

Distribution of input sets in memory. To better understand the workload behavior of GPM, we examine the memory locations of set operation inputs used in computing difference and intersection. Fig. 5.2 shows this distribution classified into four categories: (a) same bank, (b) different banks in the same bank group, (c) different bank groups on the same rank, and (d) different ranks. The figure

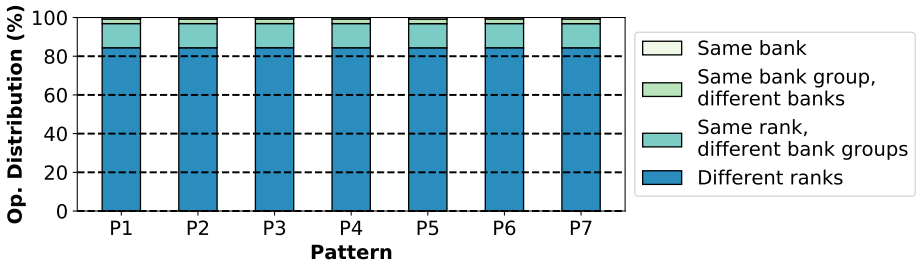


Figure 5.2: Distribution of locations of set operation inputs showing that GPM workloads mostly fetch operands from different banks.

	Dense Patterns			Sparse Patterns		Mixed Patterns	
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**
wiki-vote	2.4%	1.2%	0.7%	37.8%	5.9%	26.1%	47.8%
pokec	1.3%	1.0%	0.9%	14.6%	1.5%	25.5%	36.5%
patents	4.0%	3.0%	2.6%	13.8%	6.4%	26.4%	42.7%
livejournal	2.5%	5.4%	6.4%	45.4%	7.1%	26.1%	39.9%

Table 5.1: Percentage of vertices utilized in the next search levels out of all fetched vertices because of symmetry breaking.

shows that a majority of the time, the set operands are present in different banks. This result offers insight into where to best place NDP compute logic to optimize GPM workloads.

Takeaway 5. GPM workloads fetch data from different DRAM banks to compute set intersection and difference operations.

Adverse effect of symmetry breaking. As presented in §2.4, advanced GPM algorithms use symmetry breaking to avoid redundant computation. For triangle counting, this is reflected in lines 6 and 11 of Algorithm 1. In effect, only a fraction of the computed neighborhood or intersection results (lines 4 and 9) are used in the next phase of computation, which we call the *filter operations*. To understand the effect of filter operations, we calculate the fraction of vertices used in the current GPM iteration out of all the vertices fetched in the previous iteration to compute neighborhoods/set operations. Table 5.1 shows that 66.5% of the vertices are discarded. Intuitively, dense input patterns utilize a smaller fraction of vertices compared to sparse patterns. This is because dense pattern finding algorithms employ more constraints than their sparse counterparts because of their connectivity structure. While this improves the efficiency of GPM algorithms by avoiding redundant

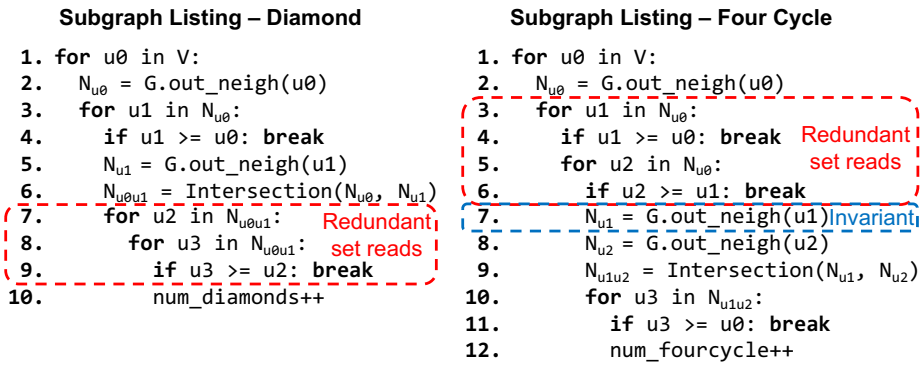


Figure 5.3: Examples of redundant load and computation in sparse pattern mining algorithms (*i.e.,* subgraph mining for diamonds and four cycles).

computation, it pollutes the CPU caches and squanders useful DRAM bandwidth.

Takeaway 6. Symmetry breaking discards most vertices fetched from memory in each iteration, leading to cache pollution and wasted DRAM bandwidth.

Redundant reads and computations for mining sparse patterns. Sparse patterns are defined as graph patterns where most nodes are not connected to all other nodes. Conversely, fully connected patterns (*e.g.,* cliques) are called dense patterns. Fig. 5.3 shows pseudo-code for mining two sparse patterns, *i.e.,* diamond and four cycle. The figure shows that, for diamond mining in lines 7–9, vertices $u2$ and $u3$ are found by iterating over the same candidate sets, *i.e.,* N_{u0u1}. The same trend exists for vertices $u1$ and $u2$ in four cycle mining algorithm (lines 3–6). Furthermore, line 7 of four cycle mining algorithm shows that neighborhood computation N_{u1} is invariant to $u2$. These properties of sparse GPM lead to redundant reads and computation. While we use two example shapes to demonstrate this concept, this redundancy is common across a wide range of sparse GPM algorithms.

Takeaway 7. Sparse pattern mining algorithms involve redundant reads and computations.

Set Operation reordering opportunity. While most prior GPM works typically process vertices in an input graph in the order of their IDs, we design an experiment to find if there is an opportunity to gain performance by reordering the memory accesses in an input graph. First, we reorder an

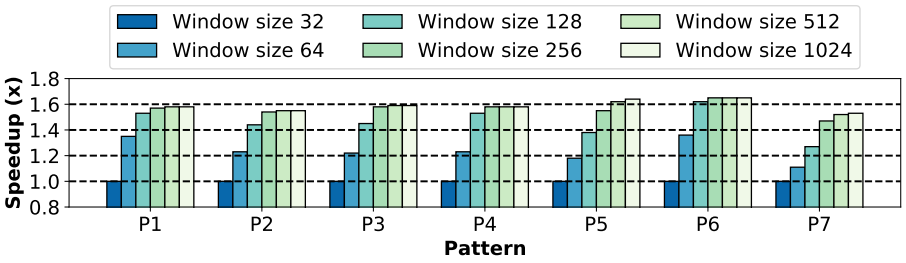


Figure 5.4: Speedup of GPM workloads for different memory controller reorder window sizes. Results are normalized to 32 window size.

input graph in software by using three graph reordering techniques, *i.e.,* DegreeSort, HubCluster, and HubSort based on a prior work [21]. This, however, does not affect the performance of GPM workloads. Second, we reordered the set operations computed in hardware by artificially increasing the memory controller read queue size. Fig. 5.4 shows the effect of using larger memory controller reordering window sizes normalized to a realistic size of 32 on GPM performance. The figure shows that a larger reordering window improves the workload performance by up to 1.6×. This is because a smaller reordering window is congested by the requests to the same bank, reducing reordering and data-parallelism opportunity. Larger windows, on the other hand, find requests to better exploit data-parallelism by sending concurrent requests to multiple banks, ranks, and channels.

Takeaway 8. GPM workloads do not fully exploit abundant data-parallelism in DRAM because of size-limited memory controller queues.

5.2.3 Why NDP for GPM?

As discussed in §5.1, NDP alleviates the performance and energy overheads of costly off-chip data transfers between the CPU and DRAM. This can be used to alleviate the wasteful data transfer in GPM algorithms because of symmetry breaking (Takeaway 6). NDP has the potential to reduce cache thrashing and energy wasted on off-chip data transfer. Additionally, NDP exposes high internal memory bandwidth that can be exploited by GPM algorithms as they offer ample parallelism (Takeaway 4).

In-DRAM compute parallelism can be best utilized by simple compute units that can be integrated

within the memory in a cost-effective manner. GPM algorithms mostly use adder and comparator logic to perform most of their computations (Takeaway 2). The simplicity of these operations allows their cost-efficient integration within memory. Resolving load-dependent control flow operations at NDP precludes the need for using expensive branch resolution mechanisms on the CPU. Moreover, irregular accesses to graph data structures resulting in high memory latency and/or bandwidth [172, 252] can be better serviced near memory at a low latency and high available bandwidth, addressing the two main bottlenecks in GPM workloads (Takeaway 3). *In summary, NDP is an attractive candidate for accelerating GPM workloads.*

5.2.4 How To Best Design NDP For GPM?

The next task is to find where should we to compute unit within the memory? As discussed in §5.1, the best place depends on the workload characteristics. As set intersection/difference operations dominate the execution time of GPM workloads (Takeaway 1), we offload them to NDP units. Furthermore, Takeaway 5 shows that GPM workloads mostly fetch data from different banks. Placing compute units inside the bank would incur significant in-DRAM data transfer. Therefore, we make a design decision to place the compute units at the buffer chip of DIMMs in NDMiner. While we use DIMM in this work, similar design principles can also be applied to the logic layer of HMC/HBM.

5.3 Hardware-Software Interface

This section discusses the hardware-software interface of NDMiner to support NDP operations for GPM acceleration.

5.3.1 Supported NDP Operations

Based on Takeaway 1, NDMiner offloads set intersection and difference operations to the NDP units. Additionally, the primary goal of NDP design is to alleviate the cost of data movement in GPM

workloads. As presented in Takeaway 6, symmetry breaking results in wasteful data movement. By using NDP, it is possible to identify and terminate loads filtered by breaking symmetry in hardware. This helps improving the overall efficiency of the program by eliding useless loads that prevents cache pollution. Therefore, NDMiner also offloads load elision operations to memory. In total, NDMiner supports five NDP operations: (a) complete set intersection, (b) complete set difference, (c) filtered set intersection, (d) filtered set difference, (e) load filtered set.

5.3.2 ISA Extensions

```
filtered_intersect   addr0, len0, addr1, len1, u_th // u_th=-1 if no filter
filtered_difference  addr0, len0, addr1, len1, u_th // u_th=-1 if no filter
      filtered_load  addr0, len0, u_th              // u_th=-1 if no filter
```

Figure 5.5: Host ISA instructions to support NDP.

To enable software to communicate NDP operations to memory through the host CPU, NDMiner introduces three instructions in the ISA as shown in Fig. 5.5. To support symmetry breaking in hardware (more details in §5.5.1), these instructions support filtering of input sets. A threshold vertex ID is specified (*i.e.,* `u_th`) that is determined at runtime by the CPU and communicated to the NDP units. If load elision is not applied, the values of `u_th` is specified as -1. The memory address ranges of input sets are indicated by the base address and length of sets. Similar to recent academic NDP proposals [7, 113, 288] and an industrial product [133], we assume that the data allocated for NDP uses physically contiguous memory blocks. Contiguous mapping ensures that NDP instructions only have to translate one address, and the rest of the addresses can be obtained using the address range, even if the addresses rarely cross the OS page boundaries. This, however, is not a fundamental limitation of NDMiner as it is also compatible with the current OS page mapping scheme, which would rarely require more than one address translations for an NDP instruction when input sets in that instruction span multiple pages.

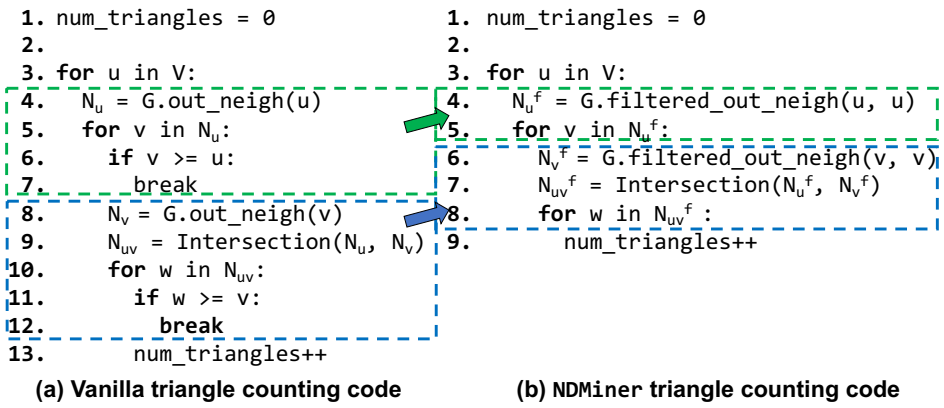


Figure 5.6: Code transformations to make use of NDP instructions.

5.3.3 Programming Model

To utilize aforementioned ISA instructions, an NDMiner compiler transforms the source code of GPM workloads. First, the compiler analyzes the source code to extract the instances amenable to NDP acceleration. These instances include set operation computations, neighborhood loads, and symmetry breaking constraints. These instances are then replaced with NDP instructions. Fig. 5.6 shows an example of source code transformations, where lines in the green and blue boxes in the original source code are replaced with filtered load operations. In this workload, the intersection operation is not modified as it receives filtered neighborhoods as input (line 7 in Fig. 5.6(b)). For workloads where neighborhoods are not filtered beforehand, **`filtered_intersect`** instruction can filter sets before computing the intersection. These code transformations are translated into the primitive ISA instructions (§5.3.2) by the compiler back-end. When the host CPU decodes these instructions, they are directly forwarded to the memory controller, bypassing the cache hierarchy. Because NDMiner only processes read-only data (Takeaway 4), bypassing the cache hierarchy does not affect the correctness of the program as all the cached data is in the clean state.

5.4 NDMiner Hardware Architecture

Fig. 5.7 shows an overview of the NDMiner hardware design. Upon receiving an NDP instruction, the NDMiner memory controller front-end converts it into multiple composite loads and set oper-

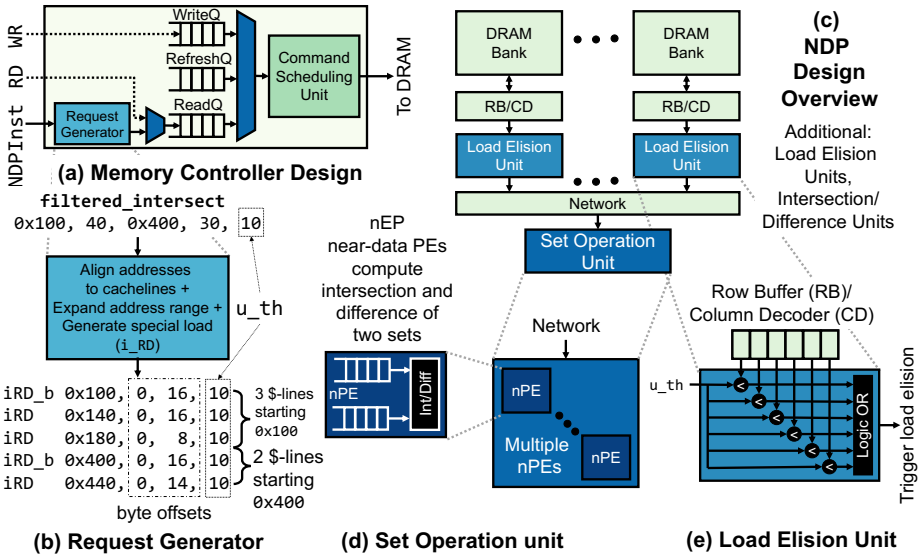


Figure 5.7: Hardware design overview of NDMiner.

ations for offloading to DRAM. This section goes over the details of NDMiner hardware design that includes the design of the memory controller, near-memory compute units, and the DRAM access protocol. NDMiner targets a minimally invasive design, where we aim to utilize the existing hardware resources as much as possible.

5.4.1 NDMiner Memory Controller Front-end Design

Fig. 5.7(a) shows the NDMiner memory controller design. We introduce a front-end logic unit called the *request generator* that converts NDP instructions into DRAM requests. This unit accepts all three instructions discussed in §5.3.2 that perform different operations. Next, we take an example of filtered intersection to describe this hardware in detail. The incoming `NDPInst` specifies base addresses of two sets as `0x100` and `0x400`, and lengths of 40 and 30, respectively. Each element in a set is 4B long; there are 16 elements in a cache line. The instruction also indicates a threshold (`u_th`) of 10, *i.e.,* the intersection result must have elements less than 10. With this information, the request generator unit first aligns the addresses to cache line boundaries, and marks the range of byte offsets to read from each cache line. This unit also creates read requests with a unique opcode (*i.e.,* `iRD`) indicating intersection operations. The figure shows two opcodes: `iRD` and `iRD_b`. The latter one marks the beginning of a cache line for a set. The generated request also contains byte offsets

and u_th as shown in Fig. 5.7(b). These requests are then enqueued into the memory controller.

5.4.2 NDMiner Memory-side Hardware Design

Fig. 5.7(c,d) show the set operation unit located at the buffer chip of DRAM based on Takeaway 5. It reads two sets from DRAM banks, and computes intersection or difference. As shown in 5.7(d), the near-data Processing Engines (nPEs) employ buffers to temporarily store the cache line of one set while the other set is being read from DRAM. After the first cache lines of both sets are read, simple comparator logic starts computing intersection/difference result. For each operation, the nPE is blocked until its completion. We name this design choice **NDMiner-Base**, where NDMiner employs one nPE per DIMM. While fetching two sets from different banks, NDMiner-Base can exploit as much as 2× compute bandwidth compared to moving data off-chip.

5.4.3 NDMiner Command Scheduling

This unit dequeues requests from the memory controller and issues commands to memory. In addition to issuing regular DRAM requests, the NDMiner command scheduler also issues NDP requests using unique opcodes. To support NDP at a minimal hardware overhead, NDMiner communicates compute operations in terms of DRAM commands, as opposed to a prior work [113] that issues composite operations.

All of the NDMiner operations are performed in conjunction with memory reads. For example, an intersection operation first reads operands from memory. Therefore, NDMiner issues compute commands following row activate and prior to row precharge. To issue commands for requests generated in Fig. 5.7(b), first, an ACT command opens a DRAM row. Then, an iRD_b command blocks an nPE for intersection and reads the first cache line to the set operation unit. On the address and data buses, the memory controller sends row/column addresses along with metadata for computation (*i.e.,* byte offsets and vertex threshold) in a time-multiplexed fashion. This obviates the need to add extra buses to support NDP.

5.5 Design Optimizations

To further improve the performance of NDMiner, this section presents novel optimization techniques.

5.5.1 NDMiner-LoadElision: Eliding Unnecessary Loads

Based on Takeaway 6, symmetry breaking results in wasted DRAM bandwidth. To alleviate this effect, we propose load elision unit (LEU) that **breaks symmetry in hardware**. Fig. 5.7(c,e) show near-memory compute logic for eliding loads. This unit compares data values read from DRAM with `u_th` and raises a signal when further loads need to be terminated. It employs a set of comparators as shown in Fig. 5.7(e). If a neighbor value read is higher than `u_th`, it triggers load elision. Because this unit directly uses cache line values read from DRAM, it is placed at the column decoder output. With 16 banks per rank and 2 ranks in a DIMM, the load elision unit can exploit the compute bandwidth as high as $2 \times 16 = 32\times$ on a single DIMM compared to moving data off-chip. We name this design choice as **NDMiner-LoadElision**. While NDP operations do not transfer data off-chip, we use the data bus response to indicate the termination of reads when the load elision is triggered. The memory sends a pre-encoded response (*e.g.,* `ff`) back to the memory controller indicating a load elision event. This response enables the memory controller to find the pending load requests and terminate them.

5.5.2 NDMiner-Overlap: Offloading Concurrent Instructions

With one nPE per DIMM, the near-memory set operation units can only exploit up to $2\times$ compute bandwidth compared to processing data off-chip. While this is favorable, there is still $16\times$ data bandwidth left unexploited. Moreover, a simple nPE design incurs low integration cost within the DRAM. To match the available data bandwidth, NDMiner integrates 16 nPEs per DIMM as shown in Fig. 5.7(d). We name this design **NDMiner-Overlap**, as multiple nPEs can overlap set operations. This includes (a) concurrently reading operands from multiple banks to exploit bank-level parallelism, and (b) concurrent set computation. While this is not a novel optimization,

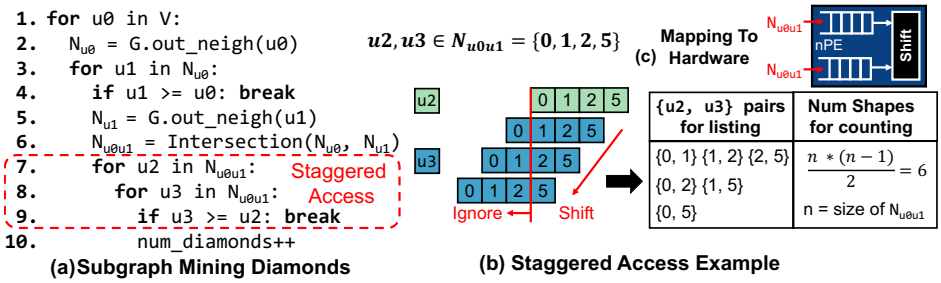


Figure 5.8: Proposed compiler optimizations and corresponding computation mapping to hardware to improve the algorithmic efficiency of sparse GPM. Consecutive loops iterating over same sets are fused to perform one set read and a composite computation (shift and record in this example).

separating this design choice from NDMiner-Base helps us understand the potential of GPM workloads to exploit in-DRAM data parallelism.

5.5.3 NDMiner-Compiler: Optimizing Algorithmic Efficiency

Based on Takeaway 7, mining sparse patterns involve redundant load and computation operations. For example, executing lines 7 and 8 in Fig. 5.8 would read N_{u0u1} several times to the NDP units redundantly loading the same data. To improve the algorithmic efficiency of these workloads, we propose the following compiler-based optimizations. First, the compiler identifies the existence of redundant reads by examining the candidate sets used in consecutive loops. As shown in Fig. 5.8(a), two loops in lines 7 and 8 iterate over the same candidate set N_{u0u1}. Furthermore, line 9 imposes a symmetry breaking constraint between $u2$ and $u3$.

Upon this identification, we propose to **fuse** these two **loops** and convert it into one set read and a **composite computation**. For example, loop fusion in Fig. 5.8(a) is converted into a staggered access of N_{u0u1} as shown in Fig. 5.8(b). Symmetry breaking constraint is the reason for this type of access pattern because $u2$ cannot be greater than $u3$. We further map this computation in hardware to nPEs, where the same candidate set is replicated in two buffers, and $\{u2, u3\}$ pairs can be found by using a shift-and-record operation. While this is useful for pattern listing algorithms, pattern counting algorithms can directly compute the number of patterns by using simple accumulation equation as shown in Fig. 5.8(b). In addition to loop fusion, our compiler pass also hoists loop invariant computations outside the loop. This includes, for example, moving N_{u1} computation in

line 7 in Fig. 5.3 before line 5 as neighborhood of $u1$ is independent of the value of $u2$. Applying compiler optimizations significantly improves algorithmic efficiency of sparse pattern mining; we name this design choice **NDMiner-Compiler**.

5.5.4 NDMiner-Reorder: Reordering Set Operations

Based on Takeaway 8, it is possible to improve the performance of GPM workloads by reordering set operations to exploit parallelism in DRAM. To further understand the reason behind this performance difference, consider an example, where Fig. 5.9(a) shows neighborhoods of selected number of nodes in a hypothetical graph. Fig. 5.9(b) shows that a traditional memory controller falls short in identifying operation reordering opportunity because of its size-limited queues. This can result in frequent bank conflicts if row IDs of queued requests are different. One straightforward way to improve performance is by increasing the size of the memory controller read queue and let the memory controller reorder a larger number of read requests. This, however, is not a practical design as it will significantly increase the latency of memory controller reordering logic, potentially hurting performance of other applications. Any other technique that uses addresses to reorder set operations would incur a similarly large overhead. Therefore, **we propose to raise the level of abstraction and reorder set operations based on vertex IDs at low cost.**

The intuition behind our proposal is to encode the vertex ID in the bank address to find a node's neighborhood. This allows us to compute the bank address of each set operation at a low-cost, obviating the necessity to decode an entire address. This can further be used to reorder operations from a *large window size* to maximize bank-level parallelism. Fig. 5.9(c,d) explain our design with an example. We propose to **remap** each node's neighborhood to different banks based on computing a simple hash function of a vertex ID. While this work uses a modulo operation to map each vertex ID to a bank, this is not a fundamental limitation, and this operation can be replaced by a more sophisticated hash function, if necessary. The row and column addresses are then encoded to have a contiguous neighborhood mappings of two vertices without overwriting each others' data. A physical address from DRAM row, column, bank, bank group, rank, and channel coordinates is

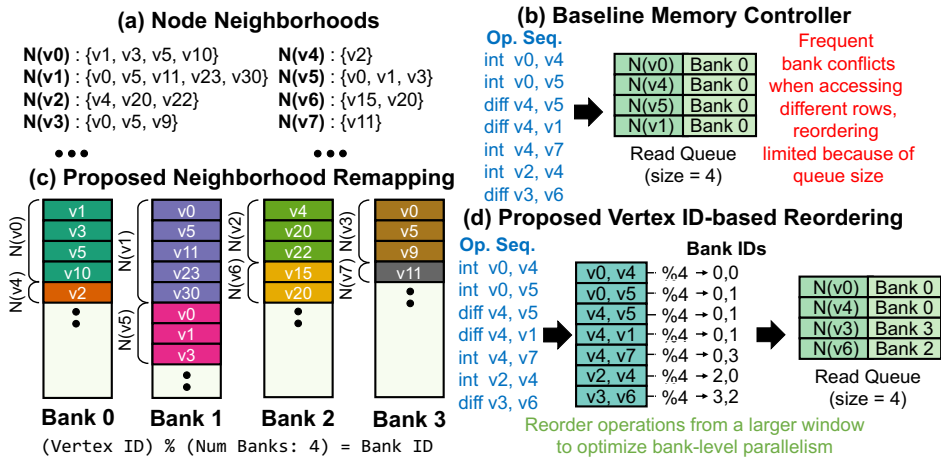


Figure 5.9: (a) Example graph's node neighborhoods, (b) baseline memory controller with a size-limited queue that leads to frequent bank conflicts when accessing different rows, (c) proposed neighborhood remapping scheme using a deterministic interleaving of neighborhoods across banks, and (d) vertex ID-based set operation reordering to exploit bank-level parallelism in DRAM.

calculated based on a prior work [201]. Fig. 5.9(c) shows the resultant mapping of nodes v0-v7's neighborhoods.

At runtime, this mapping information is used to intelligently **reorder** and selectively schedule set operations to maximize data parallelism in DRAM. Fig. 5.9(d) shows the function of reordering hardware, which takes an operation sequence as an input from the CPU and computes bank addresses of neighborhoods used in each set operation. This is computed by simply applying a modulo function to vertex IDs. In hardware, modulo operation translates to simply selecting a few low significant bits, which can be executed in parallel efficiently. Based on the bank IDs, set operations are reordered to have distinct bank IDs in the consecutive operations in the reordered sequence. Based on this reordering, a subset of this operations are offloaded to the memory controller based on the empty slot in the queues. Because the proposed vertex-based reordering scheme enables address identification at an extremely low cost, it is possible to select operations from a much larger window size compared to a size-limited memory controller queue. As presented in Fig. 5.4, this results in significant performance improvement. We name this design **NDMiner-Reorder**.

DRAM Specification
DDR4-3200, 4Gb ×8, 4 Channels × 1 DIMM × 2 Ranks, 32-entry RD/WR queue, FR-FCFS, Skylake address mapping [202]
DRAM Timing Parameters
tRCD=22, tCL=22, tRP=22, tBL=4. tCCD_S=4, tCCD_L=10, tRRD_S=4, tRRD_L=8, tFAW=34, tRC=78
DRAM Energy Parameters
IDD0=52mA, IDD1=69mA, IDD2N=37mA, IDD3N=52mA, IDD4R=168mA, IDD4W=150mA, VDD=1.2V, VPP=2.5V [165]
nPE, LEUs, and Reordering Unit Parameters
16 nPEs and 32 LEUs per channel @ 2GHz on DRAM buffer chip, 1024-entry vertex ID-based reordering unit on CPU

Table 5.2: DRAM Parameters and Configurations.

5.6 Evaluation Methodology

5.6.1 Baseline CPU Hardware Platform

For the software baselines, we use an AMD EPYC 7742 processor with 64 physical cores (128 SMT threads). The aggregate Last Level Cache (LLC) size is 256MB. The main memory in the system is 4-channel DDR4-3200 with a 512GB capacity. A prior work [40] shows that enabling hyperthreading for GPM workloads slows down performance scaling due to cache contention. Therefore, we use 64-thread implementations of our software baselines.

5.6.2 Simulation Infrastructure

We model the cycle-accurate NDMiner performance using Ramulator [118]. The configuration of modeled memory system is shown in Table 5.2. We generate a trace of NDP instructions to feed into Ramulator and model the NDMiner hardware modifications presented above. As neighborhood set load, intersection, and difference operations take a majority of workload execution time, we model this computation in Ramulator and compare it with other baselines. Notably, in addition to this computation, GPM algorithms perform other simple computations including shape count increments. These operations are left to be performed efficiently using a multi-threaded host CPU. To estimate the latency, energy consumption, and area overhead of NDP logic, we model NDMiner using Verilog HDL and synthesize using a commercial 28nm technology node on the Synopsys Design Compiler. For vector-based power estimation, we use Synopsys PrimeTime. While it is possible to clock the nPEs at a higher frequency in a logic process, we conservatively clock them at a lower frequency as they use slower transistors in DRAM technology.

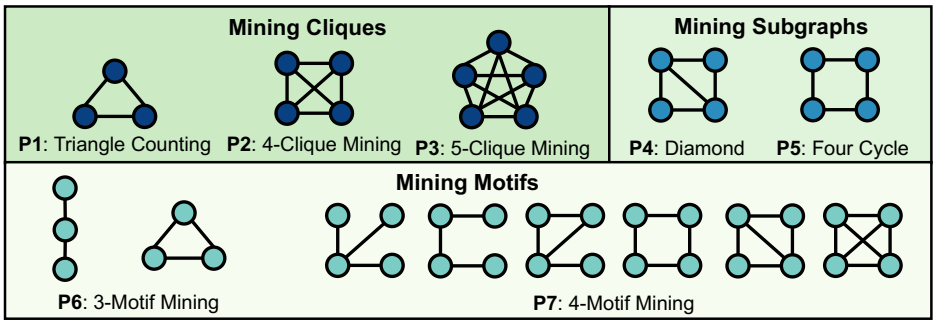


Figure 5.10: Input graph patterns used for evaluation.

Graph	#Vertices	#Edges	Avg degree	Description
wiki-vote (wi)	7.1k	103.7k	14.6	Voting network
pokec (po)	1.6M	30.6M	19.1	Social network
patents (pa)	3.7M	16.5M	4.4	Citation network
livejournal (lj)	4.0M	34.7M	8.7	Social network
orkut (or)	3.1M	117.8M	38.1	Social network

Table 5.3: Real-world graph data sets used for evaluation.

5.6.3 Algorithms and Datasets

Algorithms. We mine seven patterns **P1–P7** of varying sizes and connectivity as shown in Fig. 5.10. The first six patterns are the same as what a prior work FlexMiner [40] used. In addition, we also use 4-motif counting (P7) for comprehensive evaluation. Among these patterns, the cliques (P1–P3) are dense, fully connected patterns, and P4–P5 are sparse patterns. Motif counting counts all possible patterns with a specified number of vertices (*i.e.,* two patterns for 3-MC and six patterns for 4-MC) that includes both dense and sparse patterns. While we choose these five patterns for evaluation, NDMiner is agnostic to any specific pattern, and it can work well for any arbitrary user-defined pattern. As detailed in prior works [29, 40], the simulation times for mining large patterns is quite high (*e.g.,* days to weeks); hence we mine patterns of up to five vertices.

Datasets. We use five real-world graph datasets for evaluation as shown in Table 5.3. These datasets are diverse in terms of their sizes from small (wiki-vote) to large (orkut), and connectivity (*i.e.,* average degrees). Notably, the amount of simulation time grows exponentially with the graph size. Hence, we use similar sized datasets as prior works [40, 280]. We set a simulation timeout of 120 hours (five days) and do not include the results for workloads that do not finish execution in this time. This mostly includes mining large number of patterns (P7) on large datasets with slower

	Num nPEs per channel	Load Elision	Loop Fusion	Op Reorder
NDMiner-Base	1	✗	✗	✗
NDMiner-LoadElision	1	✓	✗	✗
NDMiner-Overlap	16	✓	✗	✗
NDMiner-Compiler	16	✓	✓	✗
NDMiner-Reorder	16	✓	✓	✓

Table 5.4: NDMiner configurations.

baselines.

5.6.4 NDMiner Configurations

To present the benefit of proposed optimization techniques, we compare NDMiner configurations listed in Table 5.4.

5.6.5 State-of-the-art Baselines

We also rigorously compare NDMiner with the following software and hardware baselines.

GAPBS+GraphPi (software) extracts algorithms from GraphPi [232] including optimized vertex traversal schedules and symmetry breaking constraints and implements them onto GAPBS [26] data structures. The purpose of this baseline is to evaluate GraphPi algorithms on optimized GAPBS graph data structures without framework-related overheads.

GraphPi (software) uses vanilla open-source GraphPi [232].

Pangolin (software) is a collection of open-source benchmarks [43] based on the implementations of state-of-the-art GPM frameworks including Pangolin [42] and Sandslash [41].

FlexMiner (hardware) is based on a GPM hardware accelerator [40]. To obtain FlexMiner execution time, we run the CPU baseline code open-sourced by authors in GraphMinerBench [43] on an Intel i9 machine (same as used in their work), and multiply speedup factors reported in the work for commonly evaluated algorithms and datasets.

SISA and IntersectX (hardware). We qualitatively compare NDMiner with these baselines [29, 210] as their open-source implementations are not available.

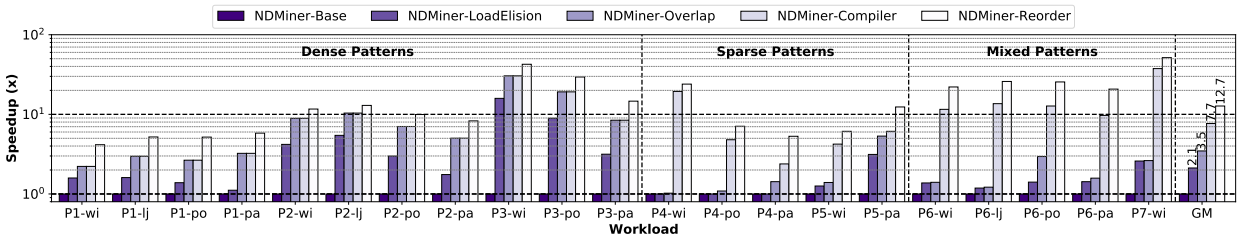


Figure 5.11: Performance comparison of NDMiner configurations showing the effectiveness of proposed optimizations. Workloads that do not simulate in 120 hours are excluded that mostly includes P7 mining. All proposed optimizations together improves the performance of NDMiner-Base by 12.7×, on average.

5.7 Evaluation Results

5.7.1 Performance Analysis

Comparison of different NDMiner configurations. We first compare the performance of various NDMiner baselines (§5.6.4) to estimate the effectiveness of proposed design optimizations. Fig. 5.11 shows the performance of NDMiner configurations normalized to NDMiner-Base. NDMiner-LoadElision outperforms NDMiner-Base by 2.1×, on average by breaking symmetry in hardware and avoiding unnecessary loads. NDMiner-Overlap further outperforms NDMiner-Base by 3.5×, on average, showing that adding extra nPEs marginally improve performance. This result also shows that merely adding 16× more NDP compute resources does not automatically offer significant performance, especially for sparse patterns. To best tap the potential of NDP, we need further optimizations.

Fig. 5.11 further shows that NDMiner-Compiler significantly improves the performance of sparse GPM algorithms (*i.e.,* P4–P7), resulting in an average improvement of 7.7×. Note that this optimization is not applicable to dense patterns P1–P3. This benefit of this optimization is attributed to the improved algorithmic efficiency, where NDMiner-Compiler avoids unnecessary load and compute operations. NDMiner-Reorder further improves the performance of GPM workloads by 12.7×, on average, compared to NDMiner-Base. This configuration outperforms all other baselines by introducing set operation reordering. This reordering fills up the size-limited memory controller queue by requests that can be serviced by different banks, ranks, and channels concurrently to

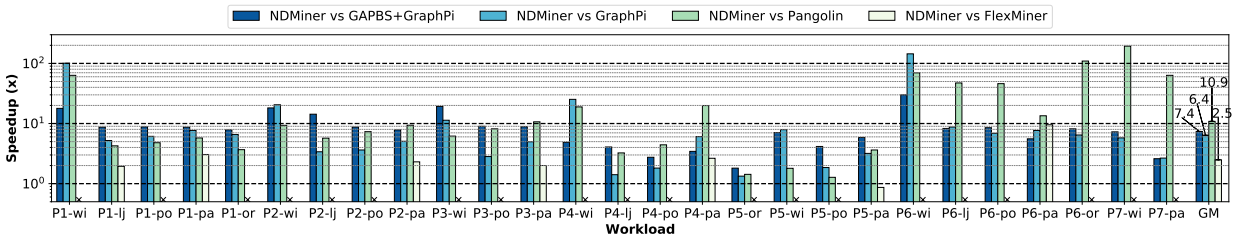


Figure 5.12: Performance comparison of state-of-the-art software and hardware baselines with NDMiner showing significant performance improvements. FlexMiner [40] is only compared against commonly evaluated datasets (others marked with "x" on x-axis). Workloads that time out are excluded.

optimize internal DRAM data parallelism.

NDMiner versus state-of-the-art baselines. Fig. 5.12 compares the performance of NDMiner with prior software and hardware optimizations for GPM. This comparison is conducted with our best-performing configuration, *i.e.,* NDMiner-Overlap. NDMiner significantly outperforms three strong software baseline, *i.e.,* GAPBS+GraphPi [26, 232], vanilla GraphPi [232], and Pangolin [42] by 7.4×, 6.4×, and 10.9×, on average. Our detailed investigation reveals that NDMiner uses the same traversal order and symmetry breaking constraints as other baselines. Therefore, these significant benefits are attributed to (a) reducing the off-chip data transfer using NDP, (b) hardware-based load elision with the knowledge of symmetry breaking constraints, (c) optimizing algorithmic efficiency of sparse patterns, and (d) exploiting high in-DRAM compute bandwidth by appropriately reordering set operation (and not because of better algorithms from GraphPi).

Fig. 5.12 also shows that NDMiner outperforms FlexMiner [40] on commonly evaluated algorithm-dataset pairs by 2.5×, on average. While FlexMiner improves GPM performance over CPU by domain-specialization, it uses a traditional memory architecture with on-chip caches and off-chip DRAM. Our profiling, however, shows that GPM workloads exhibit wasteful behavior on a traditional memory hierarchy, and can be significantly optimized by using NDP. NDMiner outperforms FlexMiner by offloading computation to NDP units, improving the algorithmic efficiency of sparse pattern mining, and reordering set operations to exploit abundant in-DRAM data parallelism.

We qualitatively compare NDMiner with SISA [29] and IntersectX [210] because of their lack of available open-source implementations. While SISA efficiently maps GPM algorithms to set

	Dense Patterns			Sparse Patterns		Mixed Patterns	
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**
Loads	4.1×	5.4×	4.9×	2.8×	1.6×	7.9×	12.7×
Comparisons	4.6×	5.0×	4.3×	1.0×	1.6×	4.6×	1.5×

Table 5.5: Reduction in loads and element-wise comparisons in set operations due to load elision and compiler optimizations. Results averaged over different datasets.

operations, it employs general-purpose NDP hardware (*e.g.,* Ambit [225]) to offload computation. NDMiner, on the other hand, employs domain-specialized NDP hardware design, circumvents unnecessary reads and computations, and reorders set operations to acquire additional performance from NDP. IntersectX optimizes GPM workloads on a CPU using a stream instruction set and its microarchitectural support. This, however, fetches data from off-chip DRAM that suffers from wasted DRAM bandwidth. Similar to FlexMiner, the performance of IntersectX can further be improved by NDMiner's domain-specialized NDP design.

Load and Computation Reduction. To better understand the performance benefits of NDMiner, Table 5.5 shows the reduction in the number of load and element-wise comparisons for computing set operations. NDMiner avoids unnecessary loads and element-wise comparisons by (a) hardware-based load elision (§5.5.1), and (b) software-based compiler optimizations using loop fusion and instruction hoisting (§5.5.3). Dense workloads only benefit from load elision that significantly improves their algorithmic efficiency. This is because dense patterns use a unique symmetry breaking constraint for each set operation, where load elision is effective. Sparse patterns, on the other hand, often compute a set operation once and reuse its result multiple times. Because each such usage might have a unique constraint, this sometimes precludes the employment of load elision because the entire set needs to be computed once. P4 (diamond) is one such pattern where intersection result is used several times with different constraints. This pattern, however, still benefits from our loop fusion technique and reduces the number of loads. Motif counting (P6-P7) algorithms mine several patterns, offering better opportunity for both load elision and compiler optimizations to be effective.

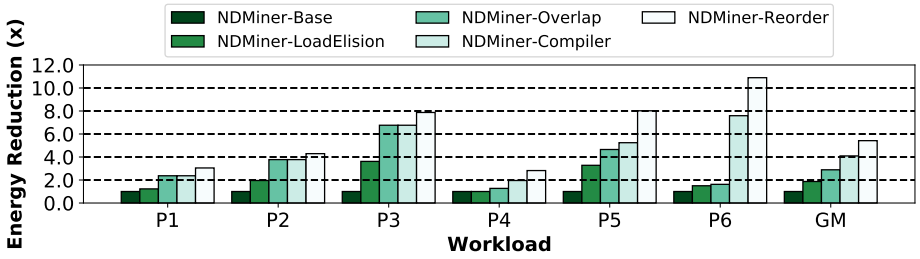


Figure 5.13: Energy consumption of NDMiner configurations normalized to NDMiner-Base on a representative patents dataset. P7 is excluded as its baseline simulation times out.

5.7.2 Energy Analysis

Fig. 5.13 compares the energy of different NDMiner configurations, normalized to NDMiner-Base, using a representative patents dataset. Energy of mining 4-motif (P7) is not reported as its simulation for NDMiner-Base times out. The figure shows that proposed optimizations improve the energy consumption of NDMiner-Base by 1.9×, 2.9×, 4.1×, and 5.1×, on average. This significant energy reduction is attributed to (a) improved memory traffic and algorithmic efficiency by load elision and compiler optimizations, and (b) reduction in static energy by speeding up the program execution by using multiple nPEs per channel and reordering set operations to exploit internal DRAM data parallelism.

5.7.3 Sensitivity Analysis

Fig. 5.14 shows the performance sensitivity of NDMiner compared to (a) different set operation reordering window sizes and (b) number of nPEs per channel. The top figure shows that increasing the window size from 1 to 4096 monotonically increases the performance by 1.6×, on average. Interestingly, there is a marginal performance increase from 1024 to 4096. The silicon and power costs, on the other hand, would increase significantly by increasing a window size by 4×. Therefore, NDMiner design employs a window size of 1024 that best trades off area and power costs with performance.

Fig. 5.14 (bottom) shows that the performance of NDMiner improves by 4.2× on average with an increase in the number of nPEs from 1 to 16. This improved performance shows the opportunity to

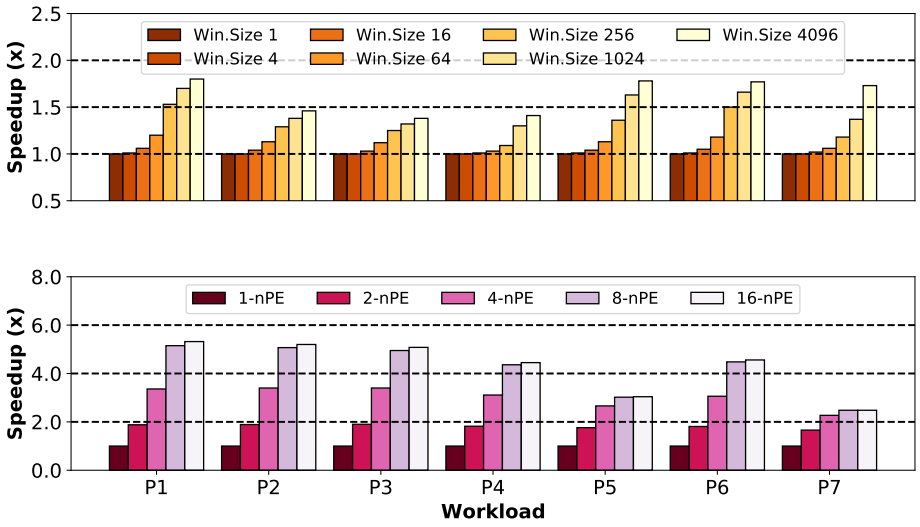


Figure 5.14: Performance sensitivity of NDMiner for different set operation reordering window sizes (top) and number of nPEs per channel (bottom) on a representative patents dataset.

	near-memory PE (nPE)	Load Elision Unit (LEU)	Operaration Reorder Unit
Location	DRAM	DRAM	CPU
Area (mm^2)	0.01237	0.00096	0.4147
Power (mW)	18.45	0.36	32.78

Table 5.6: Area and power estimates of NDMiner circuits.

overlap large portions of compute operations by availing the ample in-DRAM compute bandwidth. This trend, however, slowly saturates beyond 8 nPEs, at which point, the workload gradually shifts from being compute bounded to memory bounded. Although using 16 nPEs marginally improves performance, the area and power overhead of integrating nPEs are minimal (discussed in §5.7.4), which informs our choice of using 16 nPEs per channel.

5.7.4 Overhead Analysis

Table 5.6 shows the post-synthesis area and power overheads of NDMiner circuits. While the table shows overheads of individual circuits, NDMiner design integrates 16 nPEs and 32 LEUs in a DRAM DIMM, and one set operation reordering unit on the CPU. The area and power of NDMiner is dominated by the reordering unit as it employs two 1024-entry buffers (one to store incoming NDP instructions and the other to store reordered instructions). The cost of these hardware units, however, is negligible compared to the performance benefit they provide. Compared to a

$100mm^2$ [162] area of the DRAM buffer chip, NDMiner circuits add a minimal area overhead of 0.23%. On the flip side, NDMiner significantly improves GPM performance by 7.4×, on average.

5.8 Related Work

Table 5.7 provides a brief comparison of the most related works with NDMiner. A more detailed comparison follows.

GPM software systems. Numerous software frameworks efficiently utilize GPM algorithms on CPUs and GPUs. Early GPM systems [42, 256] rely on enumerating all possible embeddings, and then ruling out redundant embeddings using isomorphism tests. Recent works [104, 105, 159, 160, 232] avoid the expensive filter operations and prune out redundant embeddings during the search tree expansion. Other works strive to reduce the memory consumption of intermediate embeddings either by relying on SSD [265] or leveraging algorithmic techniques [60]. In addition to optimized software implementations, this work shows that GPM performance can be further improved using hardware-based techniques.

NDP architectures. To alleviate the cost of data transfer over bandwidth-limited and energy-hungry CPU-memory bus, several NDP architectures are proposed. Of these works, OMEGA [3] and PHI [174] augment the CPU memory with low-cost compute units for graph processing. Other works [7, 27, 55, 288, 299] offload graph computations to the logic layer of HMC. These proposals, however, are suitable for graph processing and cannot be directly applied for GPM acceleration because of its unique workload characteristics. Outside the context of graph processing, several other NDP architectures are proposed [45, 113, 133, 147, 196, 225, 228, 248–250]. For GPM, SISA [29] proposes to offload computation on existing PIM architectures by proposing set-centric ISA and fast set intersection algorithm. NDMiner improves SISA using domain-specific optimizations (hardware load elision, compiler optimizations, and set operation reordering).

Domain-specific accelerators. ExTensor [91] employs fast intersection circuits for tensor algebra that cannot be used for GPM out-of-the-box as it doesn't support key operations like

	Symm. Break.	NDP	Load Elision	Loop Fusion	Op Reorder
GraphZero [159]	✓	✗	✗	✗	✗
GraphPi [232]	✓	✗	✗	✗	✗
Gramer [280]	✗	✗	✗	✗	✗
FlexMiner [40]	✓	✗	✗	✗	✗
SISA [29]	✗	✓	✗	✗	✗
IntersectX [210]	✓	✗	✗	✗	✗
NDMiner	✓	✓	✓	✓	✓

Table 5.7: Comparison of NDMiner with related works.

pattern enumeration. Numerous graph processing accelerators aim at improving the irregular memory accesses via memory system optimizations [11, 24, 87, 172, 173, 186, 236, 252, 278]. As detailed in [280], graph processing and GPM workloads have distinct memory access patterns. Therefore, the effectiveness of graph processing accelerators might be limited when applied to GPM. Few recent works [40, 111, 210, 280] design specialized architectures for GPM. Out of these, FlexMiner [40] improves the performance and generality of prior accelerators [111, 280] by proposing a pattern-aware GPM accelerator. NDMiner outperforms FlexMiner by employing a domain-specific NDP architecture that includes novels optimizations like loop fusion and set operation reordering. IntersectX [210] optimizes GPM execution on a CPU by extending the ISA and architecture support. This approach, however, suffers from high on-chip storage requirement and unnecessary off-chip data transfers from DRAM. NDMiner offloads compute to low-cost NDP units augmented with domain-specific optimizations.

5.9 Chapter Conclusion

Irregular memory and complex data-dependent control flow instructions used in set operations dominate the execution time of GPM workloads. This work presented NDMiner—a domain-specialized NDP architecture to accelerate GPM. NDMiner offloaded the costly set computations to NDP. NDMiner further improved performance by uncovering and applying domain-specific optimizations. NDMiner integrated a near-data load elision unit that broke symmetry in hardware and terminated unnecessary loads. NDMiner employed compiler optimization and hardware mapping techniques that improved the algorithmic efficiency of sparse GPM workloads. NDMiner proposed

a graph remapping scheme and set operation reordering hardware to optimize the bank, rank, and channel-level parallelism in DRAM. Using dense, sparse, and mixed pattern mining algorithms, we showed that NDMiner significantly outperforms the state-of-the-art software (GraphPi) and hardware (FlexMiner) baselines by 6.4× and 2.5×, on average, at a negligible cost.

CHAPTER 6

Accelerating Temporal Motif Mining

This is a collaborative work with H. Ye, S. Vedula, K-Y Chen, Y. Chen, D. Liu, Y. Yuan D. Blaauw, A. Bronstein, T. Mudge, and R. Dreslinski.

Graphs (or networks) provide a general and useful abstraction for modeling complex phenomena, *e.g.,* social and communication networks in computational social science, protein-protein interaction networks in biology, and transaction networks in finance [22, 142]. Small subgraph patterns, referred to as *motifs*, play a crucial role in understanding the structure and function of complex systems encoded as a graph [16, 178, 230]. Mining motifs is one of the central problems in network science [168].

Most real world phenomena are not static. Static graphs aggregate the interactions that occur over networks by omitting the temporal information. While analyzing static graphs is useful, doing so completely disregards the dynamics occurring over the graph. For example, in the case of an email exchange network, a static graph renders two users "connected" irrespective of the number of emails exchanged between them. This leads to severe information loss. Temporal graphs, on the other hand, retain this information by maintaining a list of all interactions and their respective timestamps. Therefore, temporal graphs capture richer information compared to static networks [123, 190].

Temporal motifs are one of the fundamental building blocks of temporal networks, analogous to how static motifs are for static networks. Temporal motifs have been shown to be useful in user behavior characterization on social and communication networks [123, 124, 128, 192], predicting peptide binding in structural biology [163], characterizing the structure and function

of biological networks in bioinformatics [99], monitoring energy disaggregation on electrical grids [229], detecting fraud in financial transaction networks [86], and detecting insider threats in an organization's network [79, 156]. Furthermore, local motif counts have been shown to resolve symmetries and improve expressive power of graph neural networks [31]. Similarly, local temporal motif counts can be used as a subroutine for calculating node features in temporal graph learning [214].

Despite such wide utility of temporal motif mining, existing software frameworks offer sub-optimal CPU performance. This is because of the high computational complexity and irregular memory accesses of this workload. Temporal motif mining adds a time dimension to an already computationally and memory intensive static graph mining problem [29, 39, 40, 56, 211, 253, 280]. Furthermore, accesses to the graph structure and temporal edges incur irregular memory accesses that negatively impact the memory system's performance. While several acceleration techniques have been designed to speed up static graph processing [3, 11, 24, 87, 172–174, 186, 208, 222, 236, 240, 252, 278], streaming graph processing [25, 209], and static graph mining [29, 39, 40, 211, 253, 280], no prior work targets temporal motif mining. Moreover, temporal motif mining exhibits unique workload characteristics compared to previously studied graph problems as it processes temporal properties along with structural constraints (the main focus of prior works).

In this work, we present Mint—a novel hardware accelerator architecture and accompanying programming model for efficiently mining temporal motifs. To best address the challenges of efficiently executing this irregular algorithm, the design goals of Mint are three-fold: (1) realize a high degree of parallelism, (2) achieve high hardware utilization, and (3) improve memory system performance. To this end, we propose a task–centric programming model that enables asynchronous execution. The task is defined as a basic unit of computation for mining temporal motifs, *e.g.,* searching for a new edge to match. The asynchronous nature of this model unlocks a high degree of parallelism. Additionally, decoupled execution allows better hardware utilization.

Mint further proposes a new hardware accelerator to best utilize the proposed programming model. The hardware architecture is motif-agnostic, and can be programmed to mine any arbitrary

motif. The key features of the proposed hardware design include a hardware 1) *task queue* that dispatches tasks to compute units, 2) on-chip *context memory* that stores the key task context information to identify and advance the progress of an in-flight task, and 3) unique distribution of work to different compute units that perform on-chip context updates and off-chip graph traversal to find a new edge mapping. To further enhance the performance of this architecture, we make a key observation that the amount of node neighborhood data used by the algorithm reduces with respect to time. Based on this observation, we propose a novel optimization of memoizing the search index that significantly reduces the memory traffic of Mint.

We comprehensively evaluate the performance of Mint using detailed RTL models of proposed hardware and a C++ based cycle-accurate simulator. We compare the performance of Mint with state-of-the-art software frameworks running on a high-end server-grade CPU and a GPU. Mint outperforms the CPU implementations of Mackey *et al.* [156] and Paranjape *et al.* [192] by 363.1× and 2575.9× respectively, a GPU version of Mackey *et al.* by 9.2×, and PRESTO [219] by 16.2×, on average. Similar to Mackey *et al.* [156] and Paranjape *et al.* [192], Mint runs an exact mining algorithm, whereas PRESTO is an *approximate* mining algorithm. Using 28 nm commercial technology library, we implement Mint to find that it consumes just 28.3 mm^2 silicon area and 5.1 W.

Mint is the **first work** that designs a domain-specific accelerator for mining temporal motifs. The key contributions of this work are as follows:

- *Task-centric programming model* that allows for massive parallelism opportunities.
- *Hardware accelerator* architecture that uses its data path and memory design to cater to the unique properties of temporal motif mining.
- *Search index memoization* optimization that significantly reduces memory traffic.
- *Mint*—the first end-to-end system design for accelerating temporal motif mining that significantly outperforms existing software baselines running on a CPU by one–to–three orders of magnitude.

6.1 Background

6.1.1 Real-world Applications

Temporal graphs capture a rich set of information compared to static graphs by storing dynamic interactions in addition to the graph structure. Mining temporal motifs has shown to be effective across several application domains including social and communication networks [123,124,128,192], structural biology [163], bioinformatics [99], and finance [86].

In [124], the authors use temporal motifs as a tool to understand and quantify how information flows over a social network. Crucially, they demonstrate that it *cannot* be captured using its static counterparts. In financial transaction networks, certain types of temporal motifs can reveal artificial attempts to create high transaction volumes – an indication of potential financial fraud [86]. Features built with temporal motif distributions are shown to outperform their static counterparts in machine-learning based network classification [259]. In [156], authors show how temporal motif mining can be used to detect insider threats in an organization. *In summary, temporal motifs are one of the most fundamental properties (e.g., degree and centrality) computed over temporal graphs. As temporal graphs become ubiquitous, it becomes increasingly important to mine temporal motifs efficiently.*

6.1.2 Algorithmic Prior Work

Several algorithms have been proposed to mine motifs in temporal graphs. These algorithms can be broadly classified into two categories: (a) exact algorithms [125, 156, 192], and (b) approximate algorithms [152, 219, 264]. While exact algorithms aim to mine the precise temporal motifs in an entire input graph, approximate algorithms sample a subset of an input graph to estimate the number of matches in the entire graph. By limiting the amount of work, approximate algorithms achieve better scalability by reducing both computational and memory complexities. However, they suffer from inaccuracies in motif counts as they do not enumerate all motifs. In many scenarios, exact algorithms are still desired. For example, (1) in financial transaction networks, identification of *cycles*, a specific class of temporal motifs, indicates potentially fraudulent activity [86]; (2) in

organization networks, certain temporal motifs can characterize insider threats [79, 156]. In such high-risk scenarios, it is crucial to employ exact mining algorithms to enumerate *all* instances of the desired motifs, instead of approximate mining. Furthermore, approximate algorithms use exact algorithms as subroutines to process a subset of nodes/edges [152, 219]. Therefore, Mint focuses on speeding up exact temporal motif mining; it is also *directly applicable* to accelerate approximate mining algorithms.

Exact algorithms can be further divided into two sub-categories: (1) pattern-specific algorithms [125, 192], and (2) generic pattern-agnostic algorithms [156, 192]. While pattern-specific algorithms achieve better efficiency by using computation catered to a specific temporal motif, their applicability is limited. Furthermore, unlike static graph mining (*e.g.,* GraphPi [232], AutoMine [160], GraphZero [159]), no automatic framework exists for temporal motif mining that can discover optimized algorithmic schedules for arbitrary motifs. This requires designing hand-optimized algorithms for every new motif, which is error-prone and requires non-trivial programmer effort. Pattern-agnostic exact algorithms, on the other hand, can be used to mine any arbitrary temporal motifs. *In this work, we focus on optimizing a state-of-the-art pattern-agnostic exact temporal motif mining algorithm proposed by Mackey et al. [156] that outperforms prior algorithmic works.*

6.2 Why Design A New Accelerator?

This section argues the need of designing a new accelerator for accelerating the temporal motif mining problem.

6.2.1 Essence of Optimizing This Workload

Wide applicability. Static graphs do not capture rich dynamics that occur over networks [123]. Temporal networks are ubiquitous in domains ranging from communications, biological sciences, and finance. Temporal motifs are *fundamental* building blocks that constitute a temporal network [123]. Therefore, counting and mining temporal motifs is one of the primary tasks in temporal network

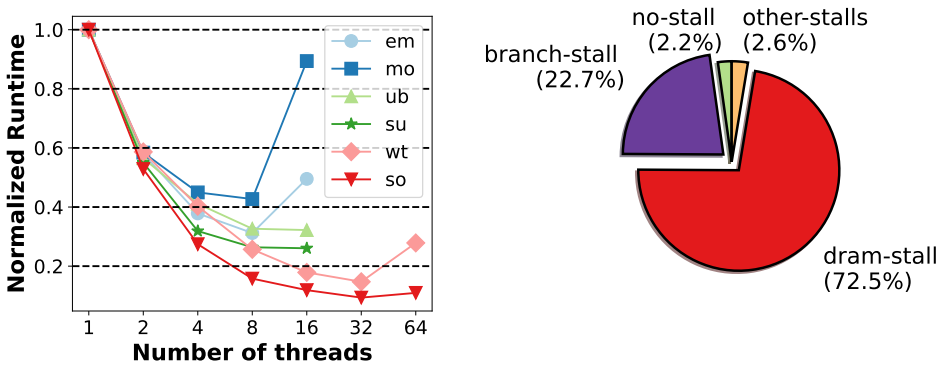


Figure 6.1: Performance scaling of M1 mining on different datasets (left), and CPU stall distribution (right) for mining M1 on a representative wiki-talk dataset.

analysis [92]. As we discussed in §6.1.1, temporal motif mining is widely applicable across several critical application domains. These include user characterization in social and communication networks [123, 124, 128, 192], understanding and explaining biological phenomena [163], monitoring electrical power grids [125], and machine learning on temporal graphs [31, 214, 259]. With the explosion of online content and digital footprint generated by social networks, temporal graphs are getting larger and richer by the day. *Therefore, designing optimization techniques for such widely applicable primitive operations is highly desirable.*

Algorithmic complexity. Let $|E_G|$ and $|E_M|$ denote the number of edges in the graph and the motif, respectively, and k denote the expected number of edges occurring in G within the duration δ. The worst-case algorithmic complexity of Algorithm 2 is $\mathcal{O}\left(|E_G| \cdot k^{|E_M|-1}\right)$: it scales linearly with $|E_G|$, polynomially with k, and *exponentially* with $|E_M|$. Intuitively, (i) increasing the temporal limit δ of a motif increases the *width* (*i.e.,* the number of nodes to visit in E_G for each edge in E_M) of the DFS search tree; and (ii) increasing motif size $|E_M|$ increases the *depth* of the DFS search tree. Therefore, the resulting complexity increases. For example, mining a 5-edge motif with temporal limit δ of 1 hour on the stackoverflow dataset implies $|E_G| = 32M, |E_M| = 5$, and $k_{avg} \approx 1500$, *leading to unreasonably high complexity*.

6.2.2 Workload Characterization and Optimization Opportunities

To understand the limitations of running existing temporal motif mining software on a commercial CPU, Fig. 6.1 shows its performance scaling and stall distribution. The figure shows the runtime of mining M1 on different datasets, normalized to a single–thread performance. The figure shows that the performance scaling saturates beyond 8–32 threads. For small datasets, the performance degrades by adding more threads as threading overhead dominates the execution time.

To better understand this trend, Fig. 6.1 (right) shows the stall distribution for mining M1 on a representative wiki-talk dataset, based on the CPI stack methodology [71]. For this experiment, we use a 32-thread configuration with three levels of cache hierarchy, 2 MB LLC slice/core. This distribution shows that the CPU spends 72.5% and 22.7% of the execution time stalled on DRAM and branch mispredictions. The DRAM stall is due to two reasons: irregular memory accesses to access graph structure and neighborhood filter operations that waste memory bandwidth (shown in lines 31, 33, 35 in Algorithm 2). Furthermore, data-dependent control flow instructions in lines 30–36 and lines 13–20 cause frequent branch mispredictions. *This calls for developing new acceleration techniques that can alleviate the memory and control-flow bottlenecks of this workload.*

6.2.3 Unique Workload Characteristics

At an algorithmic level, most prior works optimizing graph algorithms [3, 7, 11, 29, 39, 40, 56, 87, 172, 186, 208, 211, 240, 253, 278, 280] work on static graphs, whereas temporal motif mining operates on a temporal graph that adds a time dimension to the problem. While streaming graph processing accelerators [25, 209] also operate on dynamic graphs, they mostly optimize traditional graph computations (*e.g.,* PageRank). The reason is that streaming graphs measure *properties of the accumulated static graph over time*; this is *different* from temporal graph analysis (our setting), where the goal is to analyze *temporal properties* of a graph [217]. The edges of a temporal graph carry time information; therefore, the edges are *ordered*. In fact, this notion of order is *central* to temporal graph analysis. In streaming graphs, on the other hand, although the edges arrive at different points in time, they are treated as "updates" performed to the underlying accumulated static

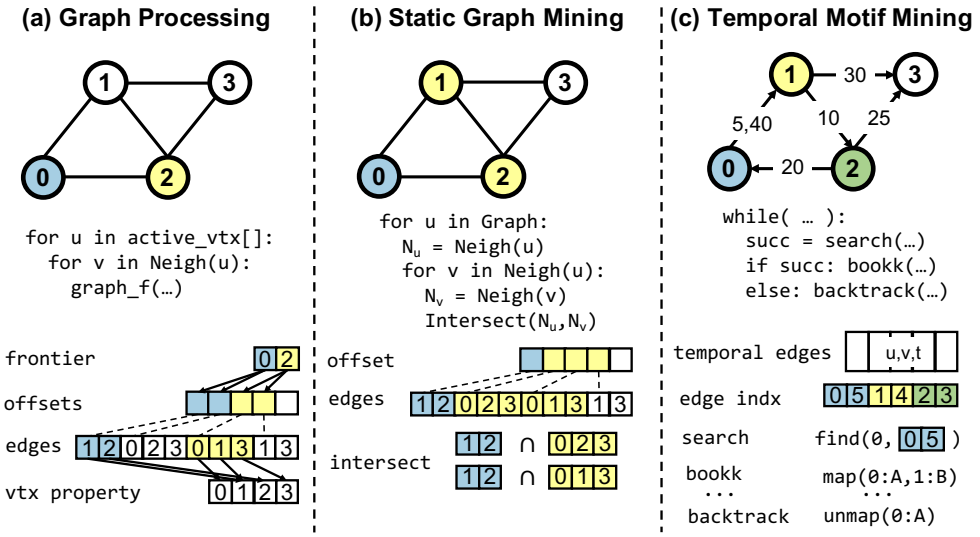


Figure 6.2: Unique workload characteristics in terms of data structures and algorithms employed in (a) graph processing, (b) static graph mining, and (c) temporal motif mining.

graph. Here, in stark contrast to temporal graph analysis, the graph properties remain unchanged even if the order of edges is permuted.

To motivate the necessity of designing a new accelerator for temporal motif mining, this subsection points out differences between its workload characteristics and other well-studied workloads, *i.e.,* graph processing and static graph mining. Fig. 6.2 shows the difference between generic pseudo-code, data structures, and computation patterns used in three algorithm categories. Graph processing algorithms (*e.g.,* PageRank and SSSP) typically pick a vertex from an active list, access its neighbors using offset and edge lists, and updates vertex properties as shown in Fig. 6.2(a). This incurs irregular data-dependent access into the offset, edge, and property lists. Graph mining algorithms, on the other hand, typically iterate over all vertices, find vertex neighborhoods, and compute set operations (*e.g.,* intersection) to mine subgraphs. The unique feature of this algorithm is the set operation computation, not present in graph processing algorithms. This has motivated the designs of novel architectures [29, 39, 40, 56, 211, 253, 280] for accelerating static graph mining.

As discussed in §2.6.2, temporal motif counting uses temporal edge list, instead of a static edge list. The key difference between these data structures is that the edges in a temporal edge list are sorted by their timestamps. Therefore, unlike the edge list used in static algorithms,

outgoing/incoming edges from the same node are not stored contiguously for temporal motif mining. Due to this design, the edge list for temporal algorithm stores edge indices instead of neighbor IDs. Furthermore, mining temporal motifs performs search, book-keeping, and back-tracking (Algorithm 2), where it spends a majority of execution time fetching neighborhood and searching for the first edge with a timestamp larger than the previously matched edge (lines 31, 33, 35 in Algorithm 2). Unlike static subgraph mining, temporal motif mining does not employ set operations as primitive computational blocks. Moreover, the amount of work performed by static/temporal motif algorithm is roughly proportional to the number of matched motifs, because each match requires a full expansion of the search tree. As shown by prior work [156], the ratio of number of matched static to temporal motifs vary by orders of magnitude. Depending on the input graph and motif, this ratio can be significantly higher or lower than 1. Therefore, the amount of work in static and temporal motif mining algorithms can be vastly different. *Due to the unique layout of data structures and computation patterns in temporal motif mining that lead to significantly different amounts of algorithmic work, it cannot be readily accelerated using prior techniques, which calls for designing a new accelerator for this problem.*

6.3 Task–Centric Programming Model

Motivated by the workload characteristics of temporal motif mining (§6.2.2), this section presents a novel task–centric programming model. The goals of this model are two-fold: (a) enable asynchronous execution to unlock massive parallelism and improve hardware utilization, and (b) reduce off-chip memory traffic.

6.3.1 Task: A Unit of Computation

A task is referred to as a single unit of computation used in temporal motif mining. Algorithm 2 performs three unique types of computations: 1) **search**: find the next edge to map, 2) **book-keeping**: update key metadata information when a valid edge is found, and 3) **backtrack**: void the

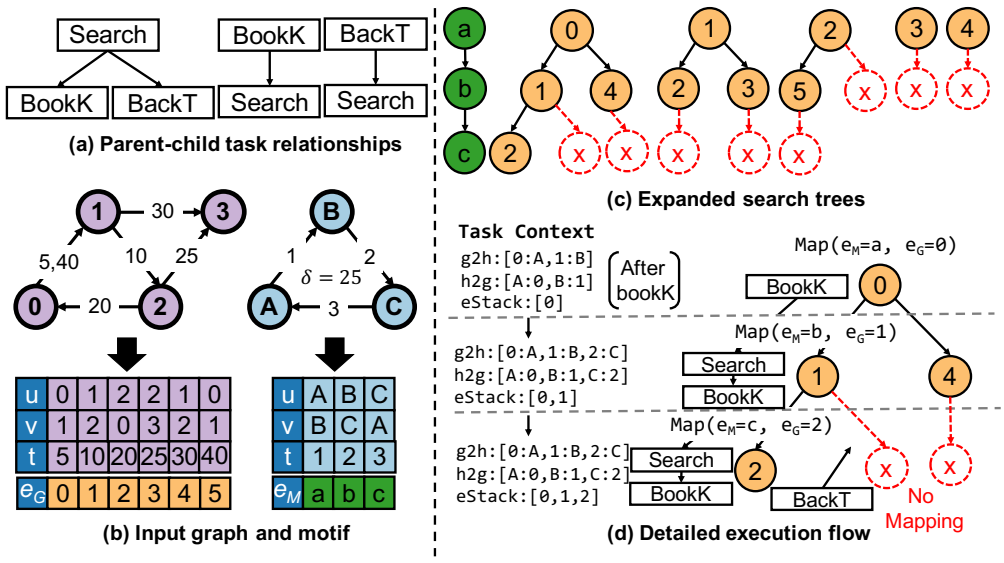


Figure 6.3: (a) Parent-child relationship between different task types, (b) an example input graph, temporal motif, and their temporal edge lists, (c) expanded search trees to mine a temporal motif, (e) a walk-through example of proposed programming model with task and metadata progression.

last mapped edge in metadata structures if no valid match is found. To capture this algorithmic behavior, we propose to represent these three functions as *tasks*.

Temporal motifs are mined using search tree expansion. Tasks that initiate the mining of a motif (that we call *root tasks*) are generated by matching the first motif edge with different temporal graph edges in chronological order. To expand the search trees further, subsequent tasks are generated by their parent tasks based on the outcome computation. For example, a search task would generate either a book-keeping or a backtracking task based on whether a matching edge is found or not. Fig. 6.3(a) shows the parent-child relationship between different task types. This allows a natural, incremental flow of task information from parents to children. A task terminates upon its completion. Upon termination, a child task is spawned if additional work is needed to traverse the search tree. The task generation stops once the whole search tree has been explored, and a new root task is generated.

6.3.2 Task Context

Each parent task communicates its progress to its child tasks to continue the mining process. We propose to capture this information in terms of a *task context*. A task context includes (a) task type, (b) last matched motif edge index (e_M), (c) last matched input edge index (e_G), (d) a mapping of graph nodes to motif nodes ($g2mMap[]$) and vice versa ($h2gMap[]$), (e) a stack of mapped temporal edge indices ($eStack$), and (f) initial timestamp ($firstEdgeTime$). A context stores minimal information required to traverse the search tree. *Therefore, a task context enables execution decoupling between the parent and child tasks.*

Memory requirement. The memory requirement of a task context is low. The task type, edge IDs, and temporal information are all integers, and can be stored with $\mathcal{O}(1)$ memory complexity. The node maps and the edge stack, on the other hand, grow linearly with the number of edges in input temporal motifs (*i.e.,* memory complexity $\mathcal{O}(|E_M|)$. As shown by prior algorithmic works [125, 152, 156, 192, 219, 264], a practical temporal motif size in real-world applications is up to eight edges. Using this conservative estimation, the memory requirement of a task context is 178 B. *This negligible memory requirement allows several task contexts to be stored on-chip and accessed at low latency in an accelerator.*

6.3.3 A Walk–Through Example

Fig. 6.3(b-d) demonstrate a walk-through example of the proposed programming model. Fig. 6.3(b) shows an example input graph, temporal motif, and their temporal edge lists. Fig. 6.3(c) shows the expanded search trees to mine the input motif. Note that each node of these trees maps one edge in the graph to a motif edge.

Fig. 6.3(d) expands the left-most search tree to explain how the programming model works. As discussed in §6.3.1, the root task automatically maps edges in the graph to the first motif edge in chronological order. Therefore, the first task performs book-keeping to map $e_M = a$ to $e_G = 0$. As shown in the simplified task context, graph nodes *0* and *1* are mapped to motif nodes *A* and *B* using $g2hNodes[]$ and $h2gNodes[]$. The matched graph edge $e_G = 0$ is pushed to the stack. Additionally,

```
1. class TaskContextType {
2.
3.   public:
4.     // define helper functions
5.
6.   private:
7.     bool _busy = false;
8.     TaskType _type;
9.     int _eG = -1, _eM = -1;
10.    MapType _h2gMap, _g2hMap;
11.    StackType _eStack;
12.    int _firstEdgeTime = -1;
13.
14.};
```

(a) Task context class

```
1. int TemporalMotifMining(…) {
2.   int num_matches = 0;
3.   TaskQuqueType t_queue;
4.
5.   while(true) {  // parallelize search trees
6.     if(t_queue.empty() && eG==n_graph_edges())
7.       break;
8.     TaskType task = t_queue.dequeue();
9.     task.process(&num_matches); // SR, BK, BT
10.    t_queue.enqueue(task.child_task());
11.       // SR->BK/BT, BK/BT->SR
12.  }
13.  return num_matches;
14.}
```

(b) Task-centric temporal motif mining

Figure 6.4: Task–centric temporal motif mining.

e_G, e_M, and $firstEdgeTime$ are also updated (not shown due to limited space).

This book-keeping task spawns a search task that finds the next edge to map to $e_M = b$. Using the graph structure and temporal information, this step finds $e_G = 1$ and spawns a book-keeping task to update the task context. This book-keeping task extends the context by mapping graph nodes *2* to motif node *C* as well as pushing $e_G = 1$ to the stack. This process continues until either a full motif is mined or if the search task cannot find any edge to map. Fig. 6.3(c) shows that there is no dependency between traversing different search trees, which results in traversing the different search trees in parallel *asynchronously*. Furthermore, in the proposed programming model, a task context is the only information necessary to advance the search. This naturally allows asynchronous, parallel task execution. *In sum, the proposed programming model with the right hardware design can achieve high throughput.*

6.3.4 Code Transformation

Fig. 6.4 shows the conversion of temporal motif mining code from an edge-centric to task–centric programming model. To achieve this, a programmer has to define the `TaskContextType` class (Fig. 6.4(a)). This includes memory allocation for context information, and helper functions to update the context. Additionally, the programmer needs to convert the main procedure used in the core algorithm. Lines 6.4–12 in Fig. 6.4(b) show these changes. The main data structure is the task queue. Because search trees can be traversed in parallel, several worker threads can dequeue

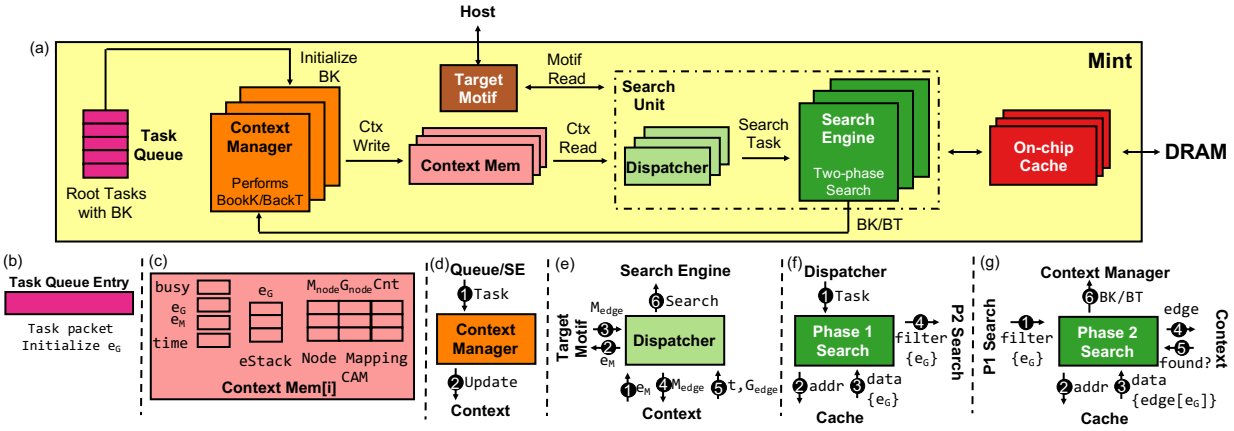


Figure 6.5: Hardware design overview of Mint: (a) overall architecture, (b) task queue entry, (b) an instance of context memory, and workflows of (d) context manager, (e) search unit dispatcher, (f) first phase of search engine, and (g) second phase of search engine. Parts (b-g) show Mint hardware components and their interactions with the rest of the system.

pending tasks from, and enqueue new tasks back to the queue for several tree expansions. The `task.process()` function executes one of three tasks: search, book-keeping, and backtracking. If a leaf node of the search tree successfully finds a match, `num_matches` is incremented. As presented in §6.3.3, traversing different search trees are independent of each other. Because this is a generic algorithm that can be used to mine any arbitrary temporal motif in any input temporal graph, the programmer effort modifying this code can be easily amortized over several executions.

6.4 Accelerator Architecture

To best utilize the proposed programming model, this section proposes a novel hardware accelerator architecture.

6.4.1 Design Overview

Fig. 6.5 shows the hardware accelerator design of Mint. It contains four memory structures, *i.e.*, task queue, target motif, context memory, and on-chip caches, and two computation blocks, *i.e.*, context manager and search unit. The task queue is a hardware FIFO queue that queues root tasks (§6.3.1). This root task is offloaded directly to the context manager with a book-keeping task to

initialize the search tree expansion. The target motif, programmed by the host CPU, stores the motif being mined, making Mint a *generic and motif-agnostic* temporal motif mining hardware design.

The context memory stores metadata information to keep track of task progress. This memory is updated by the context manager during the book-keeping/backtracking phase. The search unit, on the other hand, reads from this context memory to mine a graph edge. Finally, the on-chip caches are used to cache the graph structure and temporal edge list data to reduce the memory latency of the search unit. The on-chip context manager only updates the contexts of in-flight tasks, and does not communicate with DRAM.

As detailed in §6.3, a task can have three types: search, book-keeping, and backtracking. The context manager executes book-keeping and backtrack tasks. The search unit, on the other hand, is solely responsible for the search task. The search unit is further divided into blocks: the (1) dispatcher, and (2) search engine. The dispatcher reads an updated context, and dispatches work to the search engine. The search engine, in turn, consumes this task, and mines a graph edge to match a temporal motif edge. Upon completion of a search task, the search engine offloads either a book-keeping or backtracking task to the context manager for updating the task context, depending on the success or failure of the search.

The context manager only performs on-chip accesses to update context memory. These accesses take a single cycle. On the other hand, the search engine fetches data from DRAM, which takes multiple cycles. While some part of this latency is reduced by on-chip caches, multiple search units are necessary to exploit memory-level parallelism. Therefore, Mint employs several search engines that work on independent search tasks in parallel. To match the throughput of search engines, several context managers and memory instances are also used. While it is possible to use a fully asynchronous programming model, where any compute engine can pick up any pending task from the queue, this requires costly on-chip crossbars and routing logic, and a multi-ported task queue to enable an architecture with a large number of compute units. To simplify this microarchitectural design and routing logic, Mint's context manager, context memory, dispatcher, and search engine work in tandem to traverse a search tree. While this architecture also allows an asynchronous

task execution model, limiting the location of task offload greatly simplifies the design parameters and saves silicon area/power by avoiding costly routing logic. This, however, does not sacrifice performance because each context memory instance is busy when the assigned search engine mines an edge. This claim is further verified by the high bandwidth utilization of Mint (§6.7), showing that a fully flexible all–to–all connection between context managers and search engines is unnecessary.

6.4.2 Hardware Component Design Details

Target motif memory. This is a small register file that holds the target motif. For each temporal motif edge, it stores the source and destination IDs, and one delta time for an entire motif. Because the motif only has an edge ordering, a simple register file design is sufficient, where it is possible to use the chronological edge number e_M as an index. Prior works mine motifs with up to eight edges. Therefore, Mint supports temporal motifs of up to eight edges.

Task queue. The task queue is used to store and offload root book-keeping tasks. Fig. 6.5(b) shows the fields in each entry. Each queue entry stores a root task packet that contains a book-keeping task with additional information about mapping the first motif edge M_{edge} with different graph edges G_{edge} in chronological order as shown in §6.3.1. Therefore, each task queue entry stores the graph edge index e_G. Using e_G, Mint compute units can obtain source/destination graph nodes and edge timestamps from DRAM. Task queue initiates a search tree traversal by offloading a book-keeping task to the context manager. After this, a context manager works with a search unit to expand the rest of the search tree without communicating back to the queue.

Context memory. Fig.6.5(c) shows the context memory design, where each instance stores task context information. This includes a set of registers, a stack, and a Content Addressable Memory (CAM). Context registers store the task status (*i.e.,* busy or available), indices of the last mapped edge (e_M and e_G), and the timestamp of the last mapped graph edge. The stack $eStack$ stores indices of previously mapped edges. The stack is used by the context manager for backtracking. The CAM memory stores the node mapping information, which mimics $g2mMap[]$ and $h2gMap[]$ in hardware. The design decision of using a CAM is to quickly search which motif node is mapped

to which graph node, and vice versa. Additionally, the CAM also stores the number of times a graph node is mapped ($eCount$ in Algorithm 2).

On-chip cache. This is a standard multi-bank, multi-port set associative SRAM cache that reduces the latency of search by caching the graph structure and temporal information.

Context manager. The context manager updates the context memory while performing book-keeping or backtracking. As shown in Fig. 6.5(d), it accepts task packets either from the queue (only the root task) or the search engine (SE) (❶), and updates specific parts of context memory based on the task type (❷). For book-keeping, the manager expands a context for a newly matched edge that includes pushing an edge index to $eStack$, expanding the node mapping CAM, and incrementing their connection counts. It also updates e_G, e_M, and $time$ registers to reflect the state of the most recent search. For backtracking, a context manager pops an entry from the stack, voids node mappings, and updates edge index and time registers to invalidate the last edge mapping.

Search unit dispatcher. After the context memory is updated, the dispatcher reads this context and offloads a search task to the search engine. As shown in Fig. 6.5(e), the dispatcher first reads an updated motif edge index (e_M), and reads edge information from the target motif (❷-❸). Using this information, the dispatcher reads context memory (❹-❺) and finds the timestamp of the last mapped graph edge, and node IDs in the graph that are mapped to source and/or destination node IDs of the temporal edge that the search engine will mine next. Using this information, the dispatcher offloads a search task packet to a search engine (❻).

Search engine. A search engine performs a two-phase search in an attempt to match a motif edge to a graph edge. The first phase finds a set of graph edge indices that might map to a motif edge, and the second phase finds an exact edge. Search phase 1 (Fig. 6.5(f)) accepts a search task from the dispatcher (❶) that contains source/destination IDs of the motif edge being mined and previously mapped graph node IDs (if any). Using these graph node IDs, the search engine fetches its outgoing/incoming edge indices (❷-❸) (similar to lines 30–37 in Algorithm 2). Additionally, it filters edge indices to find all edges with timestamps after the last mapped graph edge. This is simply done by finding a subset of neighbor edge indices greater than e_G read from the context

memory. In contrast to software that employs binary search, Mint employs linear search as it is possible to efficiently perform this operation in hardware by streaming edge index cache lines using a series of comparators in parallel. These filtered edge indices are then processed in the second search phase (❹).

Fig. 6.5(g) shows the second search phase that finds an exact edge to map. Using filtered edge indices from phase 1 (❶), this phase first fetches temporal edges from memory (❷–❸). These edges are examined for both structural and temporal constraints by reading the context information (❹) to find the first valid edge that matches a motif edge. Resolving structural constraints includes ensuring that either the new graph nodes are not mapped earlier, or mapped to the same nodes in the motif that we are trying to match. Resolving temporal constraints includes checking edge timestamps against max edge time for the motif being mined, to verify the delta-time requirement. If either of these constraints are violated, a graph edge is discarded and the search engine examines the next graph edge. Based on whether a valid edge is found or not (❺), the search engine offloads either a book-keeping or a backtracking task back to the context manager (❻).

Crossbar. In this architecture, there is only one crossbar that resides between the task queue and all context managers. Because this is a one–to–all connection, there is no arbitration needed that further reduces area and power. Each search engine only serves its paired context memory because a context memory will not generate a new search task until the search engine returns the result of the previous search tree expansion. Therefore all connections within a set of context memory, search engine, dispatcher, and context manager are local, eliminating the need for convoluted NoCs or crossbars.

6.5 Design Optimizations

This section discusses a novel design optimization to reduce the memory requirement of phase 1 search. Additionally, we briefly discuss two standard optimizations that we tried that did not result in fruitful performance improvement.

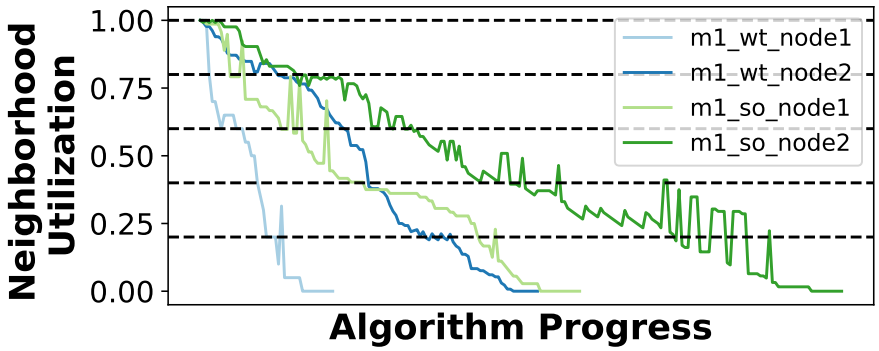


Figure 6.6: Reduction in the neighborhood data utilization for two sampled nodes while mining M1 on wiki-talk and stackoverflow. The x-axis represents the progress of an algorithm.

6.5.1 Search Index Memoization

The goal of this optimization is to reduce the memory traffic in search phase 1 by fetching mostly useful data. As detailed in §6.4.2, the first phase of search fetches outgoing/incoming neighbors of a node, and filters them based on the current e_G. Lines 31, 33, 35 shows this filter operation in Algorithm 2. To better understand the behavior of this operation, Fig. 6.6 shows the utilization of neighborhood data with respect to time. Intuitively, due to chronological order of mining edges, as the algorithm progresses, the resulting filtered edges have higher timestamps. Because node neighborhoods store edge indices in increasing time fashion, the neighborhood utilization decreases with respect to time. Notably, this is not a problem in the software implementation [156] as it employs binary search. *This results in wasted DRAM bandwidth and on-chip cache resources.*

To prevent futile data fetches, we propose a novel optimization to memoize the search result. For each node, we memoize the resulting index of the last search. Because search is performed in a chronological order, it is guaranteed that the edges discarded in any search operation will never be used in its subsequent search operations. We use two data structures for memoizing the previous search result for incoming and outgoing neighborhoods. Because the amount of memoization memory grows linearly with the number of graph nodes, Mint stores these data structures in DRAM.

Fig. 6.7 presents this optimization with an example. Suppose that an outgoing neighborhood of node 3 is accessed to expand the blue tree nodes ($e_G = 15, 18$). The outgoing neighborhood of node

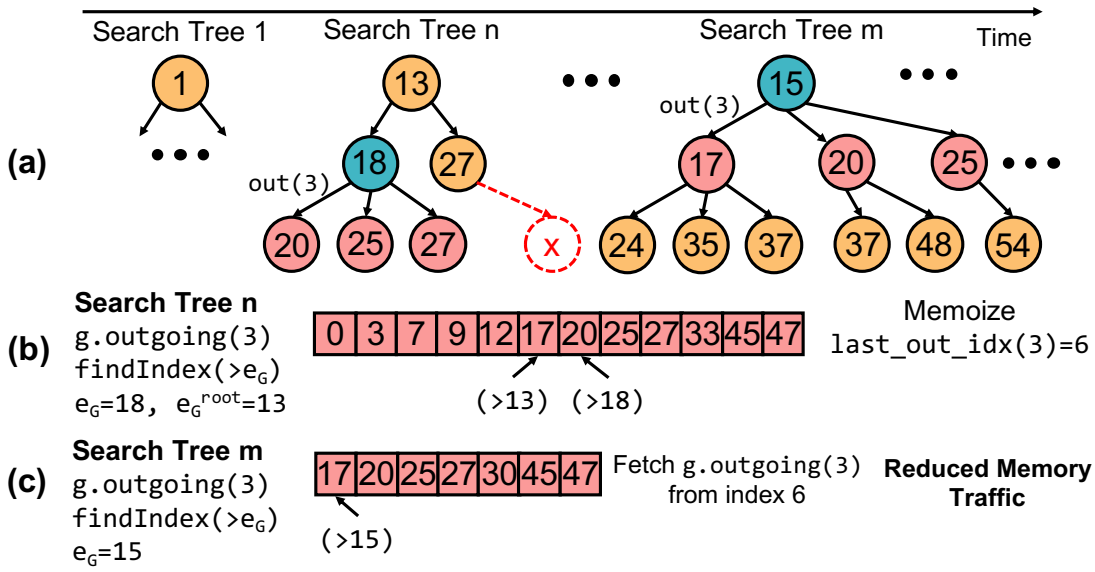


Figure 6.7: Design optimization of search index memoization to reduce memory traffic. (a) Progression of the algorithm with respect to time, (b) expanding the blue node in tree *n* by searching current $e_G = 18$ and $e_G^{root} = 13$, and (c) reduced search operation computation while expanding the blue node in three *m* due to memoization.

3 contains 12 edges with indices $0 - 47$ (shown in pink array). The first time that the search tree labeled *n* accesses outgoing neighbors of node 3, it fetches the entire neighborhood and searches for data elements greater than $e_G = 18$. The proposed optimization remembers the index of the first edge that occurs after the $e_G = 13$ of the root node ($last\_edge\_idx(3) = 6$). The next time that search tree *m* accesses outgoing neighbors of node 3, it only fetches all neighbors after index 6. This results in saving 5 unnecessary data fetches, and overall reduction in memory traffic.

The reason behind using an e_G of the root node for memoization is that all edges searched in any tree is guaranteed to have higher timestamps compared to the root nodes' edges from previous trees. However, as we expand search trees, there is no relation between the edge timestamps of non-root nodes in different trees. For example, Fig. 6.7 shows that outgoing neighborhood of node 3 is accessed by $e_G = 18$ in the earlier tree and $e_G = 15$ in the latter tree. Therefore, memoizing the index of edges greater than $e_G = 18$ for search tree *n* would result in incorrect result as it would miss an edge index 17 while expanding the search tree *m*.

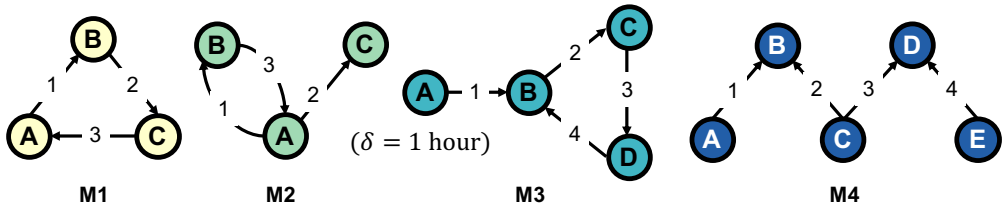


Figure 6.8: Temporal motifs used for evaluation.

6.5.2 What Didn't Work?

In addition to this novel optimization, we try two other standard optimizations employed by prior graph accelerators: (a) task coalescing, and (b) prefetching. These optimizations, however, did not yield reasonable performance improvements. First, task coalescing attempts to reduce the amount of memory traffic by coalescing phase 1 search tasks that access the same node neighborhoods. While in theory it reduces the number of memory accesses, its performance is very close to a non-task coalescing baseline because the cache lines only need to access DRAM once, and subsequent tasks can access this data from cache. Second, we attempted neighborhood prefetching for phase 1 and phase 2 of search. However, a detailed microarchitectural analysis of Mint shows that search engines are waiting for DRAM accesses for more than 98% of time and utilize more than 60% of peak DRAM bandwidth. Adding prefetching hurts performance because of high bandwidth demand and cache pollution. Therefore, Mint does not implement these optimizations.

6.6 Evaluation Methodology

6.6.1 Algorithms and Datasets

Algorithms. As discussed in §6.1.2, we use a generic, exact temporal motif mining algorithm for our study. Similar to prior works [156, 192], we mine four unique motifs (**M1-M4**) from three to five nodes in size (Fig. 6.8) with δ=1 hour. Due to long simulation times and limited space, we limit our evaluation to these motifs. However, Mint is a generic accelerator, and can be used out-of-the-box to mine any motif.

Graph	#Vertices	#Temporal Edges	Size (MB)	Time span (days)
email-eu (em)	986	332.3k	7.6	808
mathoverflow (mo)	24.8k	506.5k	12.0	2350
ask-ubuntu (ub)	159.3k	964.4k	24.5	2613
superuser (su)	194.1k	1.4M	36.0	2773
wiki-talk (wt)	1.1M	7.8M	196.7	2320
stackoverflow (so)	2.6M	36.2M	1493.0	2774

Table 6.1: Temporal graph data sets used for evaluation.

Datasets. Similar to prior works [156, 192], we use six real-world temporal graph datasets for evaluation as shown in Table 6.1 selected from SNAP [139]. These datasets are diverse in terms of their sizes from small (email-eu) to large (stackoverflow), and connectivity. email-eu is an email-exchange network between users at a large European research institute. mathoverflow, ask-ubuntu, superuser, and stackoverflow are interaction networks between users on Math Overflow, Ask Ubuntu, Super User, and Stack Overflow, in terms of comments, questions, and answers. wiki-talk is a user-editing network of pages on Wikipedia.

6.6.2 Baseline Hardware Platforms

To run the software baselines, we use a dual-socket server with two AMD EPYC 7742 processors, each with 64 physical cores (128 SMT threads). The aggregate Last Level Cache (LLC) size on each CPU is 256MB. The main memory in the system is 8-channel DDR4-3200 with a 1.5TB capacity. As shown in §6.2.2, the performance of temporal motif mining does not scale linearly with the number of threads. For each workload, we sweep the number of threads from 1 to 256, and choose the best-performing configuration for comparison. In addition to the CPU baseline, we compare the performance of Mint with an NVIDIA GeForce RTX 2080 Ti GPU.

6.6.3 Simulation Infrastructure

Table 6.2 shows the system configuration of Mint. It employs one task queue, and 512 context managers, search engines and context memory instances as detailed in §6.4.1. We use a 64-bank 64 KB on-chip SRAM cache (4 MB total), and 8-channel DDR4-3200 DRAM (same as CPU baseline).

To accurately estimate the performance of Mint, we implement a detailed two-phase simulation

Component	Modeled Parameters
Context Manager	512× context manager instances that updates context memory
Search Unit	512× dispatchers, 512× two-phase search engines
Task Queue	1× queue, 16-entry, 4 B memory per entry, 1 cycle task dequeue latency
Context Memory	512× context instances, each instance has metadata registers, edge stack, and node connectivity CAM, 2 cycle access latency
On-chip Cache	64× cache banks of 64 KB SRAM cache (4 MB total), 4-way set associative, 2 cache ports per bank, 64 B block size, 32 MSHR per bank, 2 cycle access latency
DRAM	8-channel DDR4-3200, 204.8 GB/s peak bandwidth

Table 6.2: Mint system configuration.

methodology. First, we model all hardware components (except caches) using System Verilog HDL. We synthesize this design using a commercial 28 nm technology library using the Synopsys Design Compiler. We use Synopsys PrimeTime for vector-based power estimation. Using detailed post-synthesis RTL simulations, we extract the critical path delay of our circuits and set Mint clock frequency at 1.6 GHz. Additionally, we collect the power and area numbers using RTL. We use CACTI [175] to estimate the performance/power/area of SRAM–based caches.

Second, to estimate end-to-end performance, we implement a cycle-accurate C++ simulator. This simulator faithfully models all system components, their RTL–based latencies, and their interactions. Several microarchitectural events are modeled in detail, including task queue dequeue, and stalls due to cache port contention, structural hazards at search engine, Miss Status Handling Registers (MSHRs) of cache, and memory controller. To model DRAM, we use Ramulator [118]. We verify simulator functionality by matching its compute and memory traces with an instrumented software baseline ensuring no events are missed.

6.6.4 State-of-the-art Baselines

Mackey *et al.* [156] CPU. This is a state-of-the-art generic temporal motif mining algorithm uses a DFS-based search tree traversal. We convert their code into a task–centric multi-threaded implementation (similar to proposed programming model) using work stealing OpenMP threads.

Mackey *et al.* [156] CPU w/ Memoization. This baseline implements our proposed search index memoization optimization in software on a Mackey *et al.* CPU baseline. Memoized indices are stored in a dedicated array in main memory. Because the indices are memoized based on the e_G

of a root node (§6.5.1), two search operations are triggered—one to find the memoization index, and the other to find e_G of the current node. All search operations use binary search.

Paranjape *et al.* [192]. This is an exact mining algorithm that first mines static subgraphs, and then resolves temporal constraints using a dynamic programming problem.

PRESTO [219]. PRESTO proposes a scalable edge sampling technique for approximate mining. It uses Mackey *et al.* [156]'s algorithm to mine motifs on a subset of edges.

Mackey *et al.* [156] GPU. This is a CUDA implementation of a state-of-the-art generic temporal motif mining algorithm running on an NVIDIA GPU. This baseline uses an in-house implementation as no open-source implementation of Mackey *et al.*'s algorithm exists.

Static graph mining accelerator FlexMiner [40]. Although FlexMiner does not support temporal motif mining, we divide this workload into two phases similar to a baseline algorithm presented in Paranjape *et al.* [192]: (1) mine static subgraphs by ignoring temporal information, and (2) use results of the first phase to mine temporal motifs by resolving temporal constraints. We use a state-of-the-art graph mining framework GraphPi [232] to find the performance of phase 1 on a CPU baseline (§6.6.2). To find FlexMiner performance, we reduce the GraphPi execution time by the highest speedup reported in FlexMiner (*i.e.,* 40×). We compare this FlexMiner performance to Mint by conservatively ignoring the execution time of phase 2, which provides a performance upper bound for this baseline.

6.7 Results

6.7.1 Performance Analysis

Benefit of search index memoization. To find the performance benefit of search index memoization, Fig. 6.9 compares the performance of Mint with and without applying this optimization with Mackey *et al.* [156]. This figure shows that, on average, the proposed optimization improves the performance of Mint from 91.6× to 363.1×. On average, the proposed optimization improves the performance of Mint by 4×. The reason behind this performance improvement is the reduction in

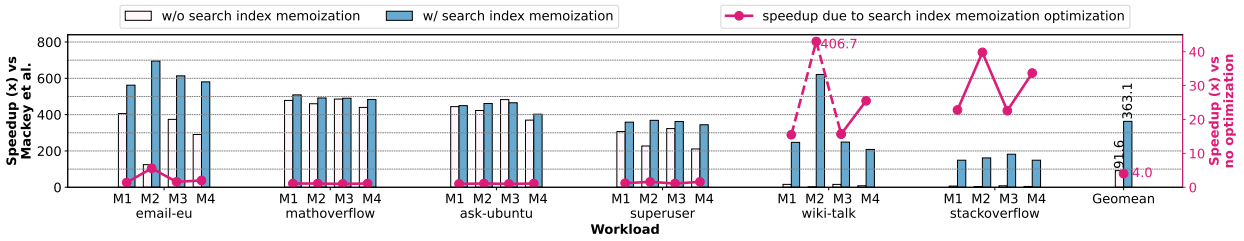


Figure 6.9: Performance improvement of Mint compared to Mackey *et al.* [156] and average memory bandwidth utilization, with and without the search index memoization optimization.

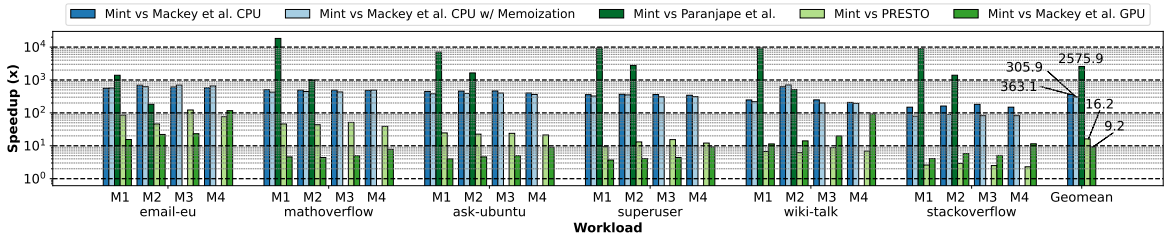


Figure 6.10: Performance improvement of Mint compared to Mackey *et al.* [156] CPU (without and with proposed optimization implemented in software), Paranjape *et al.* [192], PRESTO [219], and a GPU implementation of Mackey *et al*. The open-source codebase for Paranjape *et al.* only supports M1 and M2.

memory traffic. Our evaluation shows that this optimization reduces the memory traffic by 2.8×, on average (up to 30.6× for mining M2 on stackoverflow).

This effect is more prominent for large datasets (wiki-talk and stackoverflow) because they access large neighborhood sets, and filter operations lead to severe under-utilization of memory resources (Fig. 6.6). Our further investigation shows that the sizes of the largest 10% of vertex neighborhoods, which benefit the most from search index memoization, in wiki-talk dataset are 14.9×–38.6× larger than the four smaller datasets, on average. Similarly, the largest 10% of vertex neighborhoods in stackoverflow are 2.6×–6.7× larger than the four smaller datasets, on average. Therefore, search index memoization is the most effective in large datasets that significantly reduces futile neighborhood fetches, improving overall performance. Large vertex neighborhoods in wiki-talk and stackoverflow datasets further underscore the value of this optimization.

Comparison with state-of-the-art CPU baselines. Fig. 6.10 compares the performance of Mint with four state-of-the-art software frameworks running on CPU. Mint outperforms Mackey *et al.* [156] by 363.1×, on average. Note that both baseline and Mint use a task–centric programming

model. The high performance improvement of Mint over Mackey *et al.* is attributed to (a) converting task context updates to on-chip accesses, (b) domain-specific architecture design that efficiently executes the algorithm, and (d) search index memoization that significantly reduces memory traffic. The second bar (light blue color) shows the performance improvement of Mint over a software baseline that implements the search index memoization optimization. While search index memoization reduces the amount of work in the search phase, it comes at the cost of performing and additional search in software. As shown in Fig. 6.10, most of the performance benefit of proposed optimization in software is offset by the overhead of an additional search. The figure shows that Mint outperforms a CPU baseline that implements search index memoization by 305.9×, on average.

Mint also outperforms Paranjape *et al.* [192] by 2575.9×, on average. As shown in prior work [156], Paranjape *et al.* suffers redundant computation when the number of static subgraphs are higher than temporal motifs as it mines static subgraphs before resolving temporal constraints. Additionally, Mint benefits from an optimized programming model and domain-specific hardware design. The open-source implementation [191] of Paranjape *et al.* does not support M3 and M4; we limit our comparison to M1 and M2. PRESTO [219] is an approximate algorithm that samples temporal edges and runs exact mining algorithm as Mackey *et al.* on these edges. The goal of PRESTO is to achieve better scalability by mining motifs on a subset of edges. Fig. 6.10 shows that Mint, despite using an exact algorithm, outperforms PRESTO by 16.2×, on average. Because PRESTO is an approximate algorithm, its resulting motif counts are mostly within 10% error of the actual value, whereas Mint mines all motifs. Because PRESTO also employs the same algorithm for mining motifs on a subset of edges, Mint can also accelerate PRESTO. *This results shows the value of hardware acceleration that can achieve both better quality results (by running an exact algorithm) and superior performance by designing its data path and memory subsystem to cater to an application's unique workload characteristics.*

Comparison with a GPU baseline. Fig. 6.10 shows that Mint significantly outperforms a GPU implementation of Mackey *et al.* [156] algorithm by 9.2×, on average. As discussed in

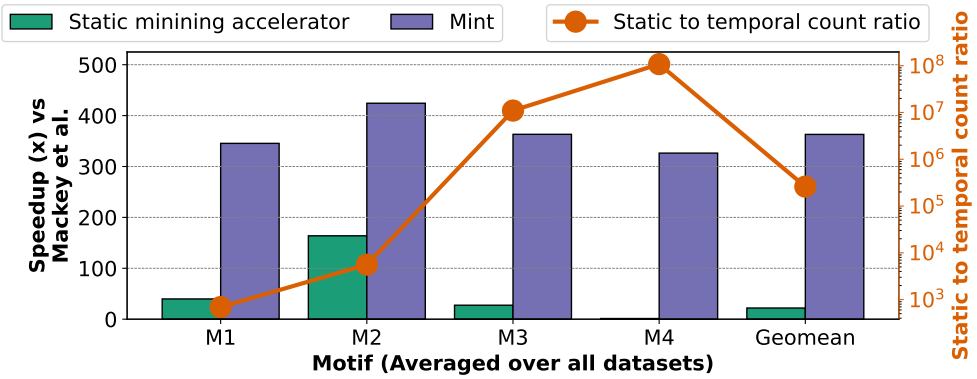


Figure 6.11: Performance of a static mining accelerator FlexMiner [40] and Mint compared to Mackey *et al.* The secondary y-axis shows the ratio between static to temporal motif counts. Results averaged over all datasets.

§6.2.2, the temporal motif mining workload is bound by irregular memory accesses and control-flow instructions. While GPU improves the performance of this workload over a CPU baseline by offering massive parallelism and exploiting higher memory bandwidth, the GPU performance is limited due to the highly irregular nature of this workload leading to frequent thread divergence and non-coalesced memory accesses. Mint, on the other hand, further improves the performance of this workload by optimizing its data path to address unique workload characteristics of temporal motif mining. Furthermore, the peak memory bandwidth of a GPU is more than 3× the peak bandwidth of Mint. Due to high memory bandwidth utilization of this workload (§6.7.2), Mint can offer even higher speedup than reported compared to a GPU in an iso-bandwidth comparison. Moreover, Mint operates at 50× lower power (§6.7.3) than GPU that uses 250 W.

Comparison with a static graph mining accelerator. We further compare the performance of Mint with a static mining accelerator FlexMiner [40]. Fig. 6.11 shows that even by ignoring the temporal constraint resolution process, Mint achieves an order of magnitude better performance, on average, compared to FlexMiner. The figure further shows that the number of mined static graphs are significantly higher than the temporal motifs, which results in much more work for the static mining accelerator to perform. Temporal motif mining effectively prunes invalid subgraphs that do not meet temporal constraints, leading to significantly less work. *This result underscores the value of designing a temporal motif mining accelerator despite the availability of static mining*

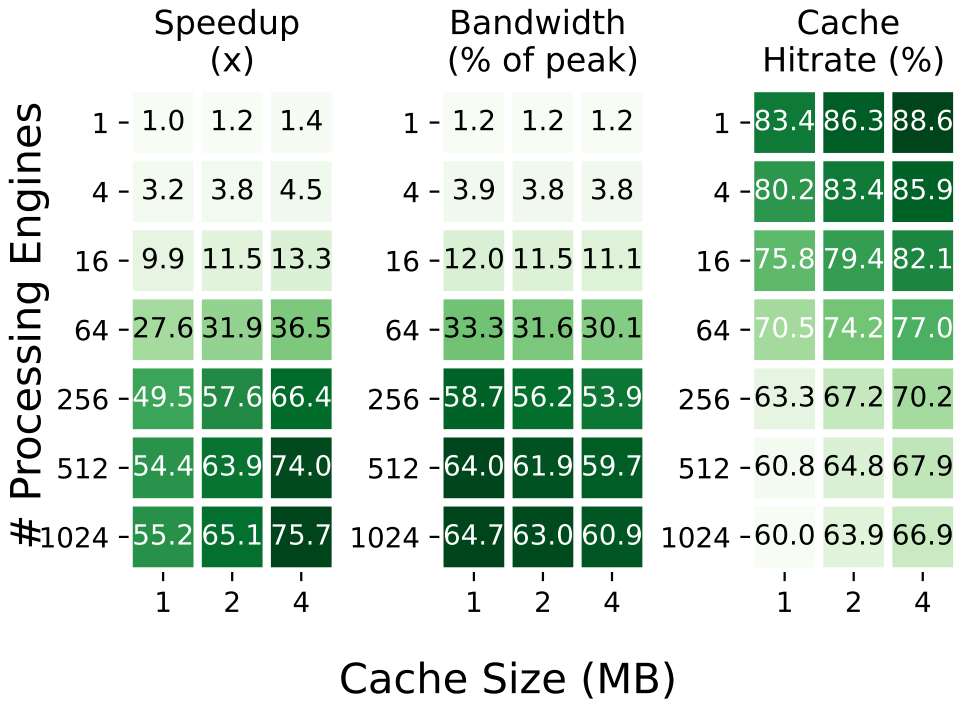


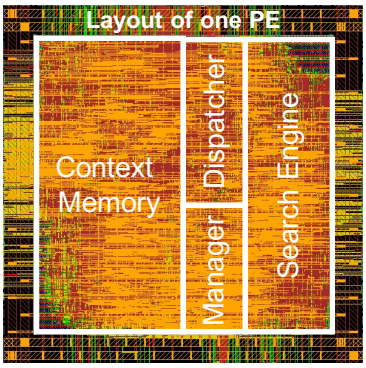
Figure 6.12: Sensitivity of performance (normalized to 1 processing engine 1 MB cache), bandwidth, and cache hit rate for mining M1 on a representative wiki-talk dataset.

accelerators [29, 39, 40, 56, 211, 253, 280].

6.7.2 Sensitivity Analysis

To demonstrate the performance sensitivity of Mint, and the benefit of employing different hardware components, Fig. 6.12 shows how the performance, average memory bandwidth utilization, and cache hit rate changes for varying number of processing engines/PEs (a PE constitutes a context manager, a context memory instance, a dispatcher, and a search engine) and cache sizes for a representative workload of M1 mining on wiki-talk. The performance is normalized to a baseline configuration of 1 PE and 1 MB cache. The performance of Mint scales with the increase in compute resources and cache size. Specifically, by increasing the number of PEs by 1024× and cache size by 4×, the performance scales by 75.7×.

Adding compute resources enables exploiting more parallelism, and a larger cache size reduces the memory latency of the search phase. Scaling compute and memory resources also scales



Component	mm² Area	mW Power
Target Motif (1x)	< 0.001	6.8
Task Queue (1x)	< 0.01	< 0.1
Context Mem (512x)	4.98	265.0
64 KB cache (64x)	19.29	4698.2
Context Manager (512x)	0.36	18.9
Dispatcher (512x)	0.53	17.4
Search Engines (512x)	3.12	67.1
Crossbar (1x)	0.05	0.3
Total	**28.3 mm²**	**5.1 W**

*Power is measured at 1.6 GHz

Figure 6.13: Layout of one processing engine (PE) and area/power measurements of an entire Mint design.

the memory bandwidth utilization. With fewer PEs, the workload is bound by the availability of compute resources that cannot saturate memory bandwidth. Adding compute resources shifts the workload from being compute bound to memory bound. Our evaluation shows that with 256 PEs, the workload slowly starts shifting from being compute to memory bound. With more PEs, Mint hardware expands multiple search trees in parallel, reducing the cache hit rate from 83.4% to 60%. However, increased memory and compute parallelism still improves overall performance of the workload. Additionally, with high concurrency, the workload starts experiencing cache port contention that constitutes 0.5% stall cycles for 1024 PEs, 4 MB cache.

6.7.3 Area and Power Analysis

Fig. 6.13 shows the layout of one PE, and the area and power consumption measurements of a full Mint design based on post-synthesis results on a 28 nm technology node. The power results includes both leakage and dynamic power consumption. The dynamic power is averaged over all workloads. The table shows that Mint consumes an overall area of 28.3 mm^2 and 5.1 W power. A majority of area and power is consumed in on-chip SRAM caches that reduce the memory latency of search engines. As shown in Fig. 6.12, caches significantly improve the performance of Mint, therefore, their high share of area and power is justified. The cache consumes approximately equal amounts of power in dynamic and leakage energies. This is because of the multi-banked cache

design, where all banks consume leakage power, whereas only one bank consumes dynamic power for each cache access. This multi-banked design, however, is desirable to reduce the performance hit due to cache contention.

6.8 Related Work

Mint is the first work that designs a novel accelerator architecture for mining temporal motifs. Below, we compare Mint with the closest related works.

Software frameworks for static graph mining. Several software frameworks implement efficient graph mining algorithms on CPUs and GPUs. Early works [256] enumerate all possible subgraphs and then rule out invalid embeddings using isomorphism tests. Recent works [42, 104, 105, 159, 160, 232] avoid the expensive isomorphism tests and prune out redundant subgraphs. Other works reduce the memory consumption of intermediate subgraphs either by relying on SSD [265] or leveraging algorithmic techniques [60]. Approximate algorithms [101, 204, 212] aim to achieve better scalability on large graphs by mining subgraphs on a subset of edges. These frameworks, however, work for mining subgraphs in static graph inputs, and do not support temporal motif mining.

Software frameworks for temporal motif mining. As discussed in §6.1.2, several software frameworks have been designed to optimize temporal motif mining. Among these, a few works [125, 156, 192] propose exact algorithms, while others [152, 219, 264] achieve better scalability by sampling and mining only a subset of edges. Mint further optimizes performance of these software frameworks by proposing a hardware accelerator. As shown in §6.7, Mint significantly outperforms state-of-the-art software baselines by 16–2576×.

Hardware acceleration for graph processing. Numerous acceleration techniques have been proposed to speed up graph processing on CPUs [11, 24, 174, 252], GPUs [222], and using dedicated accelerators [2, 3, 25, 87, 172, 173, 186, 208, 209, 236, 240, 278]. These works mostly focus on optimizing the irregular memory accesses of graph processing workloads. As discussed in §6.2.3,

the memory access and computation patterns of temporal motif mining are unique (*e.g.,* search phase is not present in traditional graph workloads). Furthermore, a few prior accelerators [2, 208, 209] employ asynchronous execution models. While these design philosophies seem similar to Mint on the surface, the domain-specific nature of accelerators result in fundamental design differences. For example, Mint uniquely employs (a) no task insertion back into the task queue, (b) unique workload division between compute units, (c) domain-specialized context memory design, and (d) lack of prefetching and task coalescing (§6.5.2) commonly employed in prior accelerators. Therefore, prior optimization techniques cannot be directly applied to accelerate temporal motif mining.

Hardware acceleration for static graph mining. Recent works propose hardware acceleration techniques for static graph mining, by either building domain-specific architectures [39, 40, 111, 211, 280] or by offloading the workload to near-data processing architectures [29, 56, 253]. While temporal motif mining is analogous to static subgraph mining, as shown in §6.2.3, their computation patterns are distinct. While acceleration techniques for static subgraph mining focus on optimizing set operations, temporal motif mining does not employ any set computation. Furthermore, Fig. 6.11 shows that using static mining accelerators to speed up temporal motif mining results in significantly more work and sub-optimal performance. Therefore, prior static graph mining accelerators cannot be directly used to support the intricate computation and memory access patterns of temporal motif mining.

Hardware acceleration for matrix operations. Matrix operations (both dense and sparse) have been heavily optimized using domain-specific accelerators [89, 189, 243, 294], GPUs [274, 283], FPGAs [149], and TPU [110]. While these techniques optimize matrix operations, as discussed in §2.6.2, temporal motif mining algorithms do not involve any matrix operations. In contrast, the studied workload employs unique operations on temporal graph data (*e.g.,* filtering temporal edge list, discovering new graph edges based on previously matched edges, and search backtracking) that cannot be expressed in terms of matrix operations efficiently. Therefore, prior matrix acceleration techniques cannot optimize the workload of temporal motif mining.

6.9 Chapter Conclusion

This work presented Mint—a novel programming model and hardware accelerator for mining temporal motifs. The programming model divided the workload execution down in terms of three fundamental tasks and proposed a task–centric asynchronous execution model that unlocked massive opportunities for parallelism. We then proposed a domain-specific architecture that optimized its data path and memory subsystem design to best accelerate temporal motif mining using the proposed programming model. The hardware accelerator is motif and dataset-agnostic, and can be programmed to wwmine any arbitrary temporal motif. To further improve performance, we proposed search index memoization that significantly reduced memory traffic. Our evaluation demonstrated that Mint significantly outperformed state-of-the-art software frameworks by 16–2576$\times$ by using 28.3 mm^2 area.

CHAPTER 7

Conclusion And Future Work

A graph is a fundamental data structure that effectively models entities and their interactions using nodes and edges. Graphs are used to model several networks including social media, cybersecurity, communication, citations, computational chemistry, high-energy physics, bio-informatics, and several more. Algorithms to analyze real-world graphs extract useful information from graph data, and enable many real-world applications. Speeding up these algorithms leads to significant impact on many application domains, such as, improving the quality of data analytics applications, product recommendations, and web services.

This dissertation presented a systematic hardware-software co-designed optimization study of a breadth of graph workloads. From an algorithmic perspective, I showcased traditional graph processing (*e.g.,* PageRank and SSSP), random walk based graph learning (*e.g.,* link prediction and node classification), graph pattern mining (*e.g.,* triangle counting), and temporal motif mining (*e.g.,* cycle detection) algorithms and their unique properties. From a graph dataset point of view, I studied both static and dynamic (specifically, temporal) graphs. On the software design front, I proposed novel compiler analysis techniques and programming models. From a performance analysis and hardware optimization standpoint, I showcased the designs of CPU and GPU micro-architectures, near-data processing system, and domain-specific hardware accelerator architecture.

More specifically, first, I presented Prodigy that improved the performance of traditional graph processing and other similar data-indirect irregular workloads. The key proposal of Prodigy was the Data Indirection Graph (DIG) that effectively captured an algorithm's data structure layout and

traversal pattern. The DIG was automatically constructed in software using compiler technology, and used in hardware for informed prefetching. Prodigy leads to more than 2.5× performance improvement of several important memory-bound workloads. Second, I presented high-performance CPU and GPU implementations of two important graph learning tasks on temporal graphs. My implementation used a scalable random walk-based algorithm to learn node embeddings. I presented an in-depth performance analysis of the graph learning tasks to find their execution bottlenecks, and proposed several optimization opportunities based on the performance analysis insights.

Third, I analyzed the performance of graph pattern mining workloads to find that costly set operations dominated their execution time. To accelerate this workload, I presented NDMiner—a near data processing architecture. In addition to reducing the CPU-memory data movement, NDMiner also incorporated several domain-specific optimizations to further improve the application performance. Compared to a state-of-the-art hardware accelerator, NDMiner improved the performance of graph pattern mining workloads by 2.5×. Fourth, I presented Mint—a novel programming model and hardware accelerator architecture for efficiently mining motifs (subgraphs) in temporal graphs. The proposed programming model enabled an asynchronous task execution and unlocked massive opportunities for parallelism. I then presented Mint hardware accelerator that incorporated a data path and memory subsystem to cater to the unique workload characteristics of temporal motif mining. Mint offers several orders of magnitude performance improvement over the state-of-the-art software running on commercial hardware platforms.

While this dissertation takes a major step forward in optimizing a variety of graph workloads, it certainly is not the end. Below, I detail several future research directions that I plan to work on next as a part of my long-term vision to improve graph workload performance.

Extending Prodigy's prefetching capabilities. Chapter 3 presented Prodigy, a hardware-software co-design technique to improve the performance of key data-indirect irregular workloads. More specifically, Prodigy can prefetch for two specific types of indirect memory access patterns defined as single-valued indirection and ranged indirection. While these types of indirect memory access patterns are prevalent across traditional graph processing, sparse matrix computation, and

scientific computing domains, there are a few other types of data-indirect memory accesses that Prodigy does not support. These include accesses for linked lists, various type of tree structures, and hash joins. As a direct extension to Prodigy, I will work on designing a generic system that can prefetch for workloads exhibiting a variety of data-indirect memory access patterns.

Optimizing streaming graph processing. In this dissertation, I presented two workload optimization studies on static graph datasets (Chapter 3 and Chapter 5), and a benchmarking and optimization study on temporal (a type of dynamic) graph datasets (Chapter 4 and Chapter 6). In addition to temporal graphs, there other ways of representing dynamic graphs. For example, streaming graphs accommodate updates to an underlying graph structures at different points in time by adding/removing nodes and edges. There are two phases involved in streaming graph processing, *i.e.,* graph update and graph query execution. As the names suggest, the graph update phase updates a graph structure by accounting for a batch of newly added/removed nodes/edges. Query execution, on the other hand, performs a graph query accounting for the latest graph mutations. In my future study, I will conduct a systematic optimization of streaming graph processing that includes both graph update and query execution.

Rethinking optimizations for distributed graph processing. This dissertation mainly focused on in-memory graphs, *i.e.,* graphs that fit within the main memory capacity of today's servers. While today's server memory can scale up to a Tera Byte (TB) in size, the sizes and complexities of real-world graphs are also steadily rising. Furthermore, several graphs store node and/or edge features to enrich the network information. These features contain several floating point numbers, creating a large-scale feature matrix. Therefore, such large-scale graph structures may require 100s of TBs to Petabyte (PB) in capacity that does not fit on one machine. As these large-scale datasets become available, I will work on optimizing a distributed model for graph processing, where the graph is partitioned across multiple compute nodes.

Enabling generic graph acceleration. While my work showed how to individually optimize different types of graph workloads, another key design challenge is to design a system for generic graph workload acceleration. I envision a system with several multi-core CPUs, GPUs, near-data

compute fabrics, and hardware accelerators that run in tandem to accelerate various types of graph workloads. To enable this heterogeneous compute environment, I will work on integrating various optimization techniques together, orchestrating unique computation types onto different hardware platforms, and managing the memory system issues.

BIBLIOGRAPHY

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[2] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient speculative parallelism for accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1247–1262, 2020.

[3] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 134–145, 2018.

[4] Charu Aggarwal and Karthik Subbian. Evolutionary network analysis: A survey. *ACM Computing Surveys (CSUR)*, 47(1):1–36, 2014.

[5] Charu C Aggarwal, Yuchen Zhao, and Philip S Yu. On clustering graph streams. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 478–489. SIAM, 2010.

[6] Nesreen K Ahmed, Ryan A Rossi, John Boaz Lee, Theodore L Willke, Rong Zhou, Xiangnan Kong, and Hoda Eldardiry. role2vec: Role-based network embeddings. *Proc. DLG KDD*, pages 1–7, 2019.

[7] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, pages 105–117, June 2015.

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

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

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

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

[12] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses. In *CGO*, pages 305–317, Feb 2017.

[13] Sam Ainsworth and Timothy M. Jones. Software prefetching for indirect memory accesses. In *CGO*, pages 305–317, Piscataway, NJ, USA, 2017.

[14] Antti Airola, Sampo Pyysalo, Jari Björne, Tapio Pahikkala, Filip Ginter, and Tapio Salakoski. All-paths graph kernel for protein-protein interaction extraction with evaluation of cross-corpus learning. *BMC bioinformatics*, 9(11):1–12, 2008.

[15] A. Akram and L. Sawalha. x86 computer architecture simulators: A comparative study. In *ICCD*, pages 638–645, Oct 2016.

[16] Uri Alon. Network motifs: theory and experimental approaches. *Nature Reviews Genetics*, 8(6):450–461, 2007.

[17] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. Data prefetching by dependence graph precomputation. In *ISCA*, pages 52–61, June 2001.

[18] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644, 2011.

[19] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, September 1991.

[20] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *HPCA*, pages 399–411, Feb 2019.

[21] 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*, pages 203–214, 2018.

[22] Albert-László Barabási. Network science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1987):20120375, 2013.

[23] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–23. IEEE, 2021.

[24] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–386. IEEE, 2019.

[25] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R. Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. Improving streaming graph processing performance using input knowledge. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1036–1050, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. In *arXiv:1508.03619 [cs.DC]*, 2015.

[27] Leul Wuletaw Belayneh and V. Bertacco. Graphvine: Exploiting multicast for scalable graph analytics. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 762–767, 2020.

[28] Rahul Bera, Anant V Nori, Onur Mutlu, and Sreenivas Subramoney. Dspatch: Dual spatial pattern prefetcher. In *MICRO*, pages 531–544, New York, NY, USA, 2019.

[29] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 282–297, 2021.

[30] Peter L Bird, Alasdair Rawsthorne, and Nigel P Topham. The effectiveness of decoupling. In *ICS*, pages 47–56, New York, NY, USA, 1993.

[31] Giorgos Bouritsas, Fabrizio Frasca, Stefanos P Zafeiriou, and Michael Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[32] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Compiling graph applications for gpu s with graphit. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 248–261. IEEE, 2021.

[33] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151. IEEE, 2012.

[34] David Callahan et al. Software prefetching. *SIGPLAN Not.*, 26(4):40–52, April 1991.

[35] Shaosheng Cao et al. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[36] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC*, pages 1–12, Nov 2011.

[37] Trevor E Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, August 2014.

[38] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[39] Qihang Chen, Boyu Tian, and Mingyu Gao. *FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators*, page 43–55. Association for Computing Machinery, New York, NY, USA, 2022.

[40] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and Arvind. Flexminer: A pattern-aware accelerator for graph pattern mining. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pages 581–594, 2021.

[41] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: A two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 378–391, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205, April 2020.

[43] Xuhao Chen and Tianhao Huang. GraphMinerBench open-source implementations. https://github.com/chenxuhao/GraphMiner.

[44] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. DaDianNao: A Machine-Learning Supercomputer. In *MICRO*, pages 609–622, 2014.

[45] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

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

[47] Young-Rae Cho and Aidong Zhang. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Transactions on Information Technology in Biomedicine*, 14(1):30–36, 2010.

[48] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. Comput. Syst.*, 22(2):214–280, May 2004.

[49] Nicholas Choma, Federico Monti, Lisa Gerhardt, Tomasz Palczewski, Zahra Ronaghi, Prabhat Prabhat, Wahid Bhimji, Michael M Bronstein, Spencer R Klein, and Joan Bruna. Graph neural networks for icecube signal classification. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 386–391. IEEE, 2018.

[50] W.W. Cohen. Enron email dataset. http://www.cs.cmu.edu/~enron/ .Accessedin2009.

[51] Jamison D Collins, Dean M Tullsen, Hong Wang, and John Paul Shen. Dynamic speculative precomputation. In *MICRO*, pages 306–317, Dec 2001.

[52] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA*, pages 14–25, June 2001.

[53] Gene Ontology Consortium. The gene ontology resource: 20 years and still going strong. *Nucleic acids research*, 47(D1):D330–D338, 2019.

[54] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *SIGARCH Computer Architecture News*, 30(5):279–290, October 2002.

[55] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2019.

[56] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. Dimmining: pruning-efficient and parallel graph mining on near-memory-computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 130–145, 2022.

[57] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[58] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[59] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA*, page 293–304, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1357–1374, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[62] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS*, pages 68–75, New York, NY, USA, 1997.

[63] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232, 2015.

[64] Eiman Ebrahimi et al. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO*, pages 316–326, New York, USA, 2009.

[65] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Prefetch-aware shared-resource management for multi-core systems. In *ISCA*, pages 141–152, June 2011.

[66] Eiman Ebrahimi, Onur Mutlu, and Yale N Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, pages 7–17, Feb 2009.

[67] Victor M Eguiluz, Dante R Chialvo, Guillermo A Cecchi, Marwan Baliki, and A Vania Apkarian. Scale-free brain functional networks. *Physical review letters*, 94(1):018102, 2005.

[68] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie. Computational RAM: implementing processors in memory. *IEEE Design Test of Computers*, 16(1):32–41, Jan 1999.

[69] Mojtaba Eskandari and Hooman Raesi. Frequent sub-graph mining for intelligent malware detection. *Security and Communication Networks*, 7(11):1872–1886, 2014.

[70] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376. IEEE, 2011.

[71] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 175–184, New York, NY, USA, 2006. Association for Computing Machinery.

[72] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[73] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 212–223, New York, NY, USA, 1998. Association for Computing Machinery.

[74] Adi Fuchs, Shie Mannor, Uri Weiser, and Yoav Etsion. Loop-aware memory prefetching using code block working sets. In *MICRO*, pages 533–544, Washington, DC, USA, 2014.

[75] A. Garg and M. C. Huang. A performance-correctness explicitly-decoupled architecture. In *MICRO*, pages 306–317, Nov 2008.

[76] Johannes Gehrke, Paul Ginsparg, and Jon Kleinberg. Overview of the 2003 kdd cup. *Acm SIGKDD Explorations Newsletter*, 5(2):149–151, 2003.

[77] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *MICRO*, 2020.

[78] Gurbinder Gill et al. Abelian: A compiler for graph analytics on distributed, heterogeneous platforms. In *Euro-Par*, pages 249–264, 2018.

[79] Joshua Glasser and Brian Lindauer. Bridging the gap: A pragmatic approach to generating insider threat data. In *2013 IEEE Security and Privacy Workshops*, pages 98–104. IEEE, 2013.

[80] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the Terasys massively parallel PIM array. *Computer*, 28(4):23–31, Apr 1995.

[81] Andrew Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research, July 2004.

[82] James R Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R Pleszkun, PB Schechter, and Honesty C Young. Pipe: A vlsi decoupled architecture. *SIGARCH Computer Architecture News*, 13(3):20–27, June 1985.

[83] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems*, 187:104816, 2020.

[84] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *PPoPP*, pages 246–260, 2018.

[85] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864, New York, NY, USA, 2016. Association for Computing Machinery.

[86] László Hajdu and Miklós Krész. Temporal network analytics for fraud detection in the banking sector. In *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*, pages 145–157. Springer, 2020.

[87] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, pages 1–13, 2016.

[88] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[89] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. In *ISCA*, page 243–254. IEEE Press, 2016.

[90] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition*, 2016.

[91] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.

[92] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

[93] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–362, 2012.

[94] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *Acm Sigplan Notices*, 46(8):267–276, 2011.

[95] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[96] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[97] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *IEEE conference on computer vision and pattern recognition*, 2017.

[98] Christopher J. Hughes and Sarita V. Adve. Memory-side prefetching for linked data structures for processor-in-memory systems. *J. Parallel Distrib. Comput.*, 65(4):448–463, April 2005.

[99] Yuriy Hulovatyy, Huili Chen, and Tijana Milenković. Exploring the structure and function of temporal networks with dynamic graphlets. *Bioinformatics*, 31(12):i171–i180, 2015.

[100] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *ICS*, pages 499–500, New York, NY, USA, 2009.

[101] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. {ASAP}: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, 2018.

[102] A. Jain and C. Lin. Rethinking belady's algorithm to accommodate prefetching. In *ISCA*, pages 110–123, June 2018.

[103] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, page 247–259, New York, NY, USA, 2013.

[104] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[105] Kasra Jamshidi and Keval Vora. A deeper dive into pattern-aware subgraph exploration with peregrine. *SIGOPS Oper. Syst. Rev.*, 55(1):1–10, June 2021.

[106] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[107] Di Jin, Mark Heimann, Ryan A Rossi, and Danai Koutra. node2bits: Compact time-and attribute-aware node representations for user stitching. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2019.

[108] Lizy Kurian John, Vinod Reddy, Paul T Hulina, Lee DL. K. John Coraor, et al. Program balance and its impact on high performance risc architectures. In *HPCA*, pages 370–379, Jan 1995.

[109] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *SIGARCH Computer Architecture News*, 25(2):252–263, May 1997.

[110] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[111] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. The triejax architecture: Accelerating graph operations through relational joins. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1217–1231, New York, NY, USA, 2020. Association for Computing Machinery.

[112] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *HPCA*, pages 206–217. IEEE, 2000.

[113] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher,

Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 790–803, 2020.

[114] M. Khan and E. Hagersten. Resource conscious prefetching for irregular applications in multicores. In *SAMOS*, pages 34–43, July 2014.

[115] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. Scalable simd-efficient graph processing on gpus. In *PACT*, pages 39–50. IEEE, 2015.

[116] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, ZeshanJ. Kim Chishti, et al. Path confidence based lookahead prefetching. In *MICRO*, pages 1–12, Oct 2016.

[117] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *SIGOPS Operating Systems Review*, 51(2):737–749, April 2017.

[118] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Comput. Archit. Lett.*, 15(1):45–49, January 2016.

[119] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[120] Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

[121] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers accelerating index traversals for in-memory databases. In *MICRO*, pages 468–479, Dec 2013.

[122] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *PACT*, pages 268–279, Washington, DC, USA, 2001. IEEE Computer Society.

[123] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[124] Lauri Kovanen, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences. *Proceedings of the National Academy of Sciences*, 110(45):18070–18075, 2013.

[125] Rohit Kumar and Toon Calders. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *Proceedings of the VLDB Endowment*, 11(11):1441–1453, 2018.

[126] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1269–1278, 2019.

[127] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, New York, NY, USA, 2010. ACM.

[128] Mayank Lahiri and Tanya Y. Berger-Wolf. Structure prediction in temporal networks using frequent subgraphs. In *2007 IEEE Symposium on Computational Intelligence and Data Mining*, pages 35–42, 2007.

[129] N. B. Lakshminarayana and H. Kim. Spare register aware prefetching for graph algorithms on gpus. In *HPCA*, pages 614–625, Feb 2014.

[130] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[131] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*. Springer, 2012.

[132] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N Patt. Prefetch-aware dram controllers. In *MICRO*, pages 200–209, Washington, DC, USA, 2008. IEEE Computer Society.

[133] Sukhan Lee, Shin haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[134] Yutaka I. Leon-Suematsu, Kentaro Inui, Sadao Kurohashi, and Yutaka Kidawara. Web spam detection by exploring densely connected subgraphs. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '11, page 124–129, USA, 2011. IEEE Computer Society.

[135] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Governance in social media: A case study of the wikipedia promotion process. In *AAAI Conference on Web and Social Media*, 2010.

[136] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650, 2010.

[137] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005.

[138] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[139] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[140] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[141] Jure Leskovec, Mary McGlohon, Christos Faloutsos, Natalie Glance, and Matthew Hurst. Cascading behavior in large blog graphs: Patterns and a model. *Society of Applied and Industrial Mathematics: Data Mining*, 2007.

[142] Ted G Lewis. *Network science: Theory and applications*. John Wiley & Sons, 2011.

[143] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788. IEEE, 2021.

[144] Jundong Li, Harsh Dani, Xia Hu, Jiliang Tang, Yi Chang, and Huan Liu. Attributed network embedding for learning in a dynamic environment. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 387–396, 2017.

[145] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[146] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, pages 469–480, Dec 2009.

[147] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[148] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Transactions on Computers*, 2020.

[149] Colin Yu Lin, Ngai Wong, and Hayden Kwok-Hay So. Design space exploration for sparse matrix-matrix multiplication on fpgas. *International Journal of Circuit Theory and Applications*, 41(2):205–219, 2013.

[150] Mikko H Lipasti, William J Schmidt, Steven R Kunkel, and Robert R Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *MICRO*, pages 231–236, Nov 1995.

[151] Or Litany, Tal Remez, Emanuele Rodola, Alex Bronstein, and Michael Bronstein. Deep functional maps: Structured prediction for dense shape correspondence. *ICCV*, 2017.

[152] Paul Liu, Austin Benson, and Moses Charikar. A sampling framework for counting temporal motifs. *arXiv preprint arXiv:1810.00980*, 2018.

[153] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G Abraham. Dynamic helper threaded prefetching on the sun ultrasparc/spl reg/ cmp processor. In *MICRO*, pages 12–104, Nov 2005.

[154] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, New York, NY, USA, 2005.

[155] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, 2019.

[156] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE international conference on big data (big data)*, pages 3972–3979. IEEE, 2018.

[157] William Mangione-Smith, Santosh G Abraham, Edward SW. Mangione-Smith Davidson, 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*, volume 1, pages 288–296 vol.1, Jan 1990.

[158] Sergei Maslov and Kim Sneppen. Specificity and stability in topology of protein networks. *Science*, 296(5569):910–913, 2002.

[159] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, , and B. Wu. Graphzero: Breaking symmetry for efficient graph mining. In *arXiv preprint arXiv:1911.12877*, 2019.

[160] Daniel Mawhirter and Bo Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 509–523, New York, NY, USA, 2019. Association for Computing Machinery.

[161] Julian J McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In *NIPS*, volume 2012, pages 548–56. Citeseer, 2012.

[162] P. J. Meaney, L. D. Curley, G. D. Gilda, M. R. Hodges, D. J. Buerkle, R. D. Siegl, and R. K. Dong. The ibm z13 memory subsystem for big data. *IBM J. Res. Dev.*, 59(4–5):4:1–4:11, July 2015.

[163] Cem Meydan, Hasan H Otu, and Osman Uğur Sezerman. Prediction of peptides binding to mhc class i and ii alleles by temporal motif mining. In *BMC bioinformatics*, volume 14, pages 1–11. Springer, 2013.

[164] P. Michaud. Best-offset hardware prefetching. In *HPCA*, pages 469–480, March 2016.

[165] Micron. DDR4 SDRAM Data sheet, MT40A2G4, MT40A1G8, MT40A512M16, 2015.

[166] Tomas Mikolov. Word2vec implementation in C. https://github.com/tmikolov/word2vec.

[167] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *International Conference on Representation Learning*, 2013.

[168] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[169] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, New York, NY, USA, 2007.

[170] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael M Bronstein. Fake news detection on social media using geometric deep learning. *arXiv preprint arXiv:1902.06673*, 2019.

[171] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):87–106, June 1991.

[172] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2018.

[173] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing. *AGP'17*, 2017.

[174] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural Support for Synchronization-and Bandwidth-Efficient Commutative Scatter Updates. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, October 2019.

[175] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to understand large caches. In *HP laboratories*, 2009.

[176] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *HPCA*, pages 129–140, Feb 2003.

[177] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, pages 96–96, Feb 2004.

[178] Mark EJ Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[179] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. Continuous-time dynamic network embeddings. In *Companion Proceedings of the The Web Conference 2018*, pages 969–976, 2018.

[180] Tam Nguyen. Cuda implementation of cbow word2vec. https://github.com/cudabigdata/word2vec_cuda.

[181] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. Prefedge: Ssd prefetcher for large-scale graph traversal. In *SYSTOR*, pages 4:1–4:12, New York, NY, USA, 2014. ACM.

[182] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In *ISCA*, pages 96–109, June 2018.

[183] NVIDIA. Nvidia nsight compute cli. https://docs.nvidia.com/nsight-compute/NsightComputeCli/.

[184] NVIDIA Forum. https://forums.developer.nvidia.com/t/some-questions-about-metrics-global-hit-rate-stall-constant-memory-dependency-etc-of-nvprof/63997.

[185] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. *SIGARCH Comput. Archit. News*, 26(3):192–203, April 1998.

[186] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy efficient architecture for graph analytics accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 166–177, 2016.

[187] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[188] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *OOPSLA*, pages 1–19, 2016.

[189] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[190] Raj Kumar Pan and Jari Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, 2011.

[191] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Temporal motifs code in SNAP. https://snap.stanford.edu/temporal-motifs/code.html.

[192] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*, pages 601–610, 2017.

[193] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5363–5370, 2020.

[194] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): chips that remember and compute. In *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, pages 224–225, Feb 1997.

[195] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, March 1997.

[196] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. Opportunistic computing in gpu architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 210–223, New York, NY, USA, 2019. Association for Computing Machinery.

[197] Ardavan Pedram, Stephen Richardson, Mark Horowitz, Sameh Galal, and Shahar Kvatinsky. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE Design & Test*, 34(2):39–50, 2016.

[198] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *ISCA*, pages 285–297, June 2015.

[199] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[200] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *ACM SIGKDD*, 2014.

[201] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 565–581, 2016.

[202] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 565–581, 2016.

[203] B. Prakash. Graph mining for cyber security. *Advances in Information Security*, 56:287–306, 04 2015.

[204] Giulia Preti, Gianmarco De Francisci Morales, and Matteo Riondato. Maniacs: Approximate mining of frequent subgraph patterns through sampling. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1348–1358, 2021.

[205] Maria Giulia Preti, Thomas AW Bolton, and Dimitri Van De Ville. The dynamic functional connectome: State-of-the-art and perspectives. *Neuroimage*, 2017.

[206] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, RajeevS. H. Pugsley Balasubramonian, et al. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *HPCA*, pages 626–637, Feb 2014.

[207] PyTorch. Pytorch c++ api. https://pytorch.org/cppdocs/.

[208] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–921, 2020.

[209] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1091–1105, New York, NY, USA, 2021. Association for Computing Machinery.

[210] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Intersectx: An efficient accelerator for graph mining. *arXiv:2012.10848v4*, 2021.

[211] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Sparsecore: Stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 186–199, New York, NY, USA, 2022. Association for Computing Machinery.

[212] Tashin Reza, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. Approximate pattern matching in massive graphs with precision and recall guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1115–1131, 2020.

[213] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 385–394, 2017.

[214] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[215] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[216] Amir Roth, Andreas Moshovos, and Gurindar S Sohi. Dependence based prefetching for linked data structures. *SIGOPS Operating Systems Review*, 32(5):115–126, October 1998.

[217] Polina Rozenshtein and Aristides Gionis. Mining temporal networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 3225–3226, 2019.

[218] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, 2020.

[219] Ilie Sarpe and Fabio Vandin. Presto: Simple and scalable sampling techniques for the rigorous approximation of temporal motif counts. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 145–153. SIAM, 2021.

[220] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[221] Albert Segura, Jose-Maria Arnau, and Antonio González. SCU: A GPU Stream Compaction Unit for Graph Processing. In *ISCA*, pages 424–435, 2019.

[222] Albert Segura, Jose-Maria Arnau, and Antonio González. Scu: a gpu stream compaction unit for graph processing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 424–435. IEEE, 2019.

[223] Jiwon Seo, Stephen Guo, and Monica S Lam. Socialite: Datalog extensions for efficient social network analysis. In *IEEE International Conference on Data Engineering (ICDE)*, pages 278–289, 2013.

[224] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 185–197, 2013.

[225] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287, 2017.

[226] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, pages 355–366, Sep. 2012.

[227] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *TACO*, 11(4):51:1–51:22, January 2015.

[228] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[229] Huijuan Shao, Manish Marwah, and Naren Ramakrishnan. A temporal motif mining approach to unsupervised energy disaggregation: Applications to residential and commercial buildings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, pages 1327–1333, 2013.

[230] Shai S Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. Network motifs in the transcriptional regulation network of escherichia coli. *Nature genetics*, 31(1):64–68, 2002.

[231] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. Efficiently prefetching complex address patterns. In *MICRO*, pages 141–152, Dec 2015.

[232] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.

[233] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[234] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *IEEE DCC*, pages 403–412, 2015.

[235] Trevor M Simonton and Gita Alaghband. Efficient and accurate word2vec implementations in gpu and shared-memory multicore architectures. In *HPEC*, pages 1–7. IEEE, 2017.

[236] Shreyas G. Singapura, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. Oscar: Optimizing scratchpad reuse for graph processing. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[237] James E. Smith. Decoupled access/execute computer architectures. *SIGARCH Computer Architecture News*, 10(3):112–119, April 1982.

[238] James E Smith, Shlomo Weiss, Nicholas Y Smith Pang, et al. A simulation study of decoupled architecture computers. *IEEE Transactions on Computers*, C-35(8):692–702, Aug 1986.

[239] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *ISCA*, pages 252–263, June 2006.

[240] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543, 2018.

[241] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, Yale NS. Srinath Patt, et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, pages 63–74, Feb 2007.

[242] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *MICRO*, pages 148–159, Dec 1998.

[243] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE, 2020.

[244] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 2020.

[245] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 16:1–16:10, 2017.

[246] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 2015.

[247] Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, Peer Bork, et al. String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47(D1):D607–D613, 2019.

[248] Nishil Talati, Ameer Haj Ali, Rotem Ben Hur, Nimrod Wald, Ronny Ronen, Pierre-Emmanuel Gaillardon, and Shahar Kvatinsky. Practical challenges in delivering the promises of real processing-in-memory machines. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1628–1633. IEEE, 2018.

[249] Nishil Talati, Saransh Gupta, Pravin Mane, and Shahar Kvatinsky. Logic design within memristive memories using memristor-aided logic (magic). *IEEE Transactions on Nanotechnology*, 15(4):635–650, 2016.

[250] Nishil Talati, Heonjae Ha, Ben Perach, Ronny Ronen, and Shahar Kvatinsky. Concept: A column-oriented memory controller for efficient memory and pim operations in rram. *IEEE Micro*, 39(1):33–43, 2019.

[251] Nishil Talati, Di Jin, Haojie Ye, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. A deep dive into understanding the random walk-based temporal graph learning. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 87–100. IEEE, 2021.

[252] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 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)*, pages 654–667, 2021.

[253] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Ndminer: Accelerating graph pattern mining using near data processing. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 2022.

[254] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.

[255] Lei Tang and Huan Liu. Graph mining applications to social network analysis. In *Managing and mining graph data*, pages 487–513. Springer, 2010.

[256] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, page 425–440, New York, NY, USA, 2015. Association for Computing Machinery.

[257] Nigel Topham, Alasdair Rawsthorne, Callum McLean, Muriel Mewissen, and Peter Bird. Compiling and optimizing for decoupled architectures. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 40–40, Dec 1995.

[258] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*, 2019.

[259] Kun Tu, Jian Li, Don Towsley, Dave Braines, and Liam D Turner. Network classification in temporal networks using motifs. *arXiv preprint arXiv:1807.03733*, 2018.

[260] Johan Ugander, Lars Backstrom, and Jon Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, page 1307–1318, New York, NY, USA, 2013. Association for Computing Machinery.

[261] 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)*, pages 372–377, Oct 1999.

[262] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[263] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*, pages 237–251, 2017.

[264] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM international conference on information & knowledge management*, pages 1505–1514, 2020.

[265] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 763–782, USA, 2018. USENIX Association.

[266] Minjie Yu Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[267] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *SIGMOD*, page 1813–1828, New York, NY, USA, 2016.

[268] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical off-chip meta-data for temporal memory streaming. In *HPCA*, pages 79–90, Feb 2009.

[269] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. PACMan: Prefetch-Aware Cache Management for high performance caching. In *MICRO*, pages 442–453, Dec 2011.

[270] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal prefetching without the off-chip metadata. In *MICRO*, pages 996–1008, New York, NY, USA, 2019.

[271] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient metadata management for irregular data prefetching. In *ISCA*, pages 449–461, New York, NY, USA, 2019.

[272] Dongkuan Xu, Wei Cheng, Dongsheng Luo, Xiao Liu, and Xiang Zhang. Spatio-temporal attentive rnn for node classification in temporal attributed graphs. In *IJCAI*, 2019.

[273] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[274] Takuma Yamaguchi and Federico Busato. Accelerating matrix multiplication with block sparse format and nvidia tensor cores, 2021.

[275] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Characterizing and Understanding GCNs on GPU. *IEEE Computer Architecture Letters*, 19(01):1–1, jan 5555.

[276] Mingyu Yan et al. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[277] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, et al. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *MICRO*, pages 615–628, New York, NY, USA, 2019. ACM.

[278] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 615–628, New York, NY, USA, 2019. Association for Computing Machinery.

[279] Chia-Lin Yang and Alvin R. Lebeck. A programmable memory hierarchy for prefetching linked data structures. In *ISHPC*, pages 160–174, London, UK, UK, 2002. Springer-Verlag.

[280] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue. A locality-aware energy-efficient accelerator for graph mining applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 895–907, 2020.

[281] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *ACM SIGKDD*, 2018.

[282] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect memory prefetcher. In *MICRO*, pages 178–190, Dec 2015.

[283] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88:106848, 2020.

[284] Hanqing Zeng and Viktor Prasanna. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *International Symposium on Field-Programmable Gate Arrays*, pages 255–265, 2020.

[285] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, et al. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454*, 2020.

[286] Dan Zhang et al. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *ASPLOS*, pages 593–607, New York, NY, USA, 2018.

[287] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *PPoPP*, pages 183–193, 2015.

[288] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, 2018.

[289] Weifeng Zhang, Dean M Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *HPCA*, pages 85–95, Washington, DC, USA, 2007.

[290] Yunming Zhang et al. Optimizing ordered graph algorithms with graphit. In *CGO*, page 158–170, New York, NY, USA, 2020.

[291] Yunming Zhang et al. Optimizing ordered graph algorithms with graphit. In *CGO*, 2020.

[292] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, 2017.

[293] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *OOPSLA*, 2:121:1–121:30, October 2018.

[294] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.

[295] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. Architectural implications of graph neural networks. *IEEE Computer Architecture Letters*, 19(1):59–62, 2020.

[296] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT*, pages 231–242, Washington, DC, USA, 2005.

[297] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[298] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[299] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 712–725, New York, NY, USA, 2019. Association for Computing Machinery.

[300] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu. Embedding temporal network via neighborhood formation. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2857–2866, 2018.