Graph Locality Prefetcher for Graph Database

by

Zhuoran Yin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions:

Contributor	Contribution
Zhuoran Yin	Manuscript writing and graph database prefetcher framework design.
Patel, H.	Manuscript editing, and feedback.

Abstract

This work presents a hardware prefetcher to improve the performance of accessing graph data representing large and complex networks. We represent complex networks as graphs, and queries amount to traversals on the graph. Unlike conventional memory hierarchies that exploit spatial and temporal locality, we observe that graph traversals do not necessarily exhibit these same notions of locality. This results in degraded performance of the memory hierarchy. Consequently, our hardware prefetcher exploits locality that is intrinsic to graph traversals, which we call graph-locality to improve the performance of the memory hierarchy. We design and evaluate our prototype using a micro-architectural simulator, and deploy benchmarks from GDBench that is oriented to evaluate the performance of graph database systems.

Acknowledgements

I am extremely grateful to my supervisor Prof. Hiren D. Patel for his guidance, support, and dedication throughout this thesis and during my graduate studies. His mentorship was paramount in providing a well rounded experience in my career. This work is not possible without his consistent enthusiasm and encouragement. I thank my thesis committee members, Prof. Werner M. Dietl and Prof. Wojciech Golab for reviewing this thesis and for their constructive feedback.

I feel extremely fortunate to have interacted with amazing instructors, academics, and fellow students during my time here at UWaterloo. I sincerely thank you all for your wisdom and inspiration. I thank my colleagues at the CAESR lab Anirudh Mohan Kaushik, Mohamed Hassan, and Dan Wang for their knowledge and support during my graduate studies. Special thanks to my friends that accompany me with my graduate studies: Chengpei Shi, Shaocong Ren, Siyuan Chen and Mingsong Liu.

Finally, my heartfelt thanks to my parents for their unconditional love and support. They are the sole reason for my success, happiness, and making my dreams come true.

Dedication

This thesis is dedicated to my parents Mrs. Wen Tian and Mr. Baojie Yin. This thesis would not be possible without their constant love, support, and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graph data structures play an important role in our lives today. They are used in applications that require representing complex relationships with multiple interacting entities. The objective is to use this information to infer interesting properties about the past, and use it to predict events of the future. Real-world examples of applications that use graph data structures include social networks such as Facebook and Twitter, recommendation systems such as those used by Amazon and Netflix, and security threat analysis systems employed by government agencies. Each of these use information about relationships to infer trends and future events.

Graph databases (GDB)s are database software systems that represent its stored information in the form of a graph. There are multiple methods of representing the information as a graph such as native-store versus non-native store. The former refers to the GDB using a graph data structure to represent the information whereas the latter refers to alternate non-graph data structures to represent the information. Similarly, there are GDBs that offer distributed computation on graphs or even in-memory GDBs where the entire graph resides on the local main memory. There are several examples of graph databases currently in use in industry such as Titan [1], OrientDB [2], and Neo4j [3]. An important component of the database software system is to deliver responses to queries quickly. These queries typically use variants of graph traversal algorithms such as breadth-first search or depth-first search to traverse the graph data structure. However, the graphs that represent today's complex networks are expected to grow in size. As a result, queries on such graphs will consume either large or impractical amounts of time rendering GDBs unusable. Consequently, there is considerable interest in expediting queries on large graphs for GDBs. In response to this challenge, researchers have investigated distributed GDBs [1] [4] [5], in-memory GDBs [6] [7], different internal storage data structures, and also using conventional relational database management systems (RDBM)s to accelerate queries on GDBs.

Current efforts towards addressing this challenge are centred towards software techniques such as pre-query clustering, effective storage, and caching for performance improvements. In this work, we take a different approach to addressing this challenge by investigating micro-architectural techniques to improve the performance of queries on large graphs. In particular, our work focuses on developing hardware prefetchers to improve the performance of accessing graph data representing large and complex networks in a computer's memory hierarchy.

A processor has multiple levels of memory, which builds up to a hierarchy. In the hierarchy, the closer memory is smaller in size but faster to access, whereas the further memory is larger in capacity but slower to access. For example, modern computing platforms use caches to exploit temporal and spatial locality. Temporal locality assumes that most recently accessed data are likely to be used in the near future, and spatial locality assumes that data near most recently accessed data are likely to be used in the future. However, the behaviour of complex workloads like GDBs, which use graph traversal algorithms to traverse the graph data structure is highly data dependent. Since every graph has a distinct structure, the assumption of locality may not hold. Therefore, speculatively bringing nodes that are spatially close to each other in memory does not consider whether the traversal may access them. This is because these spatially close nodes may not even be connected to the currently accessed node. Likewise, temporal locality may not be beneficial for subsequent traversals for repeated accesses. Thus, neither temporal nor spatial locality are appropriate means to capture the locality of graph traversals. This results in degraded performance of the memory hierarchy. Therefore, the challenge is to discover techniques to capture locality intrinsic to graphs, and leverage them in hardware to improve memory accesses for graph traversals.

Our first attempt at addressing this challenge resulted in hardware prefetchers that better capture the locality of traversal algorithms on graphs, which we call graph-locality. In particular, we focus on the breadth-first traversal algorithm. We propose three different prefetchers and call them graph-locality prefetchers (GLP)s. The main components of the GLPs include registers, buffers, a predictor, and a request initiator. The buffers are dedicated graph storage space that keep the most recent graph data available for future accesses. Unlike conventional prefetchers, GLP predictors use the accesses made by the processor in conjunction with the accessed data to propose predictions. The request initiator transmits these prediction addresses to lower-level GLPs or the main memory to retrieve the prefetch the appropriate data.

We design and evaluate our prototype using a micro-architectural simulator called Sniper-Sim [8], and deploy benchmarks from GDBench [9] on a graph database (GDB) that we built in-house. There are three prefetching policies that we develop: GLP1-simple, GLP2-aggressive, and GLP3-advanced. We compare the performance of these three GLPs against traditional configurations without any prefetchers, and with global-history buffer (GHB) and stream (STR) prefetchers. Our experiments showed that GLPs improved performance ranging from 11% to

39% for the benchmarks in GDBench when compared to traditional memory hierarchies without prefetchers. When using stream prefetchers, we noticed performance improvements over the traditional memory hierarchies ranging from 6% to 25%. Hence, we believe that this is a promising start to exploring GLPs for GDBs.

Chapter 2

Background

Modern computers are based on the Von Neumann model. The Von Neumann model defines the stored-program concept. This model requires memory storage to hold both instructions that define operations of the program, and the data. It uses a program counter to define a total order on program execution. This program counter points to an instruction in the memory. In one execution cycle, the processor fetches instructions pointed by the program counter, reads the necessary inputs for the execution of that instruction, executes the operations for the instructions, and writes the output. It is well known that the speed of the processor is significantly faster than the speed of the memory. The simplest way to address this speed discrepancy is to stall the execution of the processor when it accesses the memory. This degrades performance, and it is known as the Von Neumann bottleneck.

To address this bottleneck, the processor employs a hierarchy of memory storage to balance the need for high performance, low cost, and high capacity. The technology and the capacity largely determine the performance of the storage. For example, high performance memory storage is more costly per byte when compared to lower performance memory storage. Similarly, higher capacity (also referred to as density) storage is more costly per byte compared to those with lower capacity. A hierarchy of memory storage addresses the tradeoffs by using small but fast memories closer to the processor, and slower but larger ones further away from the processor. This allows the processor to directly interact with the fastest memory, but also provides the processor with a large memory storage when combining the capacity of all memories in the hierarchy.

2.1 Memory Hierarchy

A memory hierarchy consists of primary and secondary storage. Primary storage refers to registers, caches and memories while secondary storage refers to disks. Primary storage is close to the processor in the hierarchy of memories. This means that primary storage must be capable of delivering high performance. Primary storage can be further categorized into on-chip storage, and off-chip storage as shown in Figure 2.1. The size of the primary storage is usually in the order of several megabytes. Comparatively, secondary storage is slower, but it has a larger capacity. Primary storage is volatile whereas secondary storage is non-volatile.

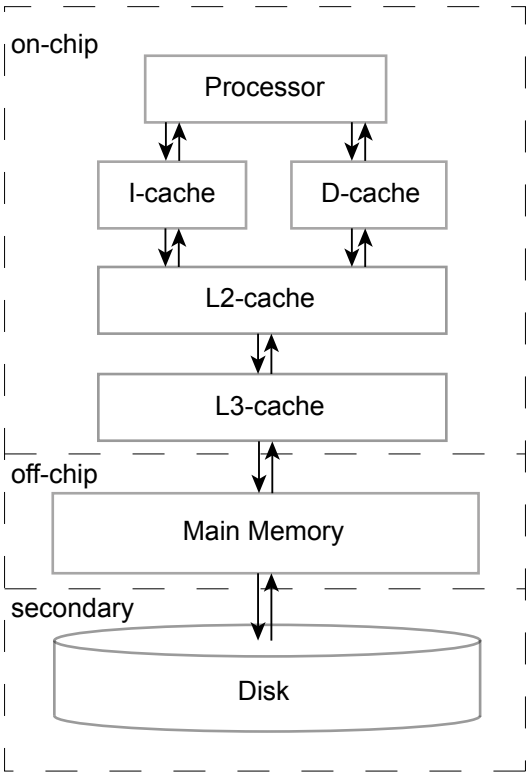

Figure 2.1: Memory hierarchy.

The top-most level of the memory hierarchy is the primary on-chip storage, which includes processor registers and caches. Registers and caches are both made up of Static Random-Access Memory (SRAM) technology. A single SRAM-based memory cell uses six transistors. Registers are essential to a processor to temporarily hold words of data for intermediate computation. Since registers use SRAM technology, they are fast. Caches also use SRAM technology, but are

significantly larger in capacity than registers. The size of the memory also determines its access latency where the larger the memory the slower it is to access. Note that most modern processors have multiple levels of cache memories in the hierarchy. Higher levels of caches are closer to the processor, and smaller in size so as to yield low access latencies. Lower levels of caches are larger in size, but slower to access. Unlike the registers that are directly referenced by instructions of an instruction-set architecture (ISA), caches are not directly addressed via instructions. However, caches play an integral role in improving the performance of modern microprocessors.

Off-chip storage typically refers to memories that use Dynamic Random-Access Memory (DRAM) technology, and are those that are off the chip. A memory cell using DRAM technology uses a transistor and a capacitor. The charge in the capacitor indicates the value in the memory cell. For example, when the capacitor is charged the value can be a one, and zero otherwise. Since capacitors leak small amounts of current, off-chip memories require being refreshed to maintain the state of the memory. This refresh behaviour distinguishes itself from memories that use SRAM, and makes it a significantly slower than SRAM-based memories. However, the amount of storage that DRAM memories can hold is much larger, and the cost per byte is lower than that of SRAM memories.

Secondary storage is on the bottom level of the memory hierarchy. Secondary storage usually refers to hard disk drives (HDD)s. Data on HDDs are in magnetic format and are non-volatile. The access latency for a HDD is on the level of microseconds. This is because HDDs use mechanical motors to seek and retrieve data from the disks. This results in significantly higher access latencies when compared to other storage.

2.1.1 Locality

Memory hierarchies exploit the principles of locality for performance improvements. There are two basic tenets in the principles of locality: spatial and temporal. Spatial locality refers to speculatively bringing in data that is near in proximity to recently accessed data in the memory. This is based on the intuition that data near recently accessed data is likely to be used in the near future as well. An example of a program that may exhibit such spatial locality is one that iterates over an array. Memory hierarchies exploit spatial locality by fetching data surrounding the accessed data. Temporal locality refers to the principle that data most recently accessed is likely to be accessed in the near future. Global variables in programs can often exploit temporal locality. Memory hierarchies use the principle of temporal locality to decide what data to replace when the capacity of a memory is reached.

2.2 Caches

Traditionally, an on-chip memory hierarchy contains multiple levels of cache memories. These caches are broadly split into two types: primary and secondary caches. Primary caches are first level caches (L1) that directly interact with the processor. They are closer to the processor, and smaller in capacity to ensure low access latencies. There is a distinct L1 cache for instruction and one for data. This separation eliminates structural hazards to the L1 cache.

Non-primary caches are secondary caches, which have more than one level. They are closer to the DRAM with larger capacities but higher access latencies. Second and third level caches (L2 or L3) are unified for both instruction and data. Unifying L2 and L3 allows different programs that have different instructions to data requirements to be accommodated.

The concept of caches includes both the design of the storage structures, and the controller for managing the accesses between them. The cache controller manages all requests sent to and received by the cache. Cache controllers are both receivers and initiator of requests. For example, as a receiver an L1 cache receives requests from the processor, and L2 receives requests from L1. As a sender, an L1 cache sends requests to L2 and L3 sends requests to the DRAM controller.

2.2.1 Memory request

When a processor initiates a memory request, the cache controller for L1 is the first in the memory hierarchy to receive the request. A memory request or access can either be a load (read) or a store (write). A load or store request consists of a memory address and a size. The address indicates the location in the address space to access, and the size denotes the number of bytes requested. The memory request to internal storage of the memory hierarchy works on the granularity of a cache line. A typical cache line is 64 bytes in size. When a cache controller receives the request, it splits the starting address of the request into three fields. They are tag, set and offset from the most significant bit to the least significant bit as in Figure 2.2.

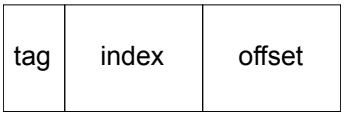

Figure 2.2: Address.

A cache controller uses index bits to reference a frame within the cache that holds a data block, uses offset bits to find a byte within the cache frame, and uses tag bits to check if the current frame is the requested cache line.

2.2.2 Cache frame

A cache frame includes the following fields: tag, data block and flag bits as in Figure 2.3.

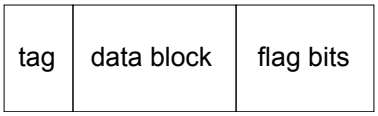

Figure 2.3: Cache frame.

As mentioned above, the tag is part of an address used to compare with the requested address. Data block holds the range of addresses starting from the corresponding tag address. Flag bits are used to indicate the state of the cache line. For uni-processors, the flag bits are typically the valid and dirty bits. The valid bit indicates whether the current cache line is a valid copy of the corresponding block in the memory. The dirty bit denotes whether the current cache line is an updated copy of the corresponding block in the memory.

I explain a simple cache look-up next. Suppose that the processor initiates a load or store request. The cache controller receives this request and splits the address into tag, index, and offset parts. The cache controller uses the index field to retrieve the location of the cache frame. In parallel, the cache controller compares the tag bits to the tag bits stored in the cache line. If the tag bits match and the valid bit is set, then the requested block is already in the cache. This is a cache hit, and the current cache line is returned to the CPU. Otherwise, the requested data is not in the cache and it must be retrieved from lower-level memories in the hierarchy by initiating further requests.

2.2.3 Associativity

The index bits of an address are mapped to a frame group. A frame group contains multiple cache frames within it. These cache frames are called sets, and each frame within a set is called a way. The associativity of a cache indicates the number of ways of the cache. For example, a direct-mapped cache has one-way sets, and the associativity is one. This implies that there is only one cache frame per index. On the other hand, a two-way set-associative cache has two-way sets, which allows for the same index to potentially address two cache frames. Finally, a fully-associative cache has only one set.

2.2.4 Replacement policies

Replacement policies become important for set-associative caches. Once a cache set is full with valid cache lines, the cache controller may need to evict an existing cache line so as to make space for a new cache line. The policy that determines which cache line gets evicted is called the replacement policy. A well-suited eviction policy should preserve the cache lines that are most likely to have access in the future.

The major replacement policies are Random, First-In First-Out (FIFO) and Least Recently Used (LRU). Random policy evicts a randomly selected cache line. FIFO follows the first-in first-out order to evict cache lines based on the order in which they are filled into the cache. LRU picks the least recently accessed cache line to replace based on the premise that it is the least likely to be used in the near future (supporting temporal locality).

2.2.5 Cache misses

A cache miss happens when the cache controller determines that the cache line corresponding to the memory request does not reside in the current cache. There are three categories of cache misses: compulsory, capacity, and conflict. Compulsory misses occur at the first request to a cache line. Compulsory misses is relevant to the size of a cache line. It occurs mostly at the start up stage of the cache when the cache is empty, and every request results in a miss. Capacity misses happen when the size of the cache cannot hold all the cache lines needed by a running program. Capacity miss is relevant to the size of the cache. Conflict misses happen when there is not enough associativity so that only a subset of blocks reside in a set. They are relevant to the associativity of the cache.

Cache parameters have different impact on the three misses as in Table 2.1. We use three symbols in the following table, +, - and 0 to denote the following. The plus symbol means increasing the parameter also increases the miss, the minus symbol means increasing the parameter decreases the miss, and the zero symbol means the miss is irrelevant to that parameter. Naturally,

Miss type	Increase cache size	Increase block size	Increase associativity
Compulsory	+	-	0
Capacity	-	0	0
Conflict	0	+	-

Table 2.1: Impact of increasing cache parameters on three types of misses.

increasing the cache size results in more compulsory. For capacity misses, a larger cache reduces capacity misses. Notice that a larger block size results false sharing yield larger number of conflict misses. However, when the associativity is high, the cache is able to hold the necessary cache lines mapped to a set. Hence, increasing the associativity reduces the conflict misses.

2.3 Prefetcher

Caches in modern processor micro-architectures take advantage of spatial and temporal locality. However, there are also situations when a does not obey these expected principles of locality resulting in performance degradation. This is where prefetching is useful. Prefetching refers to retrieving either instructions or data speculatively into the cache. The prefetching policy or logic can be independent from the notions of locality employed by caches, and the objective of the prefetching is to improve performance. Prefetchers operate in the background, and are effective in hiding the memory access latency. There are instruction and data prefetchers, and there can be hardware or software implementations. This work primarily focuses on a data prefetching.

2.3.1 Prefetching requirements

There are two properties to prefetching that determine the quality of the prefetcher. The first is the timeliness of the prefetching and the second is the accuracy. Timeliness refers to the property that the prefetched data must arrive to its storage *just in time* for the processor to request it. Notice that if prefetched data arrives significantly earlier than when the processor requests it, then it may reside in the cache resulting in the eviction of a potentially useful cache line. Furthermore, requests by the processor may end up evicting the prefetched cache line as well rendering the prefetching futile. Alternatively, when a prefetched block arrives to the storage late, the processor would not benefit from the prefetching. In this case the processor may suffer the access latency of a miss.

Accuracy refers to the ability to speculatively select the appropriate data to prefetch that the processor may request in the future. Prefetching data that will not be used by the processor is wasteful. Furthermore, it may result in additional pollution in the cache where useful cache lines may be evicted to make space for the prefetched line. Note that there is a difference between a miss (referred to as a regular miss) due to the data not residing in the cache versus a miss suffered by the processor because of pollution due to prefetches. A regular miss typically occurs when the processor requests the data and it does not reside in the cache. This regular miss also indicates that the data is necessary for the processor to resume its operation; hence, it is useful. A miss

on an address that indexes a prefetched block is different because there is no guarantee that the prefetched data will ever be useful to the processor as it is a speculation. Therefore, accuracy is an important property of prefetching so as to eliminate pollution and overheads caused by it. Another issue to consider is that redundant prefetches exacerbate memory bandwidth. This means that there may be contention to access the memory if both the prefetcher and the processor make memory requests.

2.3.2 Prefetching parameters

There are certain parameters that define the behaviours of prefetchers. I explain a few of these next.

Prefetch degree

The prefetch degree is the number of blocks to prefetch on every trigger of the prefetcher. For example, a prefetch degree of two dictates that on every miss a new prefetch is triggered that retrieves two blocks at once.

Prefetch distance

The prefetch distance refers to a look-ahead value from where to prefetch the data. For example, consider prefetching data stored in an array accessed within a loop. To account for the memory access latency, prefetches can start three iterations before the data is used to compensate for the access latency. Otherwise, the process would stall waiting for the data to arrive. In this case, the prefetch distance is three.

2.3.3 Software data prefetching

Software data prefetching uses explicit instructions in the instruction-set architecture to perform prefetches, which are called fetch or prefetch instructions. Fetch instructions are non-blocking loads that do not raise exceptions. These fetch instructions execute as regular load instructions. Since the fetch instructions are a part of the instruction-set, programmers or compilers must insert these fetch instructions at the correct locations in the program. The choice of where to insert fetch instructions into a program is known as prefetch scheduling. Once again, the insertion point of

```
1  for (i = 0; i < N; i++)
2    sum = sum + a[i] * b[i];
```

Figure 2.4: Original program.

fetch instructions must carefully consider the property of timeliness. Consider a program that does array operation as in Figure 2.4.

This example calculates the inner product of two arrays `a` and `b`. Both arrays have a size of `N`. The result program after the prefetch scheduling is in Figure 2.5. Three categories of statements exist in the altered program: prefetching prolog, loop unrolling, and prefetching epilog. Lines 2-4 are the prefetching prolog with three fetch statements that retrieve the sum variable, and the first element of the two arrays. Lines 6-13 describes the unrolled loop. The loop is unrolled such that two fetches on `a` and `b` retrieve the next cache line for each of these respective arrays for every four iterations. Therefore, the `sum` statement also constitutes four sub-statements for the calculation, but refers to different locations in the cache line. Lines 15-16 are the epilog that perform the calculation for the last cache line.

```
1  fetch( &sum );
2  fetch( &a[0] );
3  fetch( &b[0] );
4
5  for (i = 0; i < N-4; i += 4 ){
6    fetch( &a[i+4] );
7    fetch( &b[i+4] );
8    sum = sum + a[i] * b[i];
9    sum = sum + a[i+1] * b[i+1];
10   sum = sum + a[i+2] * b[i+2];
11   sum = sum + a[i+3] * b[i+3];
12 }
13
14 for ( i = N-4 ; i < N; i++)
15   sum = sum + a[i] * b[i];
```

Figure 2.5: Prefetch Scheduling.

There are several limitations of software prefetching. To begin with, locations in programs where fetch instructions can be inserted are limited. These locations are generally restricted to tight loops that access arrays or lists. This implies that fetch instructions are used when accesses to memory can be predicted on regular data structures. Another issue with software prefetching is that complex control flow may result in prefetching of unnecessary data. It is unlikely that when a cache line is fetched, subsequent blocks are also fetched because of the control flow. Finally, since fetch instructions are inserted statically, software prefetching is not flexible to

accommodate runtime behaviours and changes. Even in the case that the program is suitable for inserting software fetch instructions, fetches may result in a performance overhead, and code bloat. A fetch instructions by itself is a performance penalty not only because of additional instructions, but also because that in most cases, a processor has to perform some calculations in order to obtain the address to be prefetched. Results of calculations also need be preserved to prevent further duplicate calculations. In this case, there is a resource contention between prefetching and the program itself. The possible result of prefetching might increase the register pressure and the additional spill code is placed in memory. A higher look ahead of prefetching exacerbates performance penalty even more.

2.3.4 Hardware data prefetching

Hardware prefetching takes advantage of the hardware to launch prefetches. No modifications to software are necessary to enable prefetches. In addition, hardware prefetching can utilize run-time information to perform runtime decisions regarding prefetching. Common hardware prefetching techniques include stream and stride prefetching mechanisms.

A memory hierarchy in itself is a simple form of data prefetching. By working at the granularity of cache lines, spatial locality enables retrieving more words of data than requested by the processor. However, there are several disadvantages of increasing the cache lines. The first is cache pollution. In one eviction, more useful words in the current block are replaced to make room for a new block, which potentially causes pollution. The other one is false sharing. For example, if multiple cores access the same block of data, but each processor updates independent words within the same cache line, then large chunks of memory coherency traffic is generated to ensure coherent view on the cache line. Instead of increasing the cache line size, explicit hardware stream prefetching circumvents these disadvantages while allowing for extensions to the principles of locality. Common techniques includes one-block-look-ahead (OBL), tagged prefetch, and sequential prefetching. Figure 2.6 shows an example of hardware prefetching. The figures a, b and c shows the prefetch on miss, tagged prefetch, and sequential prefetching with degree of two, respectively.

Assume that there are three successive accesses to the same sub-figure, which represents five consecutive blocks. Each block within the sub-figure represents a memory address. The memory address for each block is at the granularity of a cache line. If the source of a memory request is from a cache controller, then we define that there is a demand fetch on the block. If the source of a memory request is from a prefetcher, then we define that there is a prefetcher on the block.

In the OBL case, prefetching starts when the cache suffers from a miss. The first iteration results in a miss because the first cache line is not in the cache yet. The first miss triggers a

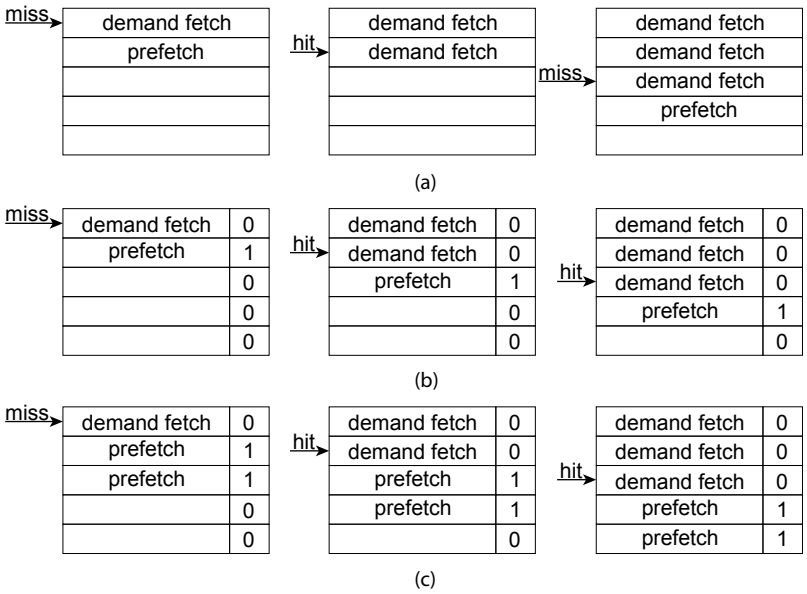

Figure 2.6: (a) Prefetch on miss, (b) Tagged prefetch, (c) Sequential prefetching degree 2.

prefetch for the second block. When there is a demand fetch for the second block, no prefetch happens because this is a cache hit. On the third access, the cache is suffering from another miss which in turn bring in the forth block. There are in total two misses in this access pattern, and the misses are interleaved for every other access in the consecutive accesses.

In the tagged prefetch and sequential prefetching case, prefetching not only happens on a miss, but also on blocks with the tag bit set. On the first access, the demand fetch for the first block results in the prefetch for the second block and the tag bit set. A cache controller set the tags only to mark the prefetched blocks so that they are differentiated from demand fetched cache lines. A demand fetch on the tagged block clears the tag bit and triggers prefetch. In this way, the prefetcher receives confirmation that the prefetched block is accurate, and continues prefetching. When there is a demand fetch on the second block, the prefetcher sets the tag bit back to zero and prefetches the third block. The third access keeps the same access pattern to prefetch the fourth block. The total number of misses for all three accesses is only one.

The only difference between tagged prefetch and sequential prefetching is that, a prefetcher launches multiple prefetches each time in sequential prefetching. For example, on the first access, a demand fetch triggers not only the load of the first block, but also two successive blocks. Similarly, a demand fetch on the second access generates two more accesses. But since the third block is already in cache, the cache controller generates no traffic for it and only the forth block is brought in. So, it is the same with the third access. The total misses for all three access is also

one, but with a higher look ahead, which reduces potential delay in the memory hierarchy.

2.4 Graph database

Graph databases promote high performance, flexibility and agility. Compared to relational databases, there is a performance benefit as data sets increase in size. Relational databases are known to perform poorly on join operations, which are essential in indexing graphs. The situation gets worse as data sets become larger. By comparison, graph queries on graph databases maintain constant performance on large sets of data. Structured data such as tables require knowledge of data to be determined at the beginning in order to connect and design a schema. Graph databases are naturally extensible, and flexible in evolving the overall data domain. For instance, graphs can be extended to existing schemas with new vertexes, edges and sub-graphs. Further, graph representations of data are schema-free. This, together with the graph database's application programming interface, and query language makes developing and managing the application transparent.

2.4.1 Data modelling with property graph

Our graph database uses a property graph model for its data. This model includes vertexes, edges and properties. Vertexes are basic building blocks of a graph, and edges connect vertexes together. Edges have directions that point from start vertexes and end vertexes. Both vertexes and edges are grouped with different types and contain properties. Types and properties are key-value pairs in arbitrary types and length.

Graph database storage

Figure 2.7 shows the UML diagram of graph database storage. There are the storage manager class and storage classes in the graph database. The GraphType class is the high level graph database manager. In the initialization of the GraphType object, a graph reader reads in all the graph data from a file to the GraphType object. Then all the GraphType object is responsible for all graph operations including adding and setting the data and structure of the graph. GraphType object keeps all the storage of the graph in-memory when database is running.

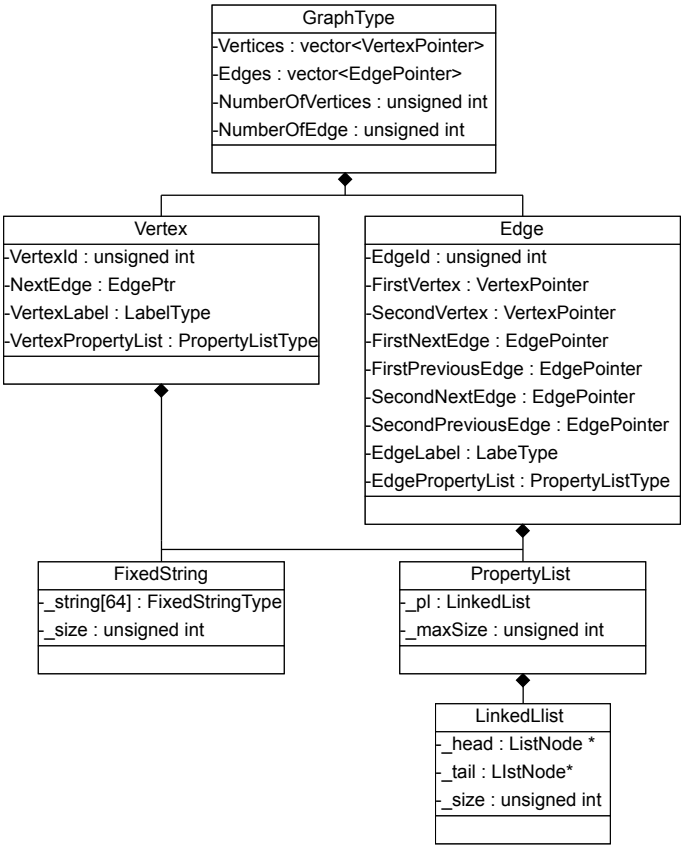

Figure 2.7: Class diagram for storage classes.

Vertex entry

Each entry of vertex is 120 bytes in length. Layout of a single entry in the vertex entry is shown in Figure 2.8. Bytes 1 to 4 is the VertexId and each vertex has a unique id number. Bytes 9 to 16 are the next edge field. This field stores the pointer to vertex's immediate edge. Bytes 17 to 84 are the vertex label field. This field stores the vertex's label, which is capable of storing 64 characters. Bytes 85 to 116 are the vertex property field. This field stores the vertex's VertexPropertyList.

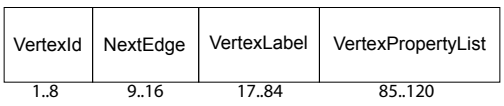

Figure 2.8: Overview of vertex.

Edge entry

Each entry of edge is 160 bytes in length. Layout of a single entry in the edge entry is shown in Figure 2.9. Similar to the vertex entry, the first four bytes store the unique EdgeId. Bytes 9 to 16 and 17 to 24 fields are FirstVertex and SecondVertex denoting the start and end of an edge edge, respectively. The following four fields are indexes to edges associated with the start and end vertexes. They are FirstNextEdge, FirstPreviousEdge, SecondNextEdge, SecondPreviousEdge. Take FirstPrevousEdge as an example, it stores a pointer to the previous edge according to the first vertex. Bytes 57 to 124 represent an EdgeLabel field. It stores the edge's group index, which is capable of storing 64 characters. The last field, from bytes 125 to byte 156 store the edge's EdgePropertyList, which is dynamically allocated.

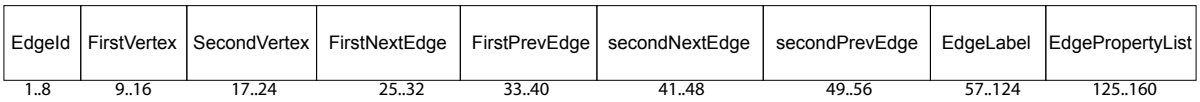

Figure 2.9: Overview of edge (relationship).

Property entry

Each entry of property is 32 bytes in length. A property entry includes a linked list, and an unsigned integer indicating the maximum size of the linked list. Each node of the linked list contains a pair. The key and value pair of the linked list node consists of a FixedString. Each node pair of the linked list stores a separate property value.

2.4.2 Traversal

Traversals are the most important operations in graph databases. A sample traversal to find an immediate friends of a vertex is shown in Figure 2.10. In this example, the Filter in the SelectionVisitor specifies the target vertex through a key-value pair. Then the query performs a breadth-first traversal from the startVertex, and based on the "FRIENDS" type edges until the query discovers the target vertex.

A simplified breadth-first traversal is shown in 2.11. The traversal uses two data structures, a vector and a queue. The queue keeps the vertexes to be visited in later accesses, and the vector of boolean indicates if the current visiting vertex has been visited. A child vertex is pushed into the queue only when the vertex has not been visit; thus, preventing infinite loops in traversing a cycle in a graph.

```
1  SelectionVisitor v1;
2  filtProperty(key, value, v1.getFilter());
3  traverseThroughType("FRIENDS", v1.getFilter());
4  breadthFirstSearch(graph, startVertex, V1);
```

Figure 2.10: A traversal example.

```
1  visited[root] = true;
2  VertexQueue.push_back(root);
3  while(!VertexQueue.empty()){
4    Vertex v = VertexQueue.front();
5    VertexQueue.pop_front();
6    while(v.hasNextChild()){
7      Vertex child = v.nextChild();
8      if(!visited[child]){
9        visited[child] = true;
10       queue.push_back(child);
11     }
12   }
13 }
```

Figure 2.11: A breadth-first traversal.

One iteration of traversal starts from grabbing the start vertex in a queue or stack. Then using the vertexes' and edges' next edge field (through nextChild()), the traversal expands from the current vertex to its edges. After evaluation by SelectionVisitor, the traversal engine makes two decisions - whether to keep traversing or to include the current path into the result set. If the decision is to keep on traversing the remaining graph, the iteration happens on again, but on an newly expanded vertex. This way, traversal repeats the expansion behaviour until it reaches the end of target graph.

The structure of traversal classes that we call traverser in the graph database is presented in Figure 2.12. A traverser is the primary class to traverse the graph, and a traversal can either be breadth-first or depth-first. The key member of the traverser is a graph visitor, which makes run-time decisions when traversing a graph. Users can choose to overload the visitor class so as to change the default decision-making logic for flexible traversing behaviour. In the previous example, a user customizes the Filter of a Visitor by setting values to _key and _value. Then by initializing a SelectionVisitor within the traverser, the visitor compare the property of each visited vertex to determine whether to include the visiting vertex.

The prefetcher hardware requires both static and dynamic information to work properly. Static information does not change between each run of the graph database, whereas dynamic information changes during each execution of a query. Therefore, users must be able to set static parameters before starting the database. In addition, the application needs to inform the

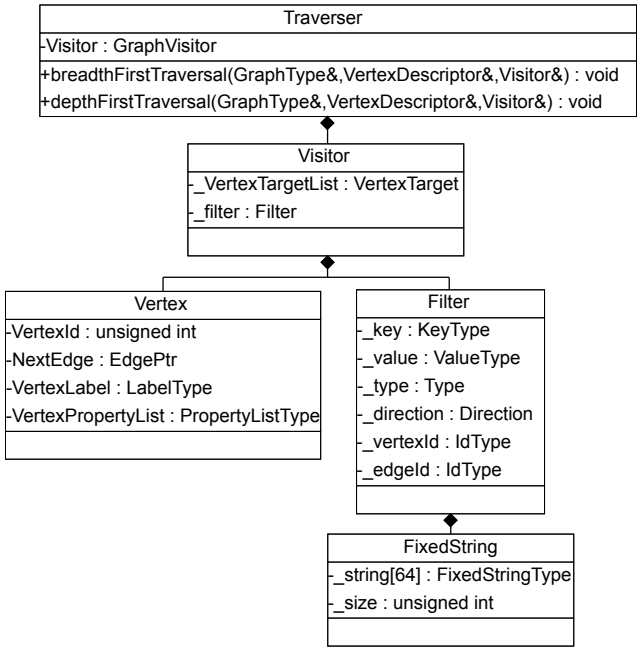

Figure 2.12: Class diagram for Traverser.

prefetcher hardware at run-time of any dynamic information necessary for the hardware to function correctly. This is accomplished by using magic instructions in software to notify the hardware of the necessary information. In hardware, this is equivalent to introducing specific instructions in the instruction-set architecture for these purposes. For further details on the use of magic instructions in SniperSim, please refer to the next section.

2.5 SniperSim

SniperSim [8] is a x86 simulator based on interval core model simulation. The interval core model simulation does not simulate cycles, instead it simulates on the granularity of events on instruction executions. Interval simulation uses an analytical model that determines the timing for each interval. Each interval includes both miss events like branch mis-predictions, cache misses and also smooth execution for a single core. The interval analytical model add the miss latency according to different miss events, which includes either independent miss mis-events and overlapping long-latency miss events. Therefore, the abstraction level for SniperSim is higher, allowing for faster simulation speed.

2.5.1 Magic instructions

Magic instructions in this work use the following template as shown in Figure 2.13. The application uses such magic instructions to pass dynamic run-time information. The simulator recognizes these magic instructions via special $xchg$ instructions. SniperSim intercepts these $xchg$ instructions and executes a predefined function that are known as hook functions. The hooks manager in SniperSim takes control of instruction execution in this case.

```
1  long long simMagic(long long cmd, long long arg0, long long arg1){
2     long long  _cmd = (cmd), _arg0 = (arg0), _arg1 = (arg1), _res;
3     __asm__ __volatile__ (
4     "mov %1, %%rax\n\t"
5     "mov %2, %%rbx\n\t"
6     "mov %3, %%rcx\n\t"
7     "xchg %%bx, %%bx\n\t"
8     : "=a" (_res)              /*output       */
9     : "g"(_cmd),               /*input        */
10      "g"(_arg0),
11      "g"(_arg1)
12    : "%rbx", "%rcx" );        /*clobbered    */
13    _res;
14 }
```

Figure 2.13: Magic instruction.

A description of a magic instruction is as follows. The magic instruction takes three arguments, cmd, $arg0$ and $arg1$. The simulator differentiates between magic instructions by different cmd values in the register rax. Users have the flexibility in defining unique integers to this variable. Both $arg0$ and $arg1$ values are in registers rbx and rcx. The application uses magic instructions to pass two arguments to the simulator. In this way, the prefetcher receives run-time dynamic information from the graph database application.

This work employs a subset of magic instructions available in SniperSim. These are shown in Table 2.2. The instruction type column in the table denotes the instruction templates to implement the magic instructions. There are three magic instruction templates. A magic instruction template ends with the number of arguments to pass to the hardware. For example, $simMagic2$ passes two arguments from the application to the hardware. An application passes information through the in-line magic instruction templates.

Instruction call	Instruction type	Meaning
SimRoiStart()	simMagic0()	Starting point of region of interest for running the benchmark.
SimRoiEnd()	simMagic0()	Ending point of region of interest for running the benchmark.
SimMarker()	simMagic2()	Marks different regions of queries within the region of interest.
SimGetProcId()	simMagic0()	Returns the process id to the application.
SimGetThreadId()	simMagic0()	Returns the thread id to the application.
SimUser()	simMagic2()	Pass dynamic information to the prefetcher.

Table 2.2: Impact of increasing cache parameters on three types of misses.

Chapter 3

Related Work

3.1 Big data platforms

Big data, the technology for storing, transferring and analysing large volume of data sets, is attracting both opportunities and challenges. Big data techniques is shown to beneficial to a wide range of industry-specific opportunities including retail, financial services, telecom and utilities [10]. According to the study in [11], the whole big data revenue is made up of database, computation, application, networking, services and so forth. Data storage, computation and application holds more than 50% of the overall market segments [11].

The big data workloads impose different requirements that make traditional data processing techniques inadequate [12] [13] [14] [15] [16] [17] [18]. Therefore it is necessary to redesign the current computing platforms. However, hardware platforms built for graph databases hardware in prior works remain largely unexplored. For example, Ferdman et al. [12] explored the difference between big data server scale-out workloads and traditional desktop workloads, and gives micro-architectural configurations based on the experimentation. Lotfi-Karmran et al. [13] [14] presented techniques scale up processor performance for efficient server scale-out workloads, However, their experimentations does not include any graph database related applications. Wu et al. [15] [16] proposed hardware assisted range partitioner, and stream buffers to leverage loads/stores in joins operations of relational databases, but the implementation can be applied only to relational databases. Song et al. [17] presented a new graph processing architecture. Their graph processor uses accelerator-based node processor for sparse matrix calculations, but they did not focused on the memory hierarchy design. Neither did they try to optimize the locality metrics for graphs.

3.2 Graph databases platforms

There has been sufficient study of database workloads on modern processors [19] [20]. Research showed that processor is not leveraging database workloads the same as scientific workloads [19]. Processor performance suffers from memory stalls, among which a large proportion are last level data cache misses. The optimization showed that, with careful design of the database and its platform leads to order of magnitude performance improvement [15] [20]. However, these evaluations and optimizations focused only at relational databases.

Recent research showed that graph databases have significant performance advantages over relational databases [21] [22] on structural queries and full-text searches. However, the inherent characteristics of graph computation makes the current computation platform less effective [23]. The challenges lie in several aspects. To begin with, graph computations are largely determined by graph data and structures, which make it hard for the hardware to extract instruction level parallelism. Besides, irregular structure and data in graph computations makes current processor hard to obtain data level parallelism. Since graph traversals exhibit little spatial and temporary locality, the memory hierarchy, especially caches proves to be ineffective [14] [23].

3.3 Graph locality models

The graph locality is the high-level characteristic of a graph, showing locality properties when a graph search executes on a processor. Prior works in building analytical models to understand locality in graphs take traversal-agnostic approaches [24] [25] to construct the model. Yuna et al. presented a new metrics called the vertex distance, and use this metric to model the locality in breadth-first graph traversals [26]. However, their exploration does not account for the interferences arising from multi-cores and parallel queries. Furthermore, they limit their exploration to only breadth-first graph traversals and static graphs. A static graph is made up of fixed sequence of vertexes and edges, and therefore do not evolve over time. This limitation makes it impractical to apply this approach to graph databases, because the graph structure in graph databases may change over time.

3.4 Graph prefetching mechanisms

Prefetching is proved to be effective technique to overcome the memory wall on modern processors [27] [28] [29] [30]. People have developed a variety of ways of prefetching including

both software and hardware approaches. Lee et al. confirmed that the integrated software and hardware prefetching yield performance improvement [28]. However, most of the approaches are confined to regular data structures such as arrays in tight loops, whereas few attempts tried prefetchings on irregular data structures or graphs. Cooksey et al. presented a hardware data prefetcher for irregular data structures [29], but it does not take into account the existence and interaction of several irregular data structures, which is common in graph traversals.

A recent research proposed a SSD software-hardware integrated prefetcher for graph traversals [30], which requires all the vertexes preloaded into the main memory. However, The premise to pre-load large datasets makes it hardly qualified for processing big-data for graph databases. Besides, the implementation focus on large scale traversals makes the prefetcher less likely to be effective for inherent small-scale scale out workloads. Finally, the lack of evaluation on memory hierarchy makes the potential of prefetching under-exploited.

Chapter 4

Prefetchers

The objective of the graph-locality prefetchers (GLP)s is to exploit breadth-first traversal patterns on graphs. The primary reason for doing this is to accelerate the performance of queries in GDBs. We implement hardware prefetchers by introducing GLPs that extend every level of the memory hierarchy.

4.1 An example of a property graph

We start by presenting an example to explain the data structure used in the GDB, and the manner in which the GLPs speculate. The graph data structure representation in our GDB uses a representation that is similar to the one implemented in an open-source GDB called Neo4j [3]. Figure 4.1 is an example of a property graph with 7 nodes, 8 relationships, and 10 properties. Each circle represents a node. For example, the circle with the name "Jerry" is a node. Edges that connect nodes are synonymous to relationships. The arrow head denotes the direction of the relationship pointing from the start to the end node. The text on the top of the edges (relationships) are relationship labels. The edge starting at the node with name "Jerry", and ending at node with name "Tom" is a relationship with a relationship label called "knows". We refer to this as "Jerry" knows "Tom". Notice that properties of nodes are inside boxes of a circle. An example of a property is the"name" of a node. Properties of relationships are at the bottom of the relationships, and inside boxes. For example, the edge between "Jerry" and "Tom" has a relationship with properties "id" and "year". For each node that has more than one neighbour, there is a dashed circle with a direction outside the node. The circle describes the previous and next relationships. The arrows on a dashed circle represents the direction for traversal. That is,

the relationships (incoming and outgoing) are linked together as a doubly-linked list. Hence, a traversal direction can either be clockwise or counter-clockwise.

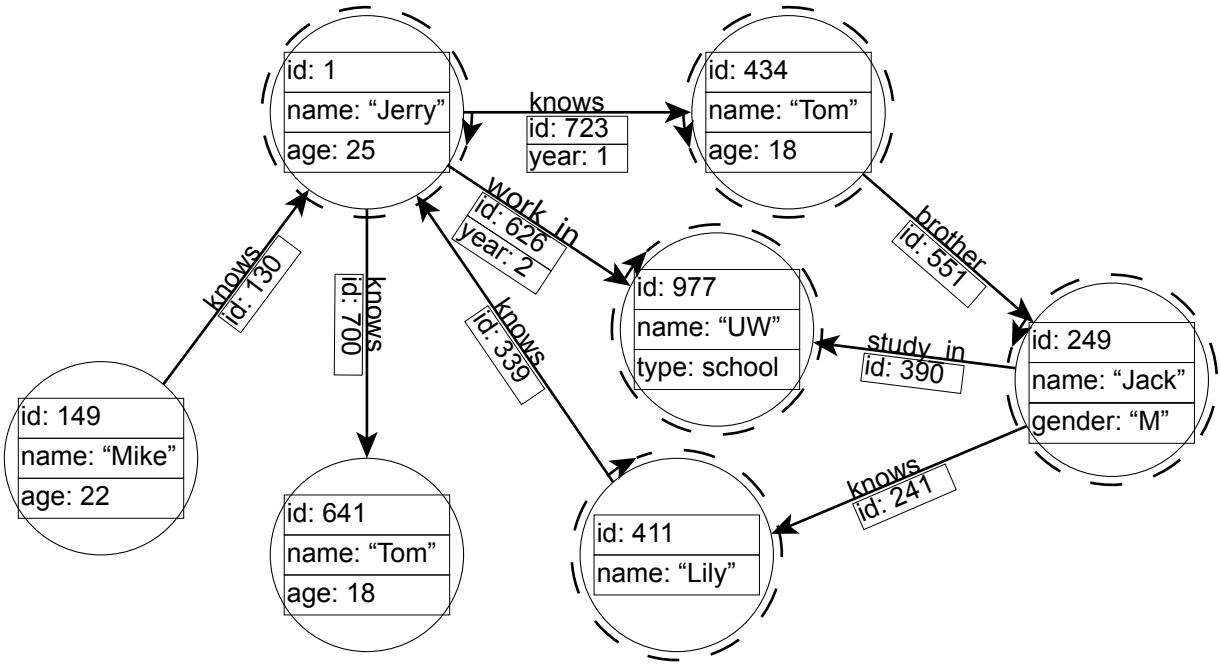

Figure 4.1: An example property graph.

For simplicity, we will denote a node with its “id” property prefixed with the character “N”. Similarly, for relationships we prefix the relationship “id” with the character “R”. For the graph in Figure 4.1, N1 knows N434 and N641, in which N1 has known N434 for one year (note the “year” property with value 1. Likewise, N149 and N411 know N1, N249 knows N411, N1 works in N977 for at least two years, N249 studies in N977, and N434 is the brother of N249.

In this property graph, there are two types of nodes: the person node, and the school node. Properties for nodes include id, name, age, gender and type. Each node can have an arbitrary number and types of properties. These are represented using a list of key-value pairs. The id property of the node represents the node id. Thus, node N1 has id 1, and it is also the first node in the node graph entries. There are four types of relationships, knows, brother, work_in, and study_in. Properties for relationships are year and id. Similar to the node, the id property of the relationship is the same as the relationship id.

4.1.1 Memory layout

Unlike conventional prefetchers, the hardware prefetchers for graphs use information about the data that they prefetch to make predictions. They use this data to discover the next data to prefetch. Consequently, it is important the memory layout of nodes and edges.

We depict the layout for node and relationship entries in Figure 4.2. Note that we use indexes instead of pointer addresses to describe the connections; however, in our implementation we use the addresses of the associated objects. For a node, the fields shown in a single entry include the VertexId, NextEdge and NextProp as described in the background chapter with Figure 2.8. The VertexId represents the node id, the NextEdge is a pointer to the first relationship, and NextProp is a pointer to the first property in the property list. For the relationship, the fields shown in a single entry include EdgeId, FirstVertex, SecondVertex, FirstNextEdge, FirstPrevEdge, SecondNextEdge, SecondPrevEdge and NextProp as described with in Figure 2.9. Once again, the EdgeId is the identifier for the relationship, and NextProp is a pointer to the first property in the property list for that edge. Recall that property lists are key-value paired lists. FirstVertex, and SecondVertex are pointers to the start (first) and end (second) nodes/vertexes. The FirstNextEdge and FirstPrevEdge are pointers to edges from the perspective of the FirstVertex. The same applies to the SecondNextEdge and SecondPrevEdge for the SecondVertex. We limit our discussion to nodes and edges because our hardware prefetchers currently does not exploit any prefetching potential for properties. Similarly, we omit the label field in the node file and relationship entries, because they are not currently used in our prefetcher; although, future incarnations of the prefetcher may be able to leverage them.

To illustrate the relation between the memory layout and the graph, we give an example using N1, R723 and N434 using the example in Figure 4.1. In the graph, N1 knows N434 through R723. In terms of node entries, the first relationship pointers for N1 and N434 point to R723. We can see this from Figure 4.2 where N1's and N434's NextEdge field refers to the relationship R723. In terms of relationship entries, R723 connects N1 to N434. The relationship entry encodes this using the following fields: FirstVertex, SecondVertex, FirstNextEdge, FirstPrevEdge, SecondNextEdge, SecondPrevEdge and NextProp. The FirstVertex field of R723 points to N1, and SecondVertex field of R723 pointers to the destination node N434. This is because the direction of R723 is from N1 to N434 resulting in the FirstVertex being N1 and the SecondVertex being N434. For a relationship, the fields that point to the next edges are based on the perspective of either the FirstVertex or SecondVertex. This means that the FirstNextEdge is the next edge after R723 from the perspective of N1, and the SecondNextEdge is the next edge from the perspective of N434. In Figure 4.2, a value of -1 denotes that the entry is not set. N1 has a clockwise traversal direction, and N434 has a counter-clockwise traversal direction. These directions are important in discovering the next relationships of the current node. For example,

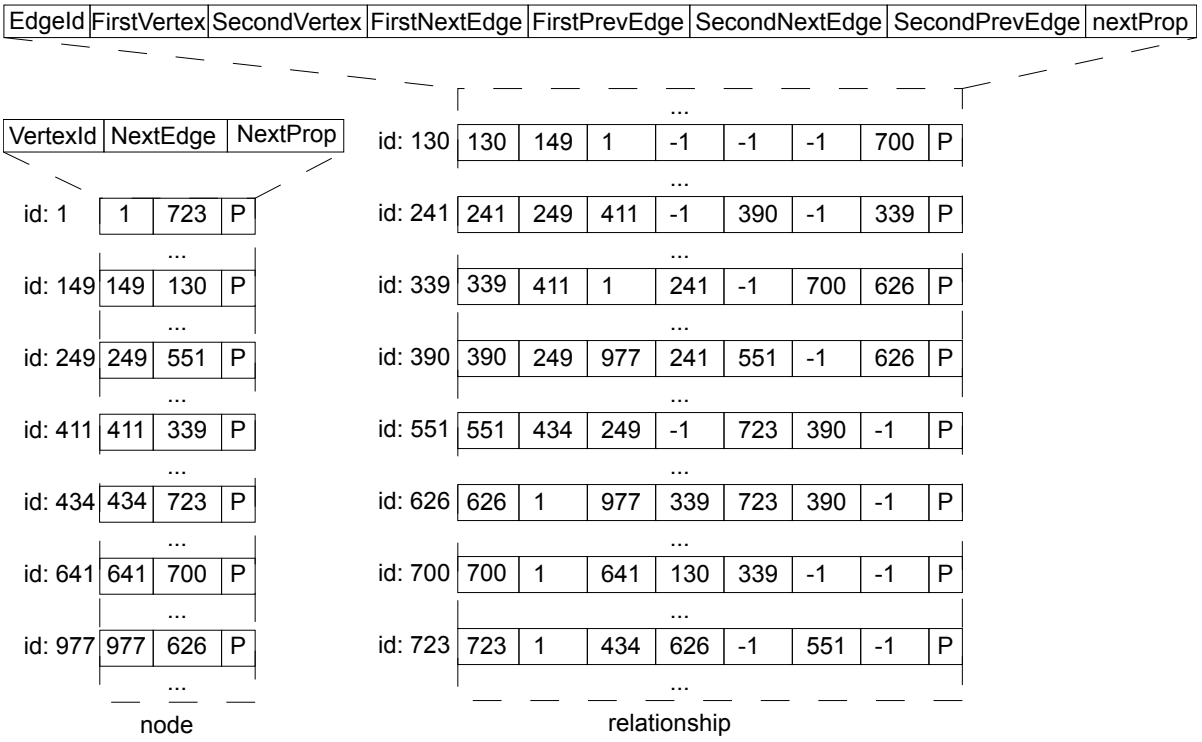

Figure 4.2: Node and relationship entries memory layout.

R723's FirstNextEdge points to R626, and FirstPreviousEdge is -1. This is correct because the next edge from the perspective of N1 is R626, and there is no previous edge. For N434, the SecondNextEdge points to R552, and SecondPreviousEdge is -1. This is also correct because the counter-clockwise direction for N434 means that the next edge from the perspective of N434 is R551. These directions are not pre-defined. They are determined during the process of populating the graph.

4.1.2 Sample query and access analysis

We use the example in Figure 4.1 to demonstrate the traversal when a query is performed. We will use a simple query that does a traditional breadth-first traversal as shown in Figure 4.3 on the graph. The objective of the query is to discover the names of all N1's immediate (1-hop) neighbours. This is an example of finding ones immediate friends in a social network.

Lines 1 and 2 initialize the two data structures used in the traversal, and the first node is pushed into the VertexQueue The traversal starts with line 3 of the while loop. The first access

```
1  root.visited = true;
2  VertexQueue.push_back(root);
3  while(!VertexQueue.empty()){
4    Vertex v = VertexQueue.front();
5    VertexQueue.pop_front();
6    while(v.hasNextChild()){
7      Vertex child = v.nextChild();
8      checkProperty(child);
9    }
10 }
```

Figure 4.3: The breadth-first traversal.

is N1 with the assumption that N1 is in the front of the queue (inserted as root). Line 4 removes the vertex from the VertexQueue, and line 5 checks whether there is a child node for the current node. In the case of N1, it does have children nodes. Line 6 takes the first child according to its NextEdge field, and line 7 checks the property of the node accordingly. Then the while loop continues until it reaches the last neighbor of N1. Hence, the access pattern resulting from this traversal interleaves accesses to node, relationship and property accesses. Since the traversal uses breadth-first, all accesses are on N1's neighbors, and there is no access to N249. Accesses to relationships are in the following sequence: R723, R626, R339, R700 and R130. The total access pattern for this breadth-first search instruction by instruction is shown in Table 4.1. The P-N in the table stands for the access to properties of the node.

Line	Memory accesses
4	N1
5-7 1st iteration	R723 N434 P-N434
5-7 2nd iteration	R626 N977 P-N977
5-7 3rd iteration	R339 N411 P-N411
5-7 4th iteration	R700 N641 P-N641
5-7 5th iteration	R130 N149 P-N149

Table 4.1: Graph entry accesses in the sample query.

This simple breadth-first traversal shows little locality in the access pattern. There is only some locality inside each node, relationship and property object. For example in line 7, the processor loads the whole name of a neighbouring node in order to do a property match. If the name of a node is larger than the size of a word, then the processor needs to launch several loads in order to retrieve all the contents in the "name". Of all the loads launched by the processor, only the first load is involved in migrating the cache line of property in the memory hierarchy.

The following loads get the property data for free because of the locality. However, no such locality exists between node accesses. Unlike the previous case where data is aligned closely together, node and relationship entries are distributed in memory. Since each single node or relationship object is larger than a cache line, one access to a member field does not yield any "free data" for the later accesses. We believe that as GDBs continue to receive read and write requests, the memory placement of the entries may change resulting in fragmentation in memory. Furthermore, there is no guarantee that entries spatially close are actually connected in the graph. Hence, we believe that breadth-first traversal on large graphs will render poor spatial locality as illustrated with this example.

4.2 Graph-locality Prefetchers

The graph-locality prefetcher (GLP) architecture makes use of buffer containers. These are containers that hold node and relationship data. Buffer controllers manage the movement of data in and out of the buffers. The buffer containers are currently fully-associative caches. In principle, graph-locality prefetchers are similar to conventional cache prefetchers in that GLPs can also be used at multiple levels in the hierarchy. Therefore, we can have level-1 GLP, level-2 GLP, and so on. Note that graph data (nodes and relationships) that do not reside in the GLP buffers end up being retrieved in the cache. This means that a miss in the GLP forces the cache hierarchy to retrieve the graph data. In this work, we focus on prefetching nodes and relationships. Hence, we introduce buffers for nodes and relationships. We do not address properties at the moment.

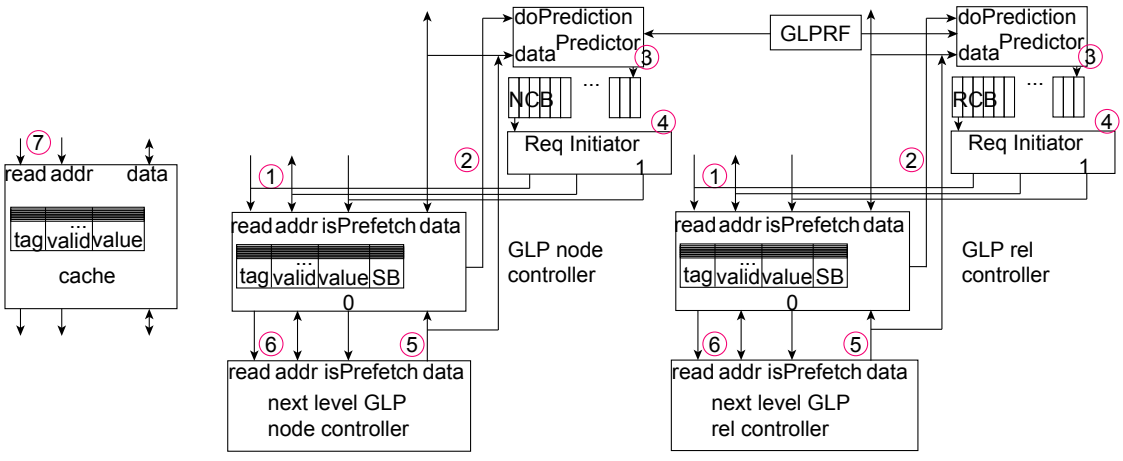

Figure 4.4: Graph-locality prefetcher architecture.

4.2.1 Graph-locality Prefetcher Architecture

We present the overall GLP architecture in Figure 4.4. The major components of the GLP include registers, a predictor, a candidate buffer, and a request initiator. The registers, their sizes and their purpose is listed in Table 4.2.

Register Number	Size	Purpose
RGLP1	8B	Stores the start node address.
RGLP2	2B	Offset register to retrieve the address of the FirstVertex in a relationship entry.
RGLP3	2B	Offset register to index the SecondVertex in a relationship entry.
RGLP4	2B	Offset register to index the FirstPreviousEdge in a relationship entry.
RGLP5	2B	Offset register to index the SecondPreviousEdge on a relationship.
RGLP6	2B	Offset register to index the FirstNextEdge in a relationship entry.
RGLP7	2B	Offset register to index the SecondNextEdge in a relationship entry.

Table 4.2: Registers in GLP.

The GLP needs a starting address to begin its prefetching and prediction. This starting address is typically the address of the start node of the query. We extend the ISA to introduce three special instructions.

The first instruction is a move instruction that moves contents from a general purpose register to a GLP register. For example, setting the start node register employs this instruction. However, it can also be used to set offsets for other offset registers. The second instruction notifies the prefetcher that the start node register has valid data, and the GLP may begin prefetching. This initiates the prefetcher to make its predictions and start with prefetching. Since GLPs are specific to graph traversals, we must disable its use when workloads are not performing queries on the GDBs. This is where the third special instruction informs the GLP to terminate the prefetching. This is beneficial because it will reduce unnecessary traffic on the memory interconnect.

The other registers listed in Table 4.2 are offset registers. These offset registers indicate the distance from the starting address of a either a node or relationship object, and the respective field that is of interest. For example, the offset register RGLP2 contains the distance in bytes from the base address of the relationship object. This allows the GLP to inspect the relationship object and retrieve the contents correctly. Note that these are GLP registers because it is possible that different implementations of the GDB may have varying offsets. These can be configured via special instructions dedicated to send information to the GLP.

GLP controller

The GLP controller orchestrates all components of the GLP. We describe the controller's operation using the circled numbers in Figure 4.4. There are several signals that the GLP controller manages in order to execute the policy These signals are **read**, **fill**, **doPrediction**, and **isPrefetch**. Please note that the GLP at each level in the hierarchy works in the same way. We describe the operation of the GLP using Figure 4.4. The numbers in the figure correspond to the explanation of its operation as listed next.

1. When the core performs a load on any memory access, the **read** signal to the GLP is asserted, and the address is provided on the address bus. The GLP checks whether the address exists in the GLP buffers by doing a tag match, and updates the corresponding age information for the replacement policy.

2. If the address is found in the GLP, this is a hit in the GLP. The **doPrediction** signal is asserted with the value stored in the status bit (SB).

3. If the SB is 0, the predictor does not make any predictions; however, if the SB is 1, then the predictor will make predictions based on the prefetch policy. When SB is 1, the predictor will use the data that resulted in a hit to discover additional addresses to insert in the node and/or relationship candidate buffer. After supplying the SB to the **doPrediction** signal, the SB for the corresponding data is cleared.

4. On every memory access, the request initiator retrieves data from the addresses inserted in the candidate buffers. The number of prefetches launched depends on the degree for the prefetcher. If the candidate buffers are empty, then no prefetches are performed. If there are candidates in both node and relationship candidate buffers, then prefetches are initiated from both the candidate buffers. These initiated prefetches result in **read**s to the lower-level GLPs, and in doing so we assert the **isPrefetch** signal to 1. This is to inform the lower-level GLPs that this request if from the prefetching, and not a demand-fetch from the core. This is important because when **isPrefetch** is 1, the lower-level predictor does not make predictions. This means that **doPrediction** for the lower-level GLP is set to 0 irrespective of the SB for the requested data. However, if **isPrefetch** is 0, then the request is coming from the core; therefore, allow the predictor for the lower-level GLP to perform predictions if it is a hit in this level, and the SB is 1.

5. Upon receiving a response for the prefetch requests from either the lower-level GLPs or the main memory, the new data fills an entry in the buffer and the SB is set to 1.

6. A miss on the GLP occurs when the address provided on a **read** does not match the tags in the buffer. This initiates a fill request from the lower-level GLPs by asserting the **fill** signal. When the data corresponding to the address is found in the lower-level GLP, it is brought in to the current level. Notice that we do not change the SB. Also note that multiple levels of the GLP buffers are inclusive.

7. If there is a miss in all levels of the GLP, then this will result in requesting the data through the cache hierarchy.

Predictor

The predictor is the core module of the prefetcher that implements the prefetch policy. The prefetch policy determines the data to be prefetched. This is done by adding the addresses associated with the data to be prefetched into the candidate buffer. The candidate buffer is a first-in-first-out (FIFO) buffer of fixed size to hold addresses. Notice that there is a candidate buffer for nodes and a separate one for relationships. These addresses are the addresses from where to prefetch data. The request initiator is responsible for transmitting prefetch signals as loads to the next level of the memory hierarchy with the addresses from the candidate buffer.

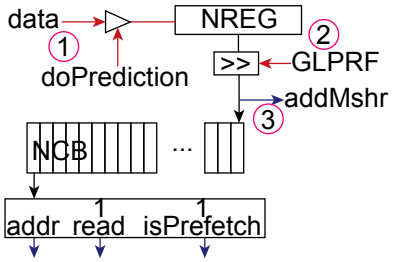

Figure 4.5: Prediction hardware for nodes.

The predictor design has logic for nodes as well as for relationships. We describe the node controllers operation using the circled numbers as shown in Figure 4.5. The numbers in the figure correspond to the explanation of its operation as listed next.

1. The node predictor only operate when the doPrediction signal is enabled by the controller, and the requested address was a hit in either the node or relationship buffer. If it is a hit in the node buffer, we insert the data that was found in the node buffer into register NREG.

2. We strip the NextEdge bytes from the register by using the offset register, which we insert into the node candidate buffer (NCB).

3. if the node predictor is doing a distance of 2 prefetch, also sends the NextEdge fields using addMshr to the MSHR.

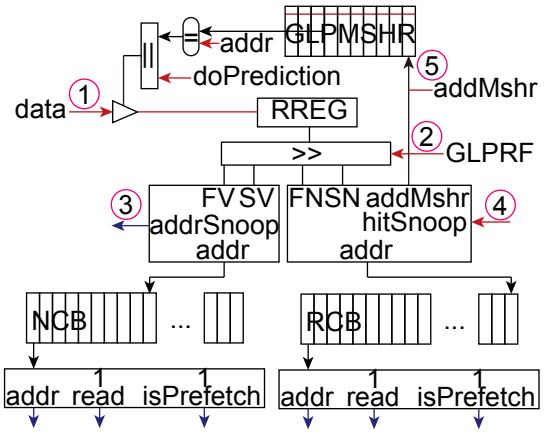

Figure 4.6: Prediction hardware for relationships.

Figure 4.6 shows the hardware for predicting relationships.

1. Once again, the relationship predictor only operate when doPrediction signal is enabled. The data for the hit is stored in RREG, which includes only four fields from the relationship object.

2. The four fields are splitted using input from GLPRF. The four fields include FirstVertex (FV), First-NextEdge (FN), SecondVertex (SV), and SecondNextEdge (SN) fields.

3. We use addrSnoop to send the FV address for a tag match to the node buffer.

4. If this is a hit, then hitSnoop is enabled, and we insert FN into the RCB. At the same time, we insert SV into NCB.

5. If the distance of the GLP is 2, then we insert FN into the GLPMSHR. This is based on the speculation that the direction of the breadth-first traversal will require the next relationship, from the perspective of the node denoted by FirstVertex, and once a relationship is accessed, its target node is always accessed.

We repeat this by sending the SV on addrSnoop, and if there is a hit, then we insert the SN into the RCB and FV into NCB. This completes the description of the prediction policy and its corresponding logic.

The miss status holding register (GLPMSHR) allows the GLP to wait for prior prefetch requests to complete before performing a prediction that uses prefetched data. This is essential for graph data prefetching because the addresses are contained within the data itself. We use the GLPMSHR for prefetches that have a distance of 2. A GLPMSHR entry has two fields, a indicator field and an edge field. The indicator field gives an indication to the predictor to prefetch according to the traversal from either the start or the end node. If the field is NULL, the predictor prefetches according to both nodes.

There are different logic when for node predictor and relationship predictor to insert into GLPMSHR. According to the previous section, node predictor inserts the NextEdge field into the GLPMSHR. In doing so, Node predictor also insert the address of current node in NREG. This is the same when relationship predictor inserts FN or SN fields. In this way, FN and FV are inserted together, SN and SV are inserted together. With the additional address on the vertex, the relationship predictor can pick direction to prefetch from its indicator field. For example, if the vertex matches with the FV field, the relationship predictor prefetches prefetches only FN and SV. This avoids additional traffic on the other direction.

4.2.2 Prefetching parameters

All of the proposed prefetchers are customizable in two ways: in their distance, and their degree. The prefetch distance for the proposed prefetchers is the number of hops to traverse from the current node or relationship before settling on either a node or relationship to prefetch from. For example, a distance of two for a node means that we first insert the address of the NextEdge relationship into the candidate buffer. Once the relationship is retrieved, its next edges (First and Second) are added into the candidate buffers.

The prefetch degree is a parameter used to select the number of addresses from the candidate buffers to prefetch the data from. For example, a prefetch degree of 2 means that the top two candidates in the candidate buffers are marked as addresses for prefetching. The request initiator would take these two addresses and initiate prefetches.

4.2.3 Prefetch policies

In order to remedy the poor locality in graph queries, we developed different prefetch policies that attempt to leverage graph locality. We explore three different graph-locality policies: GLP1-simple, GLP2-aggressive, and GLP3-advanced. Note that for each of these we also vary the degree, which amounts to the number of consecutive prefetch candidates retrieved consecutively

from the candidate buffer. Note that for each of these policies, the hardware architecture remains the same, and only the policy changes.

Conventional prefetchers use the history of addresses accessed, and the hits on the data prefetched to predict future prefetches. We believe that effective prefetchers for graphs must be fundamentally different in that they must prefetch based on the data that is retrieved. This is unlike conventional prefetchers that do not peek into the data that is retrieved. Hence, GLP1 and GLP2 are preliminary versions of prefetchers that inspect the data being prefetched to determine the data to be prefetched in the future.

All graph-locality prefetch policies begin by using the address in the start node register as an initial address. Recall that we introduce two special instructions that engage the GLPs during queries on the GDBs. The first instruction is placed at the beginning of the traversal to update the start node register, and to signal the GLP to initiate its prefetching logic. The second instruction informs the GLP to terminate prefetching, which is placed at the end of the query.

Graph-locality prefetching policies do not simply rely on the addresses provided by the core requests. Instead, they must inspect the data being prefetched to determine future candidates for prefetching. Graph-locality prefetcher 1 (GLP1-simple) is our first attempt at exploiting the breadth-first traversal for effective prefetching. Our intuition is that if we can accurately predict the breadth-first traversal via prefetching, then whenever the query performs the traversal the requested data will result in hits in the GLP buffers. Notice that the data used for prefetching is different for nodes, and for relationships. For example, when the start node is accessed, the entire node object is retrieved into the GLP buffers. The node object only contains three fields: the VertexId, and the NextEdge, and NextProp addresses. The graph-locality prefetching policy inspects the node object arriving into the GLP buffer, and inserts the address of the NextEdge into the relationship candidate buffer. This means that a potential candidate for prefetching would now contain the relationship connected to the start node. Note that this is not spatial locality, but instead it is using the structure of the graph to prefetch data.

Suppose that instead of the node a relationship is accessed. The relationship consists of multiple addresses in its object such as the FirstVertex, SecondVertex, FirstNextEdge, Second-NextEdge, FirstPreviousEdge, and SecondPreviousEdge. The prefetch policy adds the relevant addresses from the relationship object into the relationship candidate buffers. However, our objective is to follow the breadth-first traversal; therefore, we only insert a subset of these addresses into their candidate buffers. To determine which of these to insert in the candidate buffers, we have the following two cases:

1. For the accessed relationship, if the FirstVertex points to a node that is already in the GLP buffer, then we insert the FirstNextEdge of the FirstVertex node into the relationship

candidate buffer. Then we use the accessed relationship's SecondVertex address to add it into the node candidate buffer. We determine these candidates based on the speculation that the FirstVertex is the node that the breadth-first traversal is currently processing. As a result, the likelihood of accessing next relationship exiting the FirstVertex is considered alongside that its target (the SecondVertex) will also be important.

2. The second case is the same as the first except that the SecondVertex points to a node that is found in the GLP buffer. Then the addresses for the SecondNextEdge and the FirstVertex are added to their respective candidate buffers.

The above explanation assumes that the distance parameter for the GLP is set to 1. However, in the event that the distance is larger than 1, the GLPs perform the same behaviour except they perform multiple hops. Using the graph-locality prefetching policy described above, we derive three GLPs that we call GLP1-simple, GLP2-aggressive, and GLP3-advanced. The parameters for each of the GLPs, and their unique properties are described next.

- GLP1-simple: The distance is set to 1. This is the simplest graph-locality prefetcher that we explore.
- GLP2-aggressive: The distance is arbitrary, but for empirical evaluation we set this to a value of 2. This graph-locality prefetcher follows an aggressive policy by performing multiple hops before determining the candidates addresses.
- GLP3-advanced: The distance is also arbitrary much like GLP2-aggressive, but we set it to 2 for our evaluation. Additionally, we use the first of the special instructions to update the start node register on every node access. This initiates a prediction based on the node address in the start node address.

We admit that there are other variations of graph-locality prefetchers that we can develop, but we reserve that as a part of our future exploration.

Chapter 5

Results

We use a micro-architectural simulator called Snipersim [8] to develop the proposed hardware prefetchers, and compare against conventional prefetchers. We follow Intel's Xeon Gainestown architecture with the processor frequency of 2.66 GHz, and we currently only focus on single core setups. The core model is an in-order core with one-way dispatch width. This configuration has three levels in its cache hierarchy. Each cache has a 64 byte cache line, and employs least-recently used replacement policy. We have the following four classes of configurations.

- Cache only. This configuration uses a three-level cache hierarchy. No prefetchers are enabled.
- Graph-locality Prefetchers (GLP)s. All GLP configurations include the cache hierarchy and a three-level GLP hierarchy. We assume that all levels use the same prefetch policy, and we experiment with the three policies we develop. Overall, we maintain the same capacity of storage buffers as in the cache only configuration. This means we take capacity away from the caches, and assign it to the GLP buffers; thereby, ensuring that the total capacity used for storage remains the same across all experiments.
- Global History Buffer (GHB). This configuration uses the cache hierarchy, but adds the global history buffer prefetcher.
- Stream prefetcher (STR). This configuration augments the cache hierarchy with a stream prefetcher. This is added in all three levels of the memories.

Table 5.1 present the cache parameters for the cache only, and the cache with GHB and STR prefetchers. Table 5.2 shows the parameters for the caches with the prefetchers as well. The sizes of the caches and buffers for GLPs are shown in Table 5.3.

Cache	Size	Associativity	Relationship to cores
L1 I-cache	32KB	4 way	private
L1 D-cache	64KB	8 way	private
L2 Cache	512KB	8 way	private
L3 Cache	8MB	16 way	shared

Table 5.1: Cache parameters for the cache only, and cache with GHB and STR prefetchers.

Configurations	L1	L2	L3
Cache only	L1-D cache, L1-I cache	L2 cache	L3 cache
Cache + STR	L1 + STR	L2 + STR	L3 + STR
Cache + GHB	L1 + GHB	L2 + GHB	L3 + GHB
Cache + GLP1-simple	L1 + GLP	L2 + GLP	L3 + GLP
Cache + GLP2-aggressive	L1 + GLP	L2 + GLP	L3 + GLP
Cache + GLP3-advanced	L1 + GLP	L2 + GLP	L3 + GLP

Table 5.2: Cache and prefetchers for the configurations.

We also present the GLP buffer configurations Table 5.4. The baseline prefetcher has both prefetch degree and distance equal to 1. The outstanding prefetching candidate buffer for each type of store files holds 30 entries. The L1 GLP buffer holds entry size of 115 for node and relationship entries, and L2 holds 935 for each. The L3 buffer holds 1870 entries and is shared. In the cache and buffer integrated architectures, both prefetchers or buffers can be deactivated.

Cache/Buffer	Size	Associativity	Relationship to cores
L1 I-cache	32KB	4 way	private
L1 D-cache	32KB	8 way	private
L2 Cache	256KB	8 way	private
L3 Cache	8MB	16 way	shared

Table 5.3: Cache and GLP buffer parameters for GLP configurations.

GLP Buffer	N/R buffer size	N/R candidate size	Total size	Relationship to cores
L1 Buffer	115/115	30/30	32KB	private
L2 Buffer	935/935	30/30	256KB	private
L3 Buffer	1870/1870	30/30	512KB	shared

Table 5.4: Buffer configurations.

5.1 GDBench

We deploy queries from a graph database benchmark called GDBench [9] to evaluate the performance of the hardware prefetchers. The data schema of this benchmark is shown in Figure 5.1. This data schema includes two entities: persons and web pages. For each of these it has its own node type. There are also two types of relationships: friend and like. The friend relationship exists between person to person, and like relationship exists between person and web page. A person can be identified by an identifier pid, name, age, and the his/her location. These are the person node properties. The properties of a webpage include a webpage identifier wpid, URL, and the creation date of the webpage. In the properties, the name, the age, the location and the creation date are optional fields.

The GDBench data set uses a data schema in Figure 5.1. The figure is reproduced from GDBench. There are two types of nodes, the person and the web page. Friend relationships link nodes together and webpage relationships link person to web pages together. A person node has four fields, pid, name, age and location, in which age and location are optional fields. A webpage node has three fields, wpid, url and date.

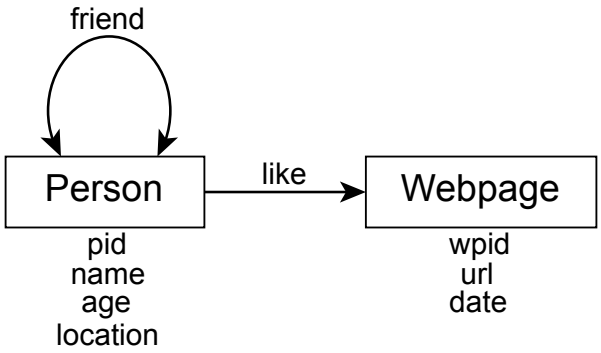

Figure 5.1: Data schema of GDBench.

The authors of GDBench provide their source code for generating graph data. There are two options for the generation of the graphs: normal distribution and power-law distribution. We noticed that the generator provided by GDBench organizes its data in an ordered manner. For instance, the nodes are sequentially generated and produced in the data input file. Loading such an input file into our graph database shows that benefits from spatial locality are good. This is because the graph has not evolved over time with updates and changes. Hence the position of the node and edges are in somewhat contiguous blocks of memory. We believe that in a long-running graph database, this would not be the case, and nodes and edges may be distributed in the

memory. We simulate this effect by re-organizing the data input file for nodes and relationships in a random fashion.

The queries associated with GDBench represent what may be considered typical workloads on social network queries. All the queries descriptions and types are in Table 5.5. The are a total of 13 queries, and each query checks for random nodes at runtime. Based on the characteristics of the workloads, we classify them into five query types: select, adjacency, reachability, pattern matching, and summarization. We also add an additional query (14) to perform a traditional breadth-first traversal on the graph. Performance of the traversal algorithm is important because breadth-first traversals are the de-facto standard pattern of traversing the graph in all the queries.

Q	Description	Type
1	Get all the persons having a name N.	Select
2	Get all the persons who like a given webpage W.	Adjacency
3	Get the webpages that person P likes.	Adjacency
4	Get the name of the person with a given PID.	Select
5	Get the friends of the friends of a given person P.	Reachability
6	Get the webpages liked by the friends of a given person P.	Reachability
7	Get persons that like a webpage which a person P likes.	Reachability
8	Is there a connection (path) between persons P1 and P2.	Reachability
9	Get the shortest path between persons P1 and P2.	Reachability
10	Get the common friends between persons P1 and P2.	Pattern matching
11	Get the common webpages that persons P1 and P2 likes.	Pattern matching
12	Get the number of friends of a person P.	Summarization
13	Breadth-first traversal.	Algorithm

Table 5.5: The queries of GDBench with their description and classification.

To run the benchmark, we load the graph data resulting from GDBench's generator into our graph database. This is the initialization phase of the GDB. This includes creating nodes and relationships in the GDB, and also associating them with their properties. Then, we execute the queries in sequence. When the queries finish, we generate reports with the statistics and terminate the simulation run. In order to use SniperSim, we augment the benchmark application with magic instructions. These magic instructions allow us to provide the start node address, and the terminate signal to the prefetchers. Additionally, we insert the region of interest markers, so as to ensure that the simulator records the execution sequence of queries only, and ignores other stages such as initialization. This is because our focus is on improving the performance of queries.

5.1.1 Query groupings

We implemented the queries using our GDB for the GDBench benchmark suite. We group these queries according to their implementation, which is dependent on the visitors used. This is important because the visitor determines the amount of work and graph accesses done. The queries from GDBench are grouped as follows.

- SelectionVisitor: Q1, Q4, Q13. These queries employ the standard breadth-first traversal. Q1 and Q4 traverse the graph until a certain property is found. Q13 traverses the entire graph in a breadth-first fashion.
- AdjacencyVisitor: Q2, Q3, Q12. These queries only check for properties within 1 hop distance. For example, Q2 checks the number of people that like a particular website.
- ReachabilityVisitor: Q5, Q6, Q7. These queries traverse the graph for a hop distance of 2. For example, Q5 retrieves the names of friends of friends of a given person.
- PathVisitor: Q8, Q9. These queries collect and return the path from one node to another. For example, Q8 finds a path between two persons.
- PatternVisitor: Q10, Q11. These queries return the names of persons that have a certain pattern between them. For example, Q10 retrieves the names of common friends between two persons.

5.2 Experimental results

We launch two groups of experiments on the architecture with a conventional memory hierarchy without prefetchers, and another with prefetchers (both conventional ones and the ones we propose). We use GDGenerator [9] to generate the data set for execution. The parameters for generation are 1,000 nodes, 50% of nodes are people, and 50% of nodes are webpages. The friend and the like relationships follow the power-law distribution. All experiments execute the same graph queries. All queries run starting with a cold cache, which simulates the big-data platform where queries may not be related to each other. We expect that when cores are dedicated to a subset of graph databases, that there would be residual data from prior accesses that later could potentially leverage. GDBench uses a random number generator for picking the target for queries. For each single query, we use the breadth-first traversal to complete the search. Thus, there are in total 13 queries.

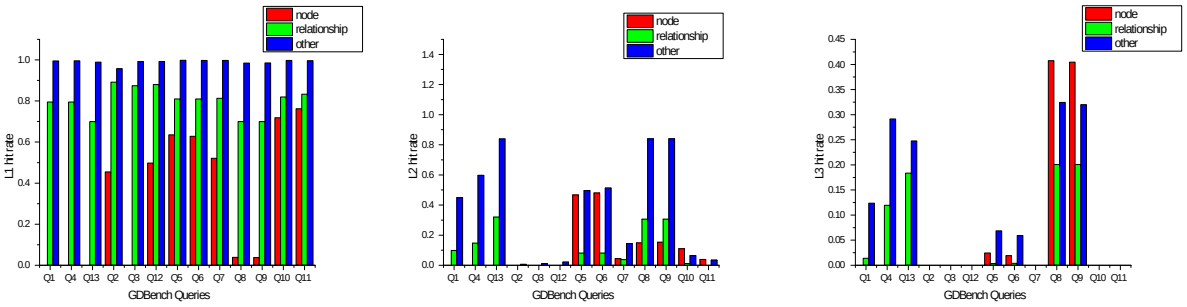

Figure 5.2: Cache hit rate comparison for different levels.

5.2.1 Cache only results

We present Figure 5.2 that shows the hit rates for data corresponding to nodes, relationships and all other data at each level of cache in the memory hierarchy. It can be observed that the hit rate for NR data is lower than that of other accesses in the L1 cache. This means that whenever the core makes accesses to NR data, it results in the stalling of the processor pipeline. Therefore, queries that have a large number of proportion of NR accesses endure larger performance degradation. To establish this, we present Figure 5.3 that shows the access rate for the L1 cache split into node, relationship and other accesses. This figure shows that for Q1, Q4, Q13, Q3, Q12, Q8 and Q9 have a larger proportion of NR accesses (higher than 4%). Furthermore, Figure 5.5 and Figure 5.6 show that the NR access rates in L2 and L3 are large. This occurs only if they are misses in the L1 cache. Hence, we would conclude that these queries will perform worse than the others such as Q2, Q5, Q6, Q7, Q10, and Q11. Using Figure 5.4, we identify that all queries other than Q2 show an instruction-per cycle (IPC) performance that is higher than Q1, Q4, Q13, Q3, Q12, Q8 and Q9. The reason for Q2 to be the anomaly is simply that this query makes very few accesses to all data (NR and other). Therefore, the latency is dominated by cold misses in the cache.

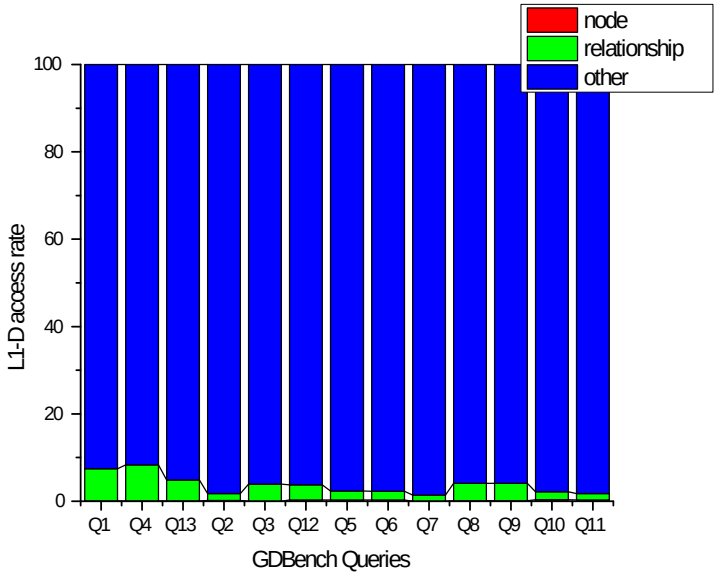

Figure 5.3: Cache only L1 performance.

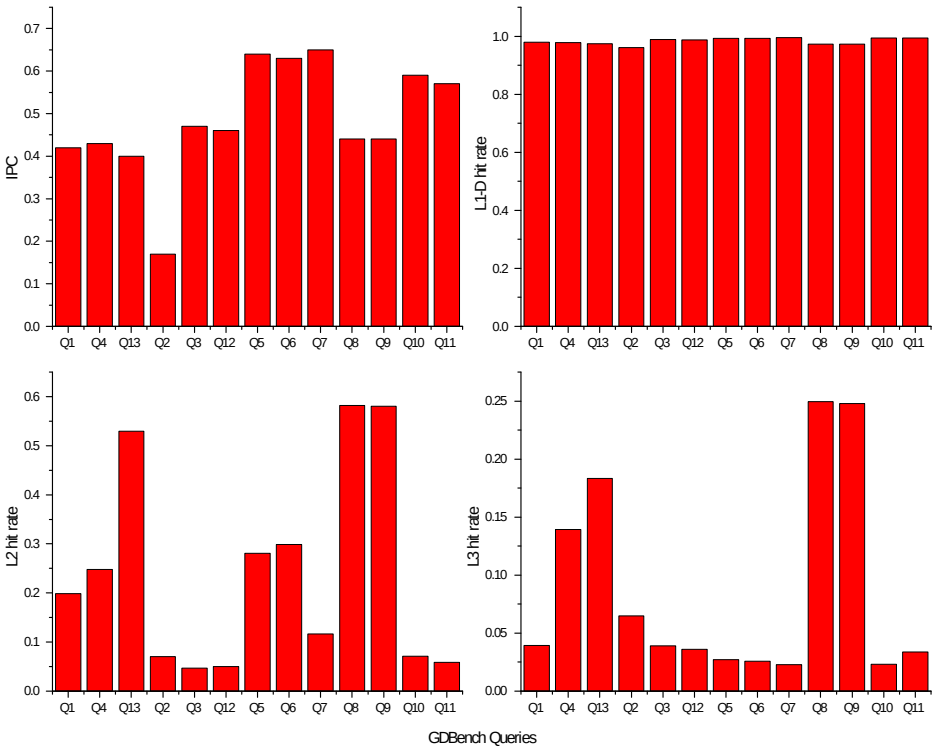

Figure 5.4: Cache only performance.

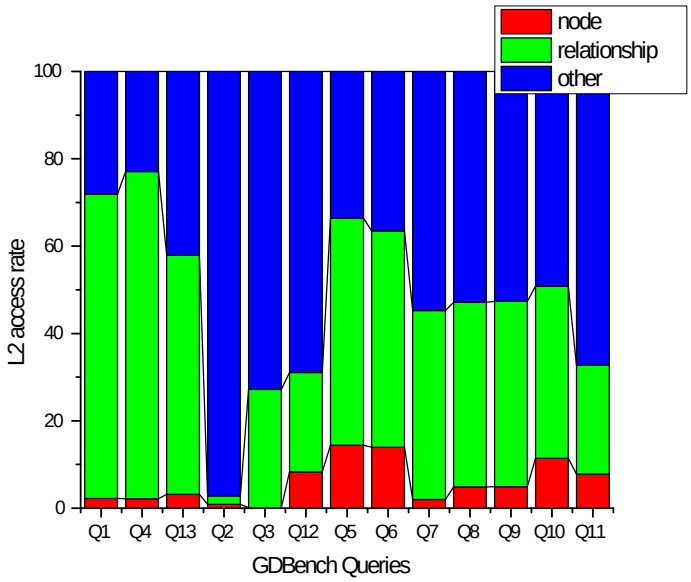

Figure 5.5: Cache only L2 performance.

5.2.2 GLP1-simple results

Figure 5.7 shows the performance of the cache when GLP1-simple was added. We add two variations of GLP1-simple with degree 1 and degree 2. We observe that by adding GLP1-simple, the performance of the cache improves. This is certainly the case for GLP1-simple with degree 2, but not for all queries with GLP1-simple with degree 1. This is because degree 1 does not bring in a large number of NR accesses into the GLP buffers. Hence, a significant portion of them continue to be fetched into the caches. We can see this from Figure 5.8 and Figure 5.9 where GLP1-simple with degree 1 has higher number of node and relationship accesses going to the cache as opposed to GLP1-simple with degree 2. Further, Figure 5.10 shows that even though the hit rate for GLP1-simple with degree 1 is higher than degree to in L1 buffer, the number of accesses to NR is significantly larger. Moreover, since we retain the overall capacity of the memory at each level, the cache sizes are also smaller when GLP1-simple is added. However, the IPC shows that by adding GLP1-simple to the memory hierarchy, the performance improved with the lowest improvement being 6% in Q8 and Q9 (ignoring the light-weighted Q2, which performance stays the same) and the largest being 38% in Q4.

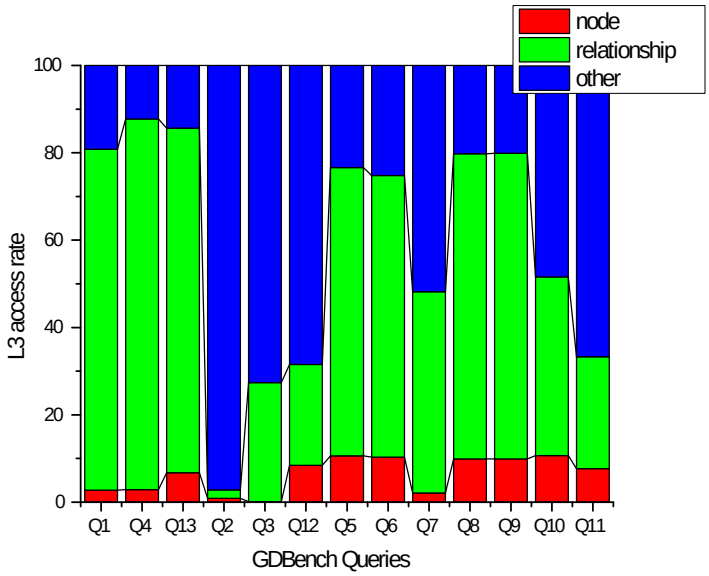

Figure 5.6: Cache only L3 performance.

5.2.3 GLP2-aggressive, GLP3-advanced, and speedup comparisons

Figure 5.11 shows that the IPC when using GLP3-advanced performs better than GLP2-aggressive, and both of these do better than GLP1-simple. This holds for both degrees 1 and 2. As before, for the same GLP degree 2 outperforms degree 1, which is because the buffer access ratio increases as the degree increases as shown in the figure. However, GLP1-simple with degree 1 experiences performance degradation in Q13, Q8 and Q9. This is because GLP1-simple with degree 1 is not capturing a sufficiently large number of NR accesses, and the cache sizes are smaller. This results in degraded performance.

Figure 5.11 also shows that the buffer access ratio for GLP3-advanced degree 2 is lower than the other degree 2 GLPs. This is because in order to implement GLP3-advanced, we added magic instructions to pass the node address for every node, which result in additional memory instructions. This causes there to be more number of memory accesses in total for that query; however, the number of buffer accesses remain the same. This is an artifact of the manner in which magic instructions are implemented. If we were to implement instructions in the ISA to perform these operations, we would not suffer this reduction in buffer access ratio.

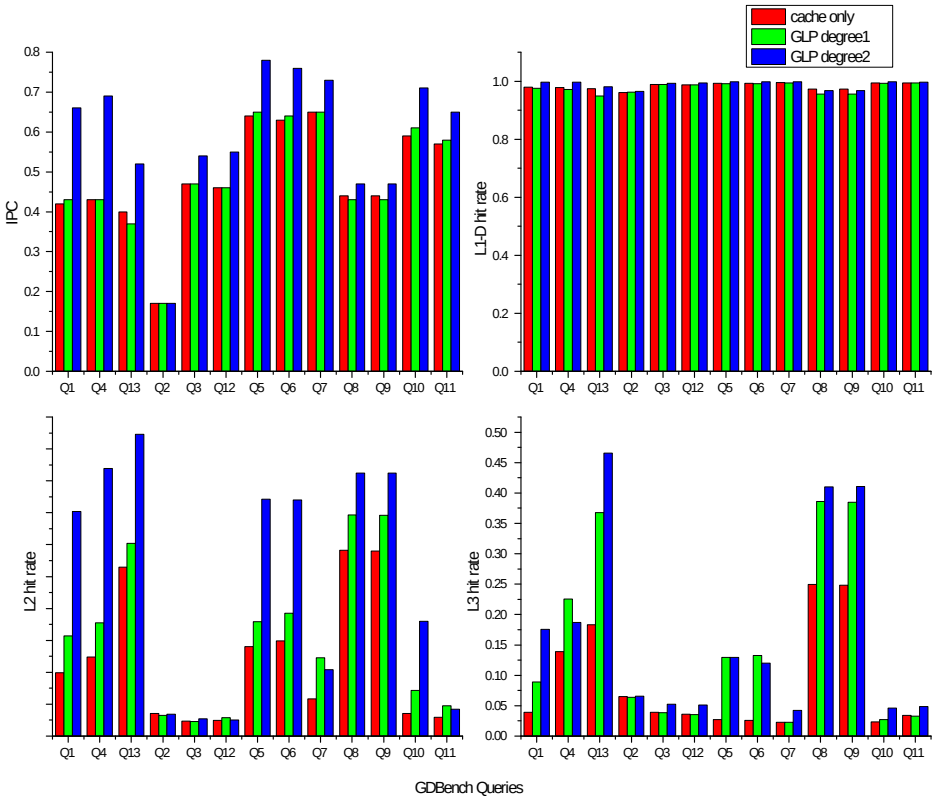

Figure 5.7: GLP1-simple performance with degree 1 and 2 compared with cache only.

5.2.4 Comparison with traditional prefetchers

Figure 5.12 shows an important limitation of traditional prefetchers. This graph shows that even when increasing the degree for both GHB and STR, the speedup is not significantly larger. Hence, this shows that traditional prefetchers are unable to capture the graph-locality effectively because graph-locality have irregular access patterns that can only be observed by inspecting the data. Nonetheless, traditional prefetchers show an improvement in performance over not using any prefetchers. We observed that for stream prefetcher, the lowest IPC speed up being 6% in Q8 and Q9, and highest being 27% in Q4. On average, the performance improvement is 13%. For GHB prefetcher, the lowest IPC speed up being 4% in Q8 and Q9, and highest being 26% in Q1 and Q4. The average performance improvement for GHB is 10%.

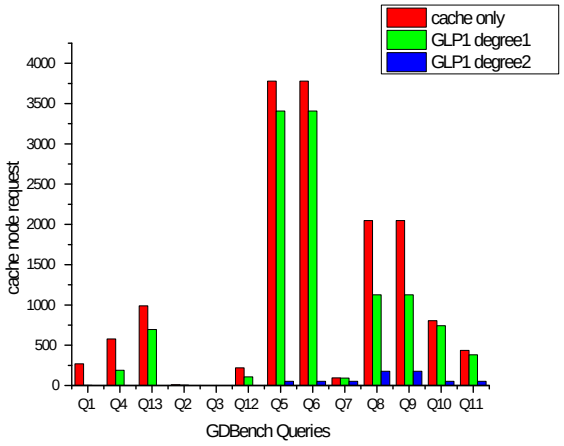

Figure 5.8: Nodes request comparison.

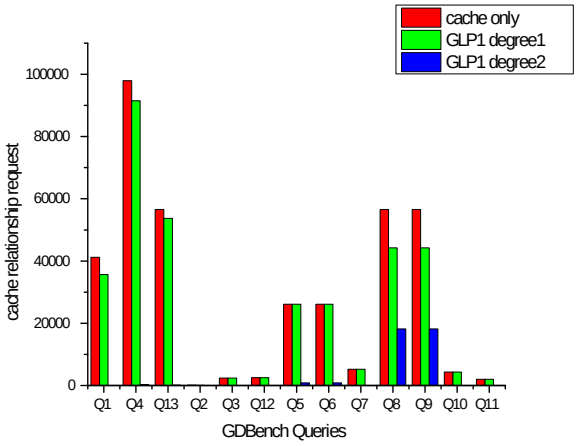

Figure 5.9: Relationship request comparison.

5.2.5 Comparison of GLPs against traditional prefetchers

Figure 5.13 shows that GLPs are resulting in improved performance than traditional prefetchers. This is with the exception of Q2, Q3, and Q12. The reason for this is because these queries employ the AdjacencyVisitor, which explores only a single-hop distance in the graph. Furthermore, the graph does not have nodes that have a large number of neighbours one hop away. As a result, the number of accesses to graph data is low.

Another reason that STR performs better in Q2 and Q3 is that the stream prefetcher results in hits with non-NR accesses as shown in Figure 5.14. This is confirmed by observing a close to 100% hit rate in GLP3 L1 buffers. Note that L1 non-NR hit rate for STR and GLP3 are the same, and L3 GLP3 non-NR hit rate is 0 across all queries. For Q12, GLP does as well as STR because the nodes have larger number of one-hop neighbours than the nodes in Q2 and Q3.

Aside from Q2, Q3, and Q12, the remaining queries yield a minimum improvement of 5% in Q7 and Q11, and a maximum improvement of 25% in Q8 when compared to STR, and a minimum of 2% in Q3 and maximum of 31% in Q4 over GHB.

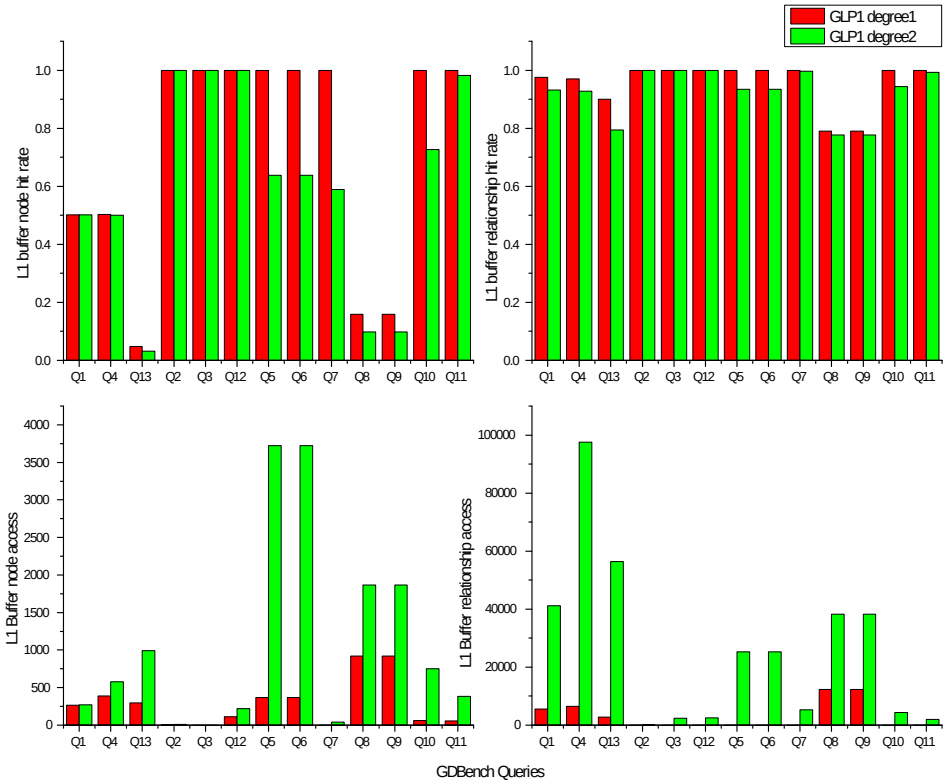

Figure 5.10: GLP1-simple performance for node and relationship.

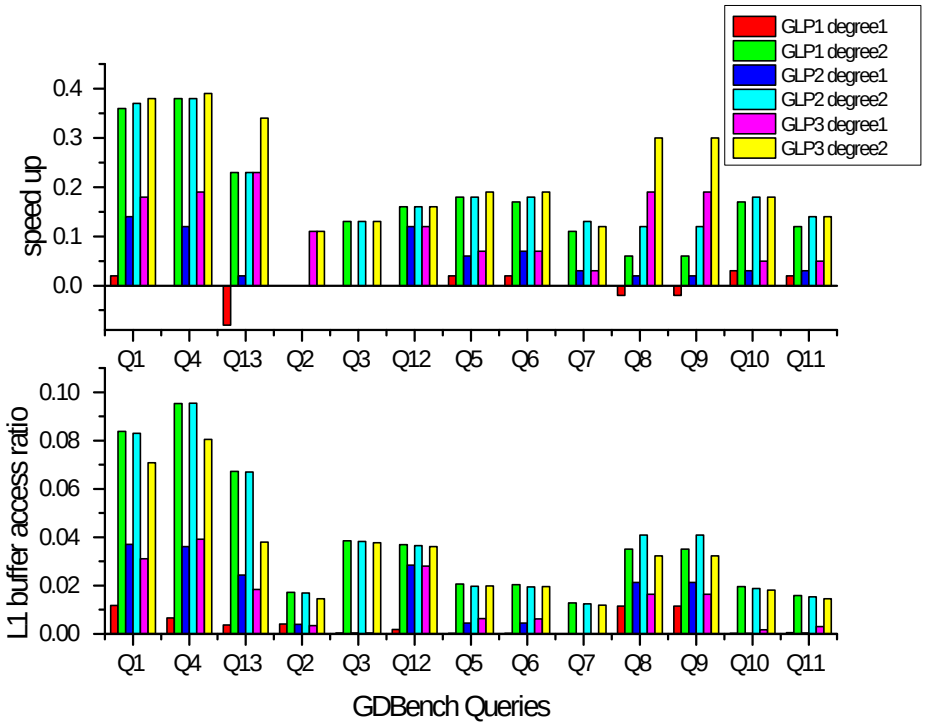

Figure 5.11: Buffer access ratio and speed up comparison.

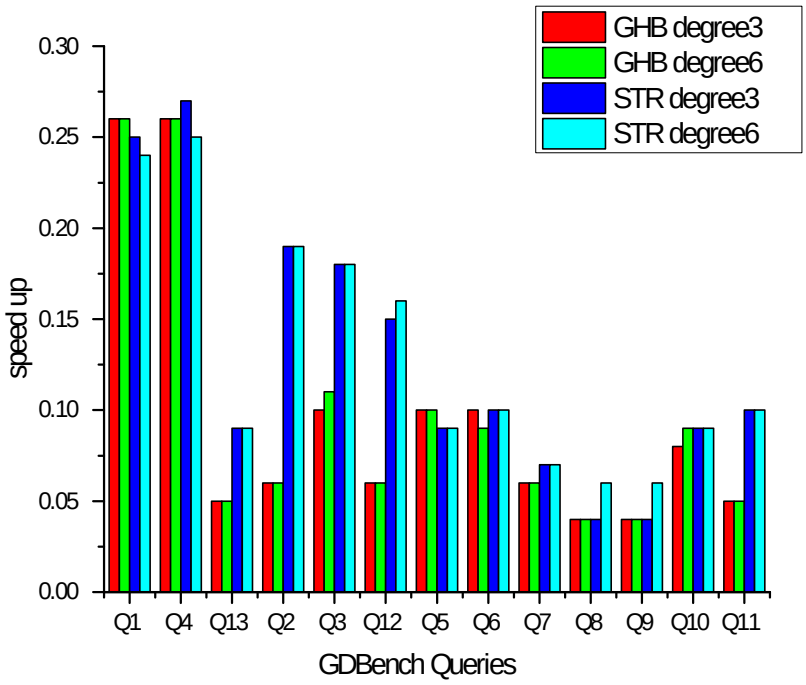

Figure 5.12: Speed up comparison of traditional prefetchers.

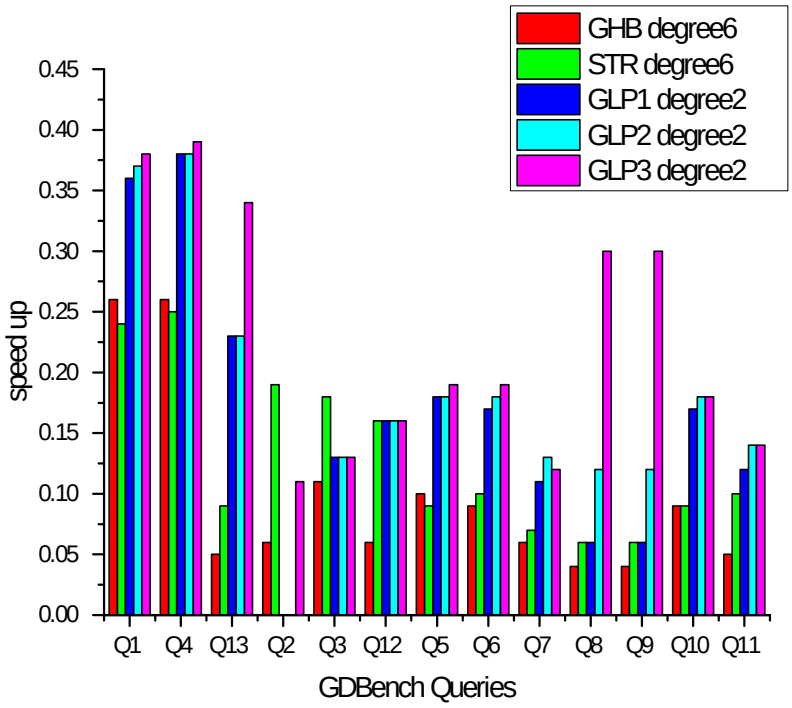

Figure 5.13: Speed up comparison of GLPs with traditional prefetchers.

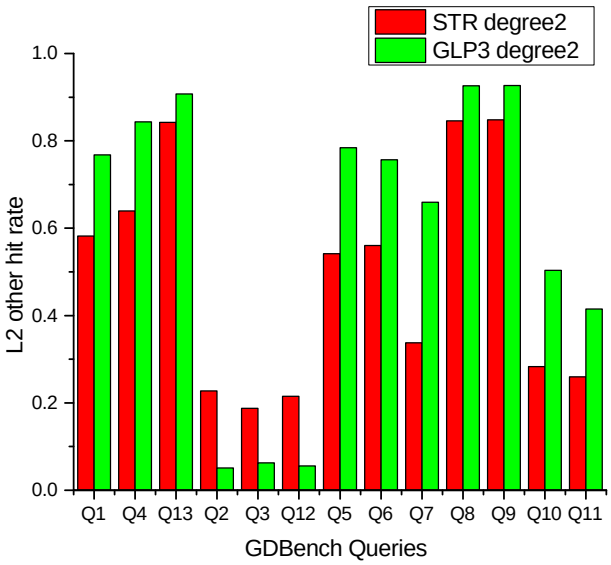

Figure 5.14: L2 hit rate comparison of GLP3 with STR.

Chapter 6

Conclusion and Future Work

In this thesis, we proposed hardware prefetchers specially designed to accelerate memory accesses for queries on graph databases. These prefetchers exploited graph-locality during breadth-first traversals. The key insight these prefetchers used was to inspect the data in order to make predictions of what data to prefetch next. In doing this, we presented three variations of our graph-locality prefetchers. Additionally, we experimented with degree 1 and degree 2 for prefetching. We compared with a traditional cache hierarchy with no prefetchers, and also one with GHB and STR prefetchers. Our results showed that GLP3-advanced performs between 11% and 39% better than the traditional cache hierarchy. In addition, the GLP3-advanced outperformed both GHB and STR prefetchers with improvements ranging from 2% to 31%. However, we noticed that this requires a degree of 2 for it to be effective.

While this thesis presents hardware prefetchers that effectively exploit graph-locality, we believe that there are several opportunities for additional future work. In particular, there are three pieces of future work that we feel would be of interest for the community. First, we feel that evaluating with GDBench, although useful, does not represent realistic workloads on GDBs. Hence, we suggest evaluating the benefits of GLPs with the Linked Data Benchmark Council (LDBC) [31] GDB benchmark that is designed for graph data management. LDBC aims to create industrial strength benchmarks targeting social networks using GDBs. LDBC includes three separate workloads, which includes interactive workload, business intelligence workload and graph analytics workload. The three workloads cover a variety of database management system use cases from industry. Further, the data generated by LDBC mimics characteristics of a real social network such as Facebook, which can stress the memory subsystem. There are pragmatic concerns when using LDBC that must be addressed. For instance, the simulation time for running LDBC benchmarks may be significantly large. Second, we currently assume that there is only one thread executing a graph traversal workload. In practice, there will be multiple

threads performing queries on either the same or different graphs. This means that we need to investigate extending the GLPs to support multiple threads. Third, we believe that a hardware prototype on an FPGA will provide for accurate performance estimates. It will also force one to consider the area and resource impact on different designs carefully.

References

[1] "Titan." https://thinkaurelius.github.io/titan/.

[2] "Orientdb." http://www.orientechnologies.com/.

[3] "Neo4j." http://neo4j.com/.

[4] "Infinitegraph." http://www.objectivity.com/infinitegraph#.

[5] "flockdb." https://github.com/twitter/flockdb.

[6] "bitsy." https://bitbucket.org/lambdazen/bitsy/wiki/Home.

[7] "Trinity." http://research.microsoft.com/en-us/projects/trinity/.

[8] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pp. 91–94, High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012.

[9] R. Angles, A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey, "Benchmarking database systems for social network applications," in *First International Workshop on Graph Data Management Experiences and Systems (GRADES)*, (New York, NY, USA), pp. 15:1–15:7, ACM, 2013.

[10] N. Wallis, "Big data in canada: Challenging complacency for competitive advantage," tech. rep., International Data Corporation (IDC), 12 2012.

[11] "Big data revenue." http://wikibon.org/wiki/v/2012_Big_Data_Revenue_by_Vendor.

[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "A case for specialized processors for scale-out workloads," *IEEE Micro*, vol. 34, no. 3, pp. 31–42, 2014.

[13] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "Noc-out: Microarchitecting a scale-out processor," in *Proceedings of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, (Washington, DC, USA), pp. 177–187, IEEE Computer Society, 2012.

[14] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*, (Washington, DC, USA), pp. 500–511, IEEE Computer Society, 2012.

[15] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, (New York, NY, USA), pp. 249–260, ACM, 2013.

[16] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Hardware partitioning for big data analytics," *IEEE Micro*, vol. 34, no. 3, pp. 109–119, 2014.

[17] W. S. Song, J. Kepner, V. Gleyzer, H. T. Nguyen, and J. I. Kramer, "Novel graph processor architecture," *Lincoln Laboratory Journal*, vol. 20, no. 1, pp. 92–104, 2013.

[18] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, (Los Alamitos, CA, USA), pp. 12:1–12:10, IEEE Computer Society Press, 2012.

[19] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "Dbmss on a modern processor: Where does time go?," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 266–277, Morgan Kaufmann Publishers Inc., 1999.

[20] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 54–65, Morgan Kaufmann Publishers Inc., 1999.

[21] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: A data provenance perspective," in *Proceedings of*

Annual Southeast Regional Conference (SE), (New York, NY, USA), pp. 42:1–42:6, ACM, 2010.

[22] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A performance evaluation of open source graph databases," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications (PPAA)*, (New York, NY, USA), pp. 11–18, ACM, 2014.

[23] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[24] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Transactions on Programming Languages and Systems*, vol. 31, pp. 20:1–20:39, Aug. 2009.

[25] C. Caval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proceedings of Annual International Conference on Supercomputing (ICS)*, (New York, NY, USA), pp. 150–159, ACM, 2003.

[26] L. Yuan, C. Ding, D. Å tefankovic, and Y. Zhang, "Modeling the locality in graph traversals," in *Proceedings of International Conference on Parallel Processing (ICPP)*, (Washington, DC, USA), pp. 138–147, IEEE Computer Society, 2012.

[27] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, pp. 174–199, June 2000.

[28] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn’t, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 2:1–2:29, Mar. 2012.

[29] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), pp. 279–290, ACM, 2002.

[30] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "Prefedge: Ssd prefetcher for large-scale graph traversal," in *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, (New York, NY, USA), pp. 4:1–4:12, ACM, 2014.

[31] P. Boncz, "Ldbc: Benchmarks for graph and rdf data management," in *Proceedings of the 17th International Database Engineering & Applications Symposium*, IDEAS '13, (New York, NY, USA), pp. 1–2, ACM, 2013.