IMP: Indirect Memory Prefetcher

Xiangyao Yu†, Christopher J. Hughes‡, Nadathur Satish‡, Srinivas Devadas†

† Massachusetts Institute of Technology
‡ Parallel Computing Lab, Intel Labs
{yxy, devadas}@mit.edu, {christopher.j.hughes, nadathur.rajagopalan.satish}@intel.com

ABSTRACT

Machine learning, graph analytics and sparse linear algebra-based applications are dominated by irregular memory accesses resulting from following edges in a graph or non-zero elements in a sparse matrix. These accesses have little temporal or spatial locality, and thus incur long memory stalls and large bandwidth requirements. A traditional streaming or striding prefetcher cannot capture these irregular access patterns.

A majority of these irregular accesses come from indirect patterns of the form $A[B[i]]$. We propose an efficient hardware indirect memory prefetcher (IMP) to capture this access pattern and hide latency. We also propose a partial cacheline accessing mechanism for these prefetches to reduce the network and DRAM bandwidth pressure from the lack of spatial locality.

Evaluated on 7 applications, IMP shows 56% speedup on average (up to 2.3×) compared to a baseline 64 core system with streaming prefetchers. This is within 23% of an idealized system. With partial cacheline accessing, we see another 9.4% speedup on average (up to 46.6%).

1. INTRODUCTION

Operations on sparse data structures, such as sparse matrices, are important in a variety of emerging workloads in the areas of machine learning, graph operations and statistical analysis as well as sparse solvers used in High-Performance Computing [10, 42]. Many important classes of algorithms, including machine learning problems (e.g., regression, classification using Support Vector Machines, and recommender systems), graph algorithms (e.g., the Graph500 benchmark and pagerank for computing ranks of webpages), as well as HPC applications (e.g., the HPCG benchmark) share similar computational and memory access patterns to that of Sparse Matrix Vector Multiplication, where the vector could be sparse or dense [5].

MICRO-48, December 05-09, 2015, Waikiki, HI, USA
© 2015 ACM. 978-1-4503-4034-2/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830807

A key bottleneck in sparse matrix operations is irregular memory access patterns and working sets that do not fit into a conventional first-level cache. This leads to accesses with high latency, which degrades performance. Furthermore, the high memory intensity in these workloads exerts bandwidth pressure in both the network on-chip (NoC) and DRAM.

The trend of multicore systems exacerbates this problem. Most multicore processors include shared structures in the memory hierarchy that are physically distributed, such as a last-level cache or directory. As the number of cores scales up, the average latency to such a structure increases. Meanwhile, multicore systems tend to use simpler cores that are less capable of hiding latency beyond the first-level data cache.

Existing latency tolerance mechanisms are insufficient to handle latency incurred by irregular memory accesses. *Out-of-Order* (OoO) execution [39] and simultaneous multithreading [40] help alleviate the problem but cannot hide all of the latency. Conventional hardware prefetchers [6] can capture streaming or strided access patterns but not irregular ones. Other techniques either require expensive hardware (e.g., runahead execution [30]) or require programming model changes (e.g., helper threads [14, 22]) and have not yet had impact in the multicore environment.

Several works, e.g., [25, 27], have proposed hiding memory latency through the programmer or compiler inserting software prefetching instructions. Mowry [26] proposed a compiler solution to prefetch indirect patterns. However, software prefetching has inherent limitations: it may not be portable across microarchitectures (e.g., with different cache hierarchies) and can incur large instruction overhead, consuming power and offsetting the gains from improved cache hit rates. Furthermore, software prefetching is not able to capture patterns that are only exposed at runtime.

In this paper, we take a different approach to address this problem. We propose hardware targeting a specific, common memory access pattern in these workloads. In particular, we observe that an overwhelming fraction of codes operating on sparse datasets rely on indirect memory accesses in the form of $A[B[i]]$, and that the $B[i]$ values are pre-computed and stored consecutively in memory [9]. For example, for graph codes,

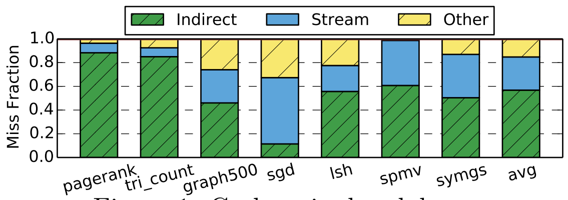


Figure 1: Cache miss breakdown.

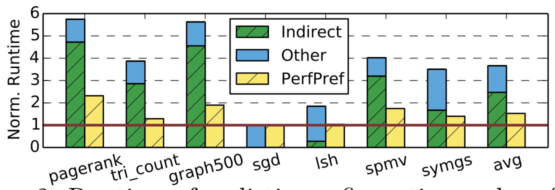


Figure 2: Runtime of realistic configuration and perfect hardware prefetcher. Indirect indicates the portion of execution time from stalls on indirect accesses.

the A may be the vertices, and B the set of edges out of a given vertex. For sparse matrix codes, A may be a dense vector (e.g., for matrix-vector multiplication), and B the set of column indices with non-zero values in a row of the matrix. If we iterate over a set of these values, a conventional hardware streaming prefetcher can capture the accesses to $B[i]$. Our approach is to detect the indirect accesses on top of the streaming pattern, and link prefetching of $B[i]$ to prefetching of $A[B[i]]$.

We propose the *Indirect Memory Prefetcher* (IMP), a cheaply realizable hardware mechanism to capture indirect memory accesses. For a set of codes operating on sparse data structures, it is able to capture the majority of the performance benefits that accrue from hiding memory access latency. Specifically, IMP is able to cover 85% of first level cache misses and achieve 56% average (up to 2.3×) speedup over the applications we evaluated. This is within 23% of the performance of an idealized prefetcher.

We also observe that indirect accesses usually exhibit little spatial locality. That is, for an indirectly accessed cacheline, most of the data on the line remains untouched at the time the line is evicted. Thus, we propose to have IMP fetch partial cachelines. When applied to a core's private cache and the shared last-level cache, this can significantly reduce both NoC and DRAM traffic. With this optimization, performance is further improved by 9.4% on average (up to 46.6%) and NoC/DRAM traffic is reduced by 16.7%/7.5% on average (up to 39.3%/28.0%).

2. MOTIVATION

Irregular memory accesses are common in many applications, especially in machine learning, graph analytics, and sparse linear algebra. Accessing neighbor vertices in a graph or data elements corresponding to non-zeroes in a sparse matrix or vector is typically done through indirection, such as $A[B[i]]$.

Many programs pre-compute the indices, $B[i]$, and store them in an index array. Indices commonly represent the structure of a graph, sparse matrix, or other irregular data structure. Such structures are often static or change infrequently; thus, computations re-use the B, amortizing the overhead of creating the array.

Accesses to $B[i]$ are typically sequential and may be captured by streaming prefetchers. The $A[B[i]]$ accesses, however, tend to touch non-consecutive memory locations. In these applications, the size of A is large and usually does not fit in the first level cache; thus, the indirect accesses to A generate many cache misses.

In Fig. 1, we show the fraction of L1 cache misses caused by different access types on a 64 core system with a 32KB L1 data cache per core. On average, indirect accesses comprise 60% of the total cache misses. Thus, if memory behavior impacts performance, indirect accesses are responsible for much of it. Also, in all cases, indirect and streaming accesses together make up the majority of cache misses.

2.1 Latency Bottleneck

The high miss rate of indirect accesses can hurt performance a lot. Fig. 2 shows the execution time broken down into cycles attributed to cache misses from indirect accesses and everything else. The runtime is normalized to that of an idealized system where all memory accesses hit in the L1 cache. We describe the other result, *PerfPref*, in Section 2.2.

Indirect memory accesses are the main performance bottleneck in most of these applications. This is expected since they generate most of the misses, and these misses are not captured by streaming prefetchers.

2.2 Bandwidth Bottleneck

The *PerfPref* bars in Fig. 2 show runtime for a system with an idealized hardware prefetcher which prefetches *all* memory requests a certain amount of time before the data is accessed. Given infinite NoC and DRAM bandwidth, this will hide all memory access latency. However, this system is on average 1.8 times slower than *Ideal*, indicating performance is limited by the realistic bandwidth modeled for the system. Bandwidth usage is high due to poor temporal and spatial locality for the irregular memory accesses, as well as the large number of cores. Unless we reduce the amount of data that the cores request, *PerfPref* is a performance upper bound for hardware prefetching. Even with realistic bandwidth limits, the difference between the pairs of bars shows we have lots of room for performance improvement by hiding latency.

3. INDIRECT MEMORY PREFETCHER

In this section, we present the *Indirect Memory Prefetcher* (IMP) that captures indirect memory accesses.

3.1 Key Idea

Indirect memory accesses in the applications we study involve two data structures: an index array B and a data array A. These codes share a pattern when generating indirect accesses: they scan a portion of their index array, reading each index and the corresponding element in the data array. That is, for some contiguous values of i, they access $B[i]$ then $A[B[i]]$. The address of $A[B[i]]$ depends on the value of $B[i]$ (Eq. (1)).

$$ADDR(\ A[B[i]]\ ) = \mathit{Coeff} \times B[i] + \mathit{BaseAddr} \qquad (1)$$

Coeff is the size of each element in A and *BaseAddr* is the address of $A[0]$. They are both constant for a specific indirect pattern. Since the values stored in B may not (and typically don't) follow a recognizable pattern, the accesses to $A[B[i]]$ are not predictable and a traditional hardware prefetcher cannot capture it.

Our key insight comprises two parts. (1) The unpredictable factor is the contents of B. If hardware has access to $B[i]$ early, it may predict *Coeff* and *BaseAddr* and use those to compute the address of $A[B[i]]$ before software accesses it. (2) In practice, $B[i]$ is often precomputed and stored in memory and can be read in advance.

IMP leverages this insight; it predicts which accesses are reads to an index array, and uses the contents of those memory locations to predict addresses of future indirect accesses. In particular, when software accesses $B[i]$, IMP will prefetch *and read the value of* $B[i + \Delta]$. It then uses predicted values of *Coeff* and *BaseAddr* to calculate the memory address of $A[B[i + \Delta]]$ following Eq. (1) and prefetches that line.

An indirect pattern is characterized by the index array and the corresponding values of *Coeff* and *BaseAddr*. To capture a pattern, IMP must learn all of these.

3.2 IMP Architecture

IMP is hardware attached to an L1 cache and snoops the access and miss stream of the cache. Fig. 3 shows the components of IMP, which work together to perform three steps.

First, it captures the stream pattern of the index array using a stream prefetcher. The *Stream Table*, part of the *Prefetch Table* (PT), is a traditional stream prefetcher working at word granularity.

Second, given an index stream, IMP detects the associated indirect pattern by computing *Coeff* and *BaseAddr*. Our approach to computing *Coeff* and *BaseAddr* is to find two index-address pairs, i.e., $(B[i],\ ADDR(\ A[B[i]]\ ))$, and then according to Eq. (1), solve for our two unknowns. Obtaining index-address pairs is non-trivial for hardware sitting at the cache; it has no visibility into the dataflow of the program, and so cannot know whether some later memory access uses the value of a previous access. IMP uses the *Indirect Pattern Detector* (*IPD*) to store candidate index-address pairs and compute *Coeff* and *BaseAddr*. Once IMP finds an indirect pattern, it stores it in the *Indirect Table*, in the PT.

Finally, IMP begins indirect prefetching, triggered by each index access. IMP's address generator uses information in the PT to create prefetches.

3.2.1 Address Generation

IMP's address generator and *IPD* both perform computations based on Eq. (1). For address generation, this

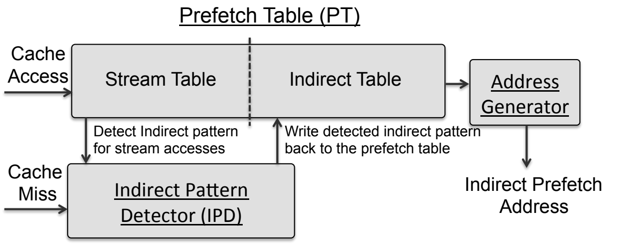

Prefetch Table (PT)
Cache Access
Stream Table
Indirect Table
Address Generator
Detect Indirect pattern for stream accesses
Write detected indirect pattern back to the prefetch table
Cache Miss
Indirect Pattern Detector (IPD)
Indirect Prefetch Address

Figure 3: The architecture of Indirect Memory Prefetcher.

requires a multiplication and an addition. For pattern detection, i.e., computing *Coeff* and *BaseAddr*, this requires a subtraction and a division. Multipliers and dividers are expensive hardware structures.

We restrict the possible values of *Coeff* to reduce hardware cost. In real workloads, *Coeff* is the size in bytes of an element of array A; thus, *Coeff* is often a small power of two. By only considering such values for *Coeff*, we can simplify Eq. (1) to Eq. (2), replacing the multiplication and division operations by shifts.

$$ADDR(\ A[B[i]]\ ) = (B[i] \ll \mathit{shift}) + \mathit{BaseAddr} \qquad (2)$$

With this optimization, address generation only needs a shifter and an adder. For the rest of the paper, we will use *shift* instead of *Coeff*.

3.2.2 Indirect Pattern Detection

As discussed in Section 2.2, a key challenge to compute *shift* and *BaseAddr* is to identify at least two $(B[i],$ $ADDR(A[B[i]]))$ pairs; the *IPD* solves this problem. The idea is to try candidate index-address pairs from the set of accesses that the cache sees, until we find a *shift* and *BaseAddr* that satisfies Eq. (2) for at least two of them.

We leverage three insights. First, we are concerned with detecting fairly long-lasting patterns; as long as we *eventually* detect a pattern, we can prefetch it effectively. Thus, it is fine to fail to recognize many index-address pairs. Second, we can narrow down candidate indices by only considering values captured by a stream prefetcher. Third, we can narrow down candidate indirect addresses for a given index by only considering cache misses soon after the index access.

Fig. 4 shows the *IPD*; it is organized as a table. Each entry is responsible for detecting one indirect pattern. On a candidate index access (i.e., anything detected as a streaming access), if the index stream has not been associated with an indirect pattern or a matching *IPD* entry, the *IPD* allocates an entry in the table and writes the index value, i.e., $B[i]$, to the *idx1* field.

For each cache miss following an index read, *IPD* pairs the miss address with *idx1* and, for each value of *shift* being considered, computes *BaseAddr* according to Eq. (2). *IPD* writes these *BaseAddrs* to the *baseaddr array* in the IPD entry. *IPD* only tracks the first few misses after the *idx1* access.

When *IPD* sees an access to the next index in that

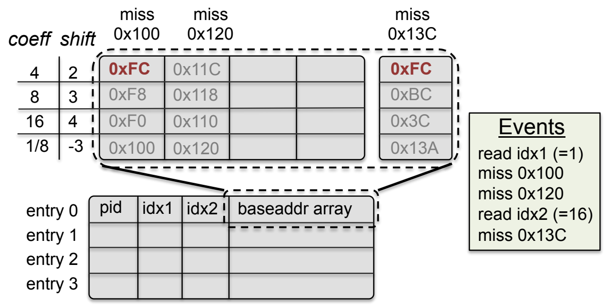
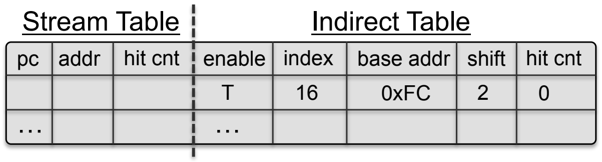


Figure 4: The Indirect Pattern Detector (IPD), and an example of indirect pattern detection (*shift*=2, *BaseAddr*=0xFC)



Stream Table			Indirect Table				
pc	addr	hit cnt	enable	index	base addr	shift	hit cnt
			T	16	0xFC	2	0
…			…				

Figure 5: Prefetch Table (PT). Stream table is the same as a traditional stream prefetcher.

stream (i.e., $B[i+1]$), it writes its value to *idx2*. *IPD* pairs later cache misses with *idx2* to compute *BaseAddrs*, as it did with *idx1*. It then compares these *BaseAddrs* with other *BaseAddrs* in the *baseaddr array* with the same *shift*. If there is a match, then *IPD* has found *shift* and *BaseAddr* values that satisfy the following set of equations:

$$\begin{cases} MissAddr1 = (B[i] \ll shift) + BaseAddr \\ MissAddr2 = (B[i+1] \ll shift) + BaseAddr \end{cases}$$

With reasonably high confidence, we can say that *MissAddr1* and *MissAddr2* are part of an indirect access pattern with *shift* and *BaseAddr* as the indirect parameters. *IPD* records these parameters in the *PT* for indirect prefetching. The *IPD* also activates the corresponding entry in the *PT* and releases the current *IPD* entry so that it can be used to detect other patterns.

If the third element is accessed in the index array (i.e., $B[i+2]$) but the indirect pattern is still not detected, then the pattern may not exist. In this case, *IPD* also releases the current entry. The index array can keep allocating *IPD* entries in the future until it eventually finds a pattern. After each failed detection, the index array waits for an exponentially increasing back-off time before the next detection to avoid thrashing the *IPD*.

3.2.3 Indirect Prefetching

Fig. 5 shows the contents of *PT*. A portion of each entry comprises the Stream Table. This tracks the index stream by storing the program counter (*pc*) of the instruction accessing the stream and the address of the most recently accessed index from the stream (*addr*). It also tracks the number of stream hits (*hit cnt*) which triggers stream prefetching when a threshold is reached.

The remaining portion of each *PT* entry comprises the Indirect Table, which tracks indirect accesses. This portion of each entry is inactive until the *IPD* enables it (i.e., sets *enable* = True) and stores the *shift* and *BaseAddr*. Before IMP starts prefetching, it needs to increase confidence that this pattern is worth prefetching.

When IMP sees an index access that matches the entry, it writes the index value to the *index* field. For every access thereafter, it checks if the address matches the expected indirect access address, according to *index*, *shift* and *BaseAddr*. If there is a match, it increments the saturating counter *hit cnt*. If IMP overwrites the index with a later index access before it finds a match, it decrements the counter.

Once the saturating counter (*hit cnt*) reaches a threshold, IMP starts indirect prefetching. On every index access that matches such a *PT* entry, IMP issues one or more prefetches for that pattern. The *prefetch distance*, the distance in the access stream between the current access and the one we prefetch, is initially small and linearly increased as more hits happen to the same pattern. Given a prefetch distance Δ, IMP reads the index element $B[i+\Delta]$ and computes the indirect prefetch address using Eq. (2). IMP places prefetched cachelines into the cache, but it could instead use a prefetch buffer.

As in a traditional stream prefetcher, IMP can load data in either *Shared* state or *Exclusive* state. It uses a simple read/write predictor to make this decision. IMP can also be used for SIMD instructions (e.g., scatter and gather) but the architecture needs to be adjusted accordingly.

Prefetchers can operate on either virtual or physical addresses. Operating on physical addresses eschews the need for address translation, but forces prefetching to stop at page boundaries. Operating on virtual addresses allows for long streams. In this paper, IMP generates addresses for both streaming and indirect prefetches in virtual space since indirect accesses are likely to touch different pages. As a consequence, IMP should be attached to a cache with address translation support, i.e., an L1 cache.

3.3 Optimization

We now present some optimizations for IMP.

3.3.1 Nested Loops

We must capture both the streaming and indirect patterns for indirect prefetching to work. Detection usually takes multiple loop iterations and prefetching does not happen during this learning phase. For short loops, the length of the learning phase limits IMP's performance gain (Listing 1).

Listing 1: Nested Loops

```
for (int i = 0; i < N; i++)
    for (int j = f(i); j < f(i+1); j ++)
        load A[B[j]]
```

Our solution to this problem, as already shown in Section 3.2.3 is to associate a stream pattern with the *PC* of the stream access. This is common in existing stream prefetchers [6].

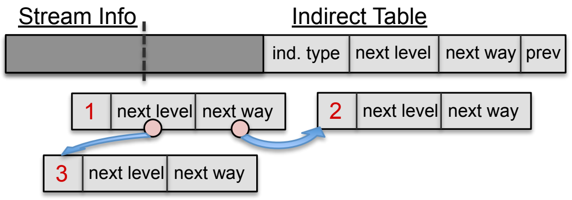


Figure 6: Fields added to the PT to support secondary indirections. The *ind type* can be primary (entry 1), second-way (entry 2) or second-level (entry 3).

When IMP detects an indirect pattern, it stores the PC of the index access in the PT. When the application completes an outer loop iteration and begins the next one, the streaming pattern to the index array is interrupted and possibly restarted at a different point, since the values of `j` see a hiccup. However, the index accesses come from the same instruction (PC) as before. Thus, rather than re-learning the pattern, IMP simply changes its position within the index array by updating the *addr* field in the stream table (Fig. 5).

3.3.2 Multi-way and Multi-level indirection

While many indirect accesses strictly follow the $A[B[i]]$ pattern, we also observe some variants of this pattern. These include *multi-way* and *multi-level* indirections as shown in the following code snippets. In multi-way indirection (Listing 2), two indirect patterns use the same index array. In multi-level indirection (Listing 3), a value read via an indirect access is an index for another data array. We call these additional uses of prefetched values *secondary indirections*.

Listing 2: Multi-way

```
for(i = 0;  i < N;  i++)
    load  A[B[i]]
    load  C[B[i]]
```

Listing 3: Multi-level

```
for(i = 0;  i < N;  i++)
    load  A[B[C[i]]]
```

We modify the structure of the PT to support these more complicated patterns. Specifically, we dedicate some PT entries to secondary indirections. We also add several fields to each entry, as shown in Fig. 6. The *ind type* indicates whether the entry is primary (default indirect pattern), multi-way or multi-level. The remaining fields link related PT entries in a tree, with a primary entry as the root. The *next way* and *next level* fields hold PT entry numbers of children, and the *prev* field indicates the parent entry.

When prefetching for a primary pattern is triggered, IMP traverses the tree of secondary patterns attached to it. IMP issues a prefetch for each second-way indirection immediately after prefetching for its parent, since all second-way indirections share the same index value with the parent. For each second-level indirection, IMP needs the index value accessed by the parent's prefetch and thus a second level prefetch can only be issued after the parent prefetch returns.

IMP detects secondary patterns similarly to primary patterns. After it detects a primary pattern, it tries to detect secondary patterns by allocating another entry in the *IPD*.

4. PARTIAL CACHELINE ACCESSING

Our target applications exhibit poor spatial locality in addition to poor temporal locality. Since their working sets are large and accesses are irregular, we expect to miss the first level cache with each indirect access. Further, each indirect access only consumes a portion of a cacheline, but forces the eviction of a full cacheline that (most likely) has similarly been only partly consumed. The result is that the hardware retrieves full cachelines, only to discard most of the data without using it. Since bandwidth is a serious performance bottleneck, as discussed in Section 2.2, this not only wastes power, but degrades performance.

Several previous works [45, 46, 18, 34] have proposed to address this issue using a sector cache and partial cacheline accessing. In these schemes, when a cache miss occurs, a predictor decides the granularity of the data loaded from lower in the memory hierarchy. For our applications, since the partially consumed lines are also indirectly accessed, we only consider partial accesses for indirect memory accesses and build the granularity prediction logic into IMP.

We consider partial cacheline accesses to both cache and DRAM.

4.1 Sector Cache

A cache supporting sub-cacheline granularity accesses is known as a sector cache [21]. Fig. 7 shows one cacheline in a sector cache. Each cacheline is split into multiple sectors, and each sector has a valid bit. During an access, if the cacheline is found, the valid bits need to be checked for the bytes requested. On a cacheline miss or a sector miss, the cache may retrieve a partial cacheline by requesting only some of the sectors. Only those sectors are sent through the memory hierarchy and stored into the cache.

Many commodity DRAM modules have a large minimum data transfer size, e.g., 32B or 64B. Since our cacheline size is 64B, this limits the potential of partial accesses to DRAM. We assume a DRAM access granularity of 32B, since at least one commercial processor has this feature [2]. Future DRAM technologies might allow for finer-grained accesses.

For the rest of this section, we assume all levels of cache are sectored and that we can access partial cachelines in all caches or DRAM. However, our proposal also applies to systems with just sectored upper level caches.

Besides sector cache, other solutions exist to access caches at sub-cacheline granularity. Prefetched data can be stored in a prefetch buffer instead of the cache, or two caches with different cacheline sizes can coexist. We only evaluate partial cacheline accessing in the sector cache context, but our mechanism is compatible with alternatives.

4.2 Granularity Predictor

To benefit from a sector cache we need to decide, for each cache miss, how many sectors to retrieve. For

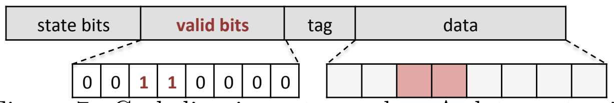


Figure 7: Cacheline in sector cache. A data sector is valid only when the corresponding valid bit is set.

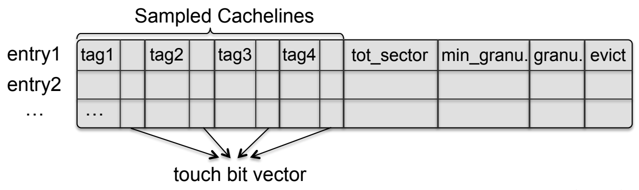


Figure 8: Partial cacheline Granularity Predictor (*GP*).

this paper, we only trigger partial accesses for indirect accesses, since other memory accesses (e.g., index accesses) are likely to have better spatial locality and benefit less from partial cacheline accessing.

The best access size depends on both the application and input, and thus needs to be determined dynamically. For example, if data array A is small and fits in cache, even though each indirect access to it touches only one sector in the cacheline, all the sectors may eventually be touched before the line is evicted. In this case, we should retrieve all sectors.

We add a *Granularity Predictor (GP)* to IMP to predict the best number of sectors to prefetch. Its architecture is shown in Fig. 8.

When IMP finds an indirect pattern, it allocates an entry in the *GP* and initializes the access granularity to a full cacheline. As IMP issues indirect prefetches, the *GP* randomly selects up to N prefetched cachelines to track; this sampling approach limits hardware cost since not all cachelines need to be tracked [46]. For each of these lines, the *GP* records the tag and a *touch bit vector*, which indicates which sectors in the cacheline have been touched by demand accesses. On a demand access, the *GP* checks if it is tracking the line, and if so, sets the appropriate bits in the touch bit vector.

On an eviction of a line the *GP* is tracking, the *GP* increments *evict* and computes the minimum access granularity of the cacheline by counting the smallest number of consecutive touched sectors. If this is smaller than *min_granu* field of the entry, *min_granu* is updated to this new value. The *GP* also counts the total number of sectors touched and adds this to the *tot_sector* field. After every N sampled cachelines are evicted, the *GP* updates the access granularity (*granu*) using Algorithm 1. Then, both *evict* and *tot_sector* are reset to 0 and *min_granu* is set to the number of sectors in a cacheline.

In the algorithm, *costFull* is the total number of sectors retrieved if the N cachelines are accessed in full. The +1 accounts for the header for each request. *costPartial* is the number of sectors retrieved if the N cachelines are partially accessed. Here, we have a header for each partial access.

Headers mean that retrieving few sectors at a time can carry high overhead. Thus, if software will eventually touch most sectors, it is more efficient to retrieve entire cachelines at once.

Algorithm 1 Granularity Prediction

```
costFull = N × (num_sectors_per_cacheline + 1)
costPartial = tot_sector + tot_sector / min_granu
if costFull ≤ costPartial then
    granu = cacheline_size
else
    granu = min_granu
end if
```

Table 1: System Configuration.

System Configuration	
Frequency	1 GHz
Number of Cores	N = 16, 64, 256
Core Model	In-order, single-issue
Memory Subsystem	
Cacheline Size	64 bytes
L1 I Cache	16 KB, 4-way
L1 D Cache	32 KB, 4-way
Shared L2 Cache	$2/\sqrt{N}$ MB per tile, 8-way
Directory Protocol	ACKwise [19]
2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	64 bits
Memory Model	
DRAMsim	10-10-10-24 DDR3, 8 banks per rank, 1 rank per MC
Simple DRAM Model	100 ns latency, 10 GB/s per MC

5. METHODOLOGY

We use the Graphite simulator [24] for our experiments. Table 1 shows the baseline system we model.

5.1 Scalability Assumptions

Table 1 shows how we change certain components of the processor with the number of cores, N. Specifically, we assume that the total L2 capacity and the total DRAM bandwidth are proportional to $\sqrt{N}$ rather than N, due to chip area and pin constraints. Our system uses the ACKwise coherence protocol [19] which is not fully-mapped and broadcasts an invalidation if the number of sharers exceeds 4.

We place the memory controllers in a diamond topology. This has been shown to provide the best performance for multicore architectures with a mesh network and X-Y routing, since the traffic can be uniformly distributed [3]. We use DRAMSim [43] in experiments without partial cacheline accessing. For experiments with partial accesses, we use a simpler model with a fixed latency and bandwidth limit. In our experiments, the simpler model produces results within 5% of DRAMSim.

5.2 IMP Parameters

We attach an IMP to each L1 cache. Table 2 shows the default parameters. Each *PT* has 16 entries and

Table 2: IMP Configuration.

IMP - Prefetch Table (*PT*)	
Table Size	16 entries
Max Number of Indirect Ways	2
Max Number of Indirect Levels	2
Max Prefetch Distance	16
IMP - Indirect Pattern Detector (*IPD*)	
Table Size	4 entries
Shift Values	2, 3, 4, -3
BaseAddr Array Length	4
Partial Accessing - Granularity Predictor (*GP*)	
L1 Sector Size	8 bytes
L2 Sector Size	32 bytes
Number of Samples	4 cachelines

the maximum prefetch distance is 16. For multi-way and multi-level indirection, we support only two ways of indirect patterns and two levels of indirection for each way. The *PT* can support more ways and levels with no extra storage overhead, but this is enough for all our applications. Each *IPD* has 4 entries, and in each entry, the BaseAddr array stores up to 4 misses. We consider four values of *shift*: 2, 3, 4 and -3. This corresponds to *Coeff* of 4 (e.g., for 32-bit integers), 8 (e.g., for double-precision floating point values), 16 (for small structures) and 1/8 (for bit vectors), respectively. Long bit vectors are sometimes used to indicate, in a dense way, which elements in a vector are non-zero. Since a byte contains 8 bits, a coefficient of 1/8 translates from bit offset to byte offset. With additional storage overhead, the *IPD* can support more *shift* values.

We consider systems with and without partial cacheline accessing. For systems with it, only IMP initiates partial cacheline accesses. The L1 cache has 8 byte sectors, which is the same size as an on-die network flit. The L2 cache has 32 byte sectors, which is half of a cacheline. Smaller L2 sectors would save even more bandwidth, but may require expensive changes to DRAM [4].

In our evaluation, we assume both the cache and TLB provide an extra port for prefetch requests. However, prefetches rarely conflict with normal memory accesses and only 1% performance is lost on average if they share a port.

5.3 Applications

We evaluate IMP on a set of graph analytics, machine learning, and sparse linear algebra codes intended to represent their domains.

For graph analytics and machine learning, we use the set of workloads from [37], and added LSH [38]. For BFS, included in that suite, we use Graph500 [29], intended to rate supercomputers on graph analytics tasks.

For sparse linear algebra, we use HPCG [10], a newly introduced component of the Top500 supercomputer rankings. This benchmark is intended to represent sparse high-performance computing workloads.

The only changes we made to the code were in porting to the simulator and adding software prefetches.

Pagerank: Pagerank is a graph algorithm for ranking a website based on the rank of the sites that link to it [32]. Pagerank is iterative; at each iteration, we update the pagerank of each vertex using the weighted sum of its neighbors' pageranks and degrees. Neighbor vertices are stored in a CSR representation [9], and accesses to neighbors require indirect accesses.

Triangle Counting: Counting the number of triangles in a graph is key to graph statistics such as clustering coefficients [7]. Our workload uses acyclic directed graphs, and each vertex has a neighbor list stored in CSR format. To find triangles, we intersect each vertex's neighbor list with its neighbor's neighbor lists. For efficiency, the local neighborhood list is converted to a bit vector which is indirectly accessed.

Graph500: Graph500 [29] is the first serious approach to augment the Top500 with data-intensive applications. It runs a breadth first search (BFS) on a graph following a power law distribution. Accessing neighbor vertices incurs indirect accesses.

SGD for Collaborative Filtering: Collaborative Filtering is a machine learning approach to predict how a user would rate an item based on an incomplete set of (user, item) ratings. Stochastic Gradient Descent (SGD) is an approach to solve the underlying matrix factorization problem [16]. The idea is to decompose the ratings matrix into two smaller matrices, a (user $\times$ features) matrix, and a (features $\times$ item) matrix, and learn these iteratively. For each available (user, item) entry, the algorithm makes a rating prediction using a dot-product of the row for the user in the first matrix and the column for the item in the second matrix, computes the error from the actual rating and updates the row and column entries using least squares error minimization. SGD uses indirect memory lookups into the row and column entries of non-zero matrix entries.

LSH: Locality Sensitive Hashing (LSH) [11] is a probabilistic machine learning algorithm for nearest neighbor search in high dimensional spaces. LSH uses multiple hash functions selected to cause similar objects to have a high probability of colliding. During a query, LSH finds and concatenates matching hash buckets for each hash table. With the right number of hash tables, LSH can guarantee with any desired accuracy that this list will contain all neighbors within a given query distance. The list may also contain objects far from the query; these are filtered out by computing distances from each list object to the query. Filtering is expensive [38] and involves indirect accesses to the entire dataset with the list of potential neighbors as indices.

SpMV: Sparse Matrix-Vector Multiplication (SpMV) is perhaps the most important sparse linear algebra primitive. Our code is from the HPCG benchmark [10] and has been optimized for multicore processors [33]. The sparse matrix is represented in the CSR format [9] and the vector is dense. The program scans the non-zero elements in the matrix row by row and indirectly accesses the corresponding elements in the vector.

SymGS: Symmetric Gauss-Seidel smoother (SymGS) is a key operation in the multigrid sparse solver from HPCG [10]. SymGS performs a forward and back triangular solve. The code groups rows to balance parallelism and data locality [33]. Like SpMV, SymGS scans non-zero elements in each row of the matrix and indirectly accesses corresponding elements in the vector.

5.4 Baselines

We use the following configurations in our evaluation.

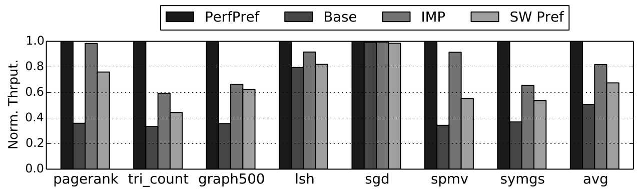
Ideal is an idealized configuration where all memory accesses hit in the L1. It is equivalent to a system with perfect prefetching and infinite memory and network bandwidth.

Perfect Prefetching is a weaker idealized configuration with a magic memory prefetcher. The prefetcher is able to look into the future and for each memory access, a prefetch request to that address is issued several thousand cycles before the request. Unlike **Ideal**, this configuration has finite memory and network bandwidth.

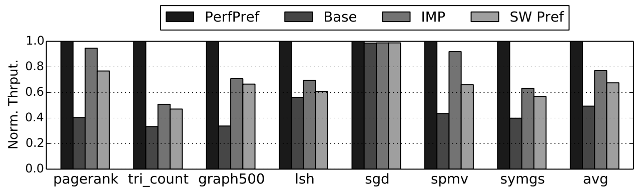
Baseline is the baseline system without IMP or any software prefetching. This configuration has a stream prefetcher attached to each L1 data cache.

Software Prefetching follows the algorithm proposed by Mowry [26] which inserts indirect prefetching instructions through the compilers. Despite the technique having been developed many years ago, no popular compilers (*i.e.*, gcc and icc) support it today. Therefore, we manually insert prefetching instructions into the source code following the original algorithm. We tried all possible prefetch distances and use the best one for each loop in each application. The underlying hardware is the same as *Baseline* which has a hardware stream prefetcher.

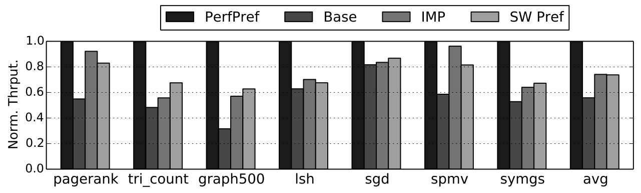
We also compared to a correlation prefetcher based on Global History Buffer (GHB) [31]. However, when attached to each L1 cache, our experiments show that GHB provides no benefits on top of the stream prefetcher because it cannot capture our workloads' indirect patterns with reasonably sized GHB buffers. GHB also increases pressure on the network and DRAM because it prefetches lines that are never accessed.



(a) 16 cores



(b) 64 cores



(c) 256 cores

Figure 9: Performance normalized to *Perfect Prefetching*

Table 3: Effectiveness of Streaming Prefetcher and IMP.

Application	Streaming Prefetcher			Streaming + IMP		
	Cov.	Acc.	Lat.	Cov.	Acc.	Lat.
Pagerank	0.08	1.00	3.07	0.96	1.00	1.13
Tri_Count	0.07	0.50	5.78	0.91	0.72	3.57
Graph500	0.28	0.87	3.36	0.74	0.94	1.61
SGD	0.56	0.70	2.21	0.67	0.72	1.68
LSH	0.22	0.50	5.90	0.78	0.66	4.04
SpMV	0.38	0.98	1.89	0.99	0.98	1.00
SymGS	0.37	0.97	3.25	0.87	0.96	2.01
Average	0.28	0.79	3.64	0.85	0.85	2.15

6. EVALUATION

6.1 Performance of IMP

Fig. 9 shows the performance of the configurations described in Section 5.4 and also of IMP, as described in Section 3 and without partial cacheline accessing. The results are normalized to *Perfect Prefetching*.

As shown and discussed in Section 2, *Baseline* performs significantly worse than *Perfect Prefetching*. The gap narrows with increasing core count because increased bandwidth pressure limits the ability of prefetching to hide latency. Still, even at 256 cores, we see significant performance potential.

Adding IMP significantly improves performance in all cases. On average, it provides 74%/56%/33% (up to 2.7×/2.3×/1.7×) speedup across all applications for 16/64/256 cores. Further, IMP brings average performance within 18%/23%/26% of the idealized *Perfect Prefetching* configuration, indicating that it harvests much of the potential improvement; other latency hiding techiques have limited room for additional improvement on top of IMP.

Although not shown in the figure, we also ran some SPLASH-2 benchmarks [44] that do not exhibit indirect access patterns and observed that IMP does not hurt performance on these benchmarks. This is as expected since it is very unlikely for IMP to trigger prefetching if no indirection exists.

6.1.1 Prefetching Effectiveness

Table 3 shows prefetch coverage, accuracy and latency to help us understand the performance. *Coverage* is the ratio of misses captured by prefetches to the overall number of misses. *Accuracy* is the ratio of the

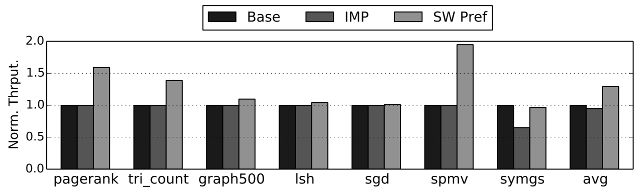


Figure 10: Instruction overhead of software prefetching at 64 cores, normalized to *Baseline*

prefetched cachelines that are later accessed to the total number of prefetches. *Latency* is the ratio of average memory access latency to that of *Perfect Prefetching*.

Table 3 shows the statistics at 64 cores. Relative to just a streaming prefetcher, IMP significantly improves prefetch coverage. Stream prefetching alone has an average coverage of 28%, while with IMP coverage is improved to 86%. For all but *SGD*, IMP covers the majority of the misses left by traditional streaming prefetching.

IMP is also accurate, achieving over 95% accuracy for half of the applications. The other applications are dominated by loops with small trip counts; accuracy is somewhat lower as IMP launches prefetches beyond the end of the loop.

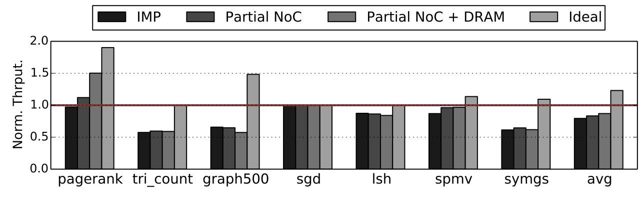
Small trip counts also contribute to *late* prefetches, or prefetches that only partly cover memory latency. At the beginning of the loop, IMP launches prefetches, but too late to fully hide latency. We see this, for example, in *Triangle Counting*, which has a coverage of 91%, but still high latency and a large remaining performance gap between IMP and *Perfect Prefetching*. For applications with large trip counts (*Pagerank* and *SpMV*), IMP is very timely, achieving memory latency close to *Perfect Prefetching*.

6.1.2 Software Prefetching

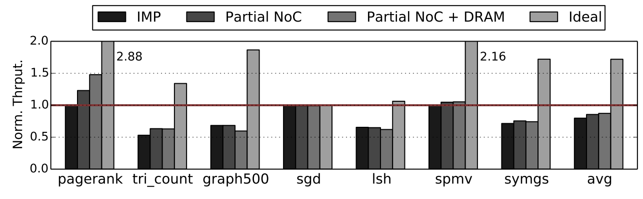
Fig. 9 shows that software prefetching also provides performance gains over *Baseline*. However, the gain is generally smaller than that from IMP, for the following two reasons.

First, not all access pattern information is exposed at compile time. In *Pagerank* and *SpMV*, for example, most runtime is spent in a nested loop where the start and end of the inner loop are input-dependent. However, for real inputs, the start is always the same as the end of the previous outer loop iteration, i.e., the code always scans the index array with unit stride. IMP detects this dynamically, but the compiler can not take advantage of it. As a result, the software prefetching algorithm only inserts prefetches within the inner loop and thus cannot hide all the latency.

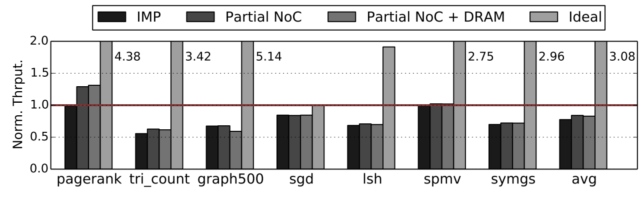
Second, software indirect prefetching incurs large instruction overhead. For each indirect prefetch $A[B[i+\Delta]]$, we must compute $i+\Delta$, then load the content of $B[i+\Delta]$ and compute the address of $A[B[i+\Delta]]$. Fig. 10 shows the instruction count of IMP and software prefetching relative to the *Baseline*. For all workloads except *SymGS*, IMP has the same instruction count as



(a) 16 cores



(b) 64 cores



(c) 256 cores

Figure 11: Performance with partial cacheline accessing normalized to *Perfect Prefetching*.

the *Baseline*. *SymGS* has busy waiting so the instruction count increases as runtime increases. On average, software prefetching incurs 29% (up to 2×) more instructions than IMP.

6.2 Performance of Partial Cacheline Accessing

Fig. 11 shows the performance of IMP with and without partial cacheline accessing normalized to *Perfect Prefetching*. The figure also shows *Ideal*, for reference. Partial cacheline accessing can reduce both the NoC traffic between L1 and L2, and off-chip traffic between L2 and DRAM. The figure includes results with partial accessing just in the NoC, and with it in the NoC and DRAM.

The performance of *Perfect Prefetching* decreases relative to *Ideal* as core count increases. The only difference between *Perfect Prefetching* and *Ideal* is the consideration of contention. Due to our scalability assumptions in Section 5.1, both the DRAM bandwidth and NoC bisection bandwidth scale with the square root of the core count. However, NoC and DRAM traffic scales linearly with core count. As core count increases, both subsystems become bottlenecks to scalability.

Partial cacheline accessing improves the performance of IMP. With partial accessing enabled only in the NoC, performance is improved by 4.9%/7.2%/8.4% (up to 15%/22%/31%) on 16/64/256 cores. Using it for DRAM as well, the performance gain becomes 9.5%/9.4%/6.9%

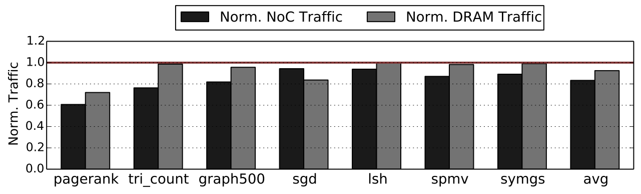


Figure 12: NoC and DRAM traffic of partial cacheline accessing normalized to full cacheline accessing (64 cores).

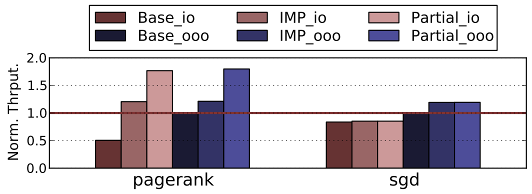


Figure 13: Performance of IMP on an in-order and out-of-order core, normalized to baseline out-of-order core at 64 cores.

(up to 55%/47%/33%) on 16/64/256 cores.

For applications with heavy off-chip traffic (e.g., *Pagerank*), reducing DRAM traffic significantly boosts performance. The performance can even be higher than *Perfect Prefetching* which always accesses full cachelines. For several other applications (*Triangle Counting*, *Graph500*, *LSH*, *SymGS*), partial accessing for DRAM hurts performance. With our inputs, these applications exhibit poor spatial locality in L1 but better spatial locality in L2. Thus, it is better to load partial cachelines from L2 but full cachelines from DRAM. Our granularity predictor chooses a single granularity for all prefetches from a given indirect pattern, whereas a more complex predictor could detect the best granularity at each level of the cache hierarchy.

Fig. 12 shows the NoC and DRAM traffic at 64 cores with partial cacheline accessing, normalized to full cacheline accessing. On average, partial accessing reduces NoC traffic by 16.7% and DRAM traffic by 7.5%. The reduction is most significant in *Pagerank*, 39% in NoC and 28% in DRAM, which sees the largest performance gain.

6.3 Sensitivity Studies

6.3.1 Core Design

Our default core is relatively simple, as one would expect for a system with up to 256 cores. Intuitively, more complex designs, including the use of out-of-order (OoO) execution, may hide memory latency better, reducing the need for IMP.

Fig. 13 shows the performance of IMP and partial cacheline accessing on both our default core and an out-of-order core, for 64 cores. Partial cacheline accessing is enabled in both NoC and DRAM. Our out-of-order core is still modest, using a 32-entry reorder buffer. This mimics Intel's announced Knights Landing many-core design based on the Silvermont core [1]. We show results for one memory-bound application, *Pagerank*, and one compute-bound application, *SGD*.

For both applications, OoO execution improves performance, but IMP continues to provide significant performance benefits.

For *Pagerank*, OoO execution hides some latency, but still leaves room for improvement. As the application is memory-bound, and IMP is so effective, with IMP, we see very little difference between the core designs.

For *SGD*, OoO execution improves performance significantly by exploiting more instruction level parallelism. IMP actually provides *more* benefit for the core with OoO execution because the OoO execution hides little latency, but accelerates the computation, making the application less compute-bound.

Over all our applications, on average, IMP and partial accessing provide 20% and 37% speedup, respectively, on OoO cores compared to the baseline OoO with only streaming prefetchers.

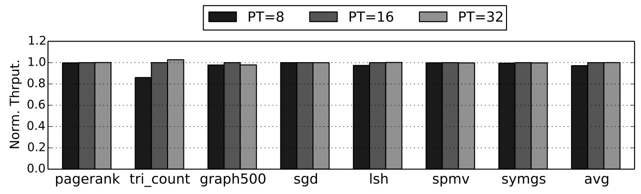


Figure 14: Sensitivity to PT size at 64 cores. Performance normalized to the default PT size = 16.

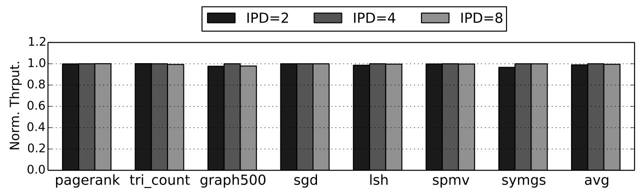


Figure 15: Sensitivity to IPD size at 64 cores. Performance normalized to the default IPD size of 4.

6.3.2 IMP Design Parameters

We now study sensitivity to three hardware parameters for IMP: PT size, IPD size and prefetch distance.

Fig. 14 shows the performance of IMP at 64 cores for various numbers of PT entries. Most applications are not sensitive to the PT size. These have few concurrent indirect patterns, so even eight entries is enough to capture all the patterns. *Triangle Counting* and *LSH* see some additional benefit from more entries. With 16 entries, the performance is 16.3% and 2.7% higher than with 8 entries for these two applications, respectively. However, the improvement of 32 entries over 16 entries is only 2.7% for *Triangle Counting* and less than 0.2% for the other applications.

Fig. 15 shows the performance of IMP at 64 cores for various numbers of IPD entries. Most applications are not sensitive to the IPD size. The IPD is only used for *detecting* indirect patterns; an entry is released as soon as IPD detects an indirect pattern or finds that no indi-

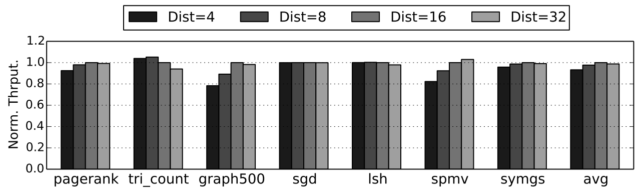


Figure 16: Sensitivity to max indirect prefetch distance at 64 cores. Performance normalized to default prefetch distance of 16.

rect pattern exists for the streaming access. Since most indirect patterns are very stable, the *IPD* sees little use. One exception is *SymGS*, where indirect patterns are frequently detected. With 4 *IPD* entries, performance is 3.5% higher than with 2 entries. The improvement of 8 entries over 4 entries is less than 0.1%.

Finally, Fig. 16 shows the performance of IMP at 64 cores for various maximum indirect prefetch distances. For applications with long streams of indirect accesses (*Graph500*, *Pagerank* and *SpMV*), a large prefetch distance helps improve performance. However, for applications with many short loops with indirect accesses (*Triangle Counting*), increasing the prefetch distance may hurt performance since more prefetches are for elements beyond the end of the loop, and thus go unused. A scheme to detect this situation and dynamically decrease prefetch distance would help.

6.4 Hardware Cost

6.4.1 Storage Cost of Basic IMP

IMP's storage overhead consists of a *Prefetch Table* (*PT*) and an *Indirect Pattern Detector* (*IPD*). For the following discussion, we assume an address space of 48 bits.

In our default configuration, the *PT* has 16 entries. We assume a baseline CPU has a stream prefetcher, so the only new overheads in the *PT* are from the indirect table. The most expensive fields are the *BaseAddr* (48 bits) and *index* (48 bits). Overall, each entry requires less than 120 bits. With 16 entries, the total *PT* storage overhead is less than 2 Kbits.

The *IPD* has 4 entries. In each entry, the most expensive fields are the two index values (48 bits each) and the *BaseAddr* array which stores 4×4 *BaseAddrs* (48 bits each). In total, the *IPD* requires 3.5 Kbits.

Overall, IMP requires 5.5 Kbits or only 0.7 KB of storage. If this is still too large, we can reduce the number of entries in *PT* and/or *IPD*. As shown in Section 6.3.2, performance of most applications degrades little when we shrink these tables.

6.4.2 Storage Cost of Partial Cacheline Accessing

We assume sector caches for both L1 and L2 to support partial cacheline accessing. In this paper, we split each L1 cacheline into 8 sectors, and each L2 cacheline into 2 sectors. This requires an 8-bit/2-bit valid mask for each L1/L2 cacheline, which is 1.6% and 0.4% storage overhead for the L1 and L2, respectively.

The Granularity Predictor (*GP*) also incurs overhead. The *GP* has the same number of entries (16) as the *PT*. We assume 4 samples in each entry. Each sample includes an address tag ($48 - \log_2 64 = 42$ bits) and a bit mask (8 bits). Including the other fields in Fig. 8, the total storage for an entry is less than 210 bits. The overall storage of *GP* is 3.4 Kbits or 420 bytes.

6.4.3 Energy Cost

We model the energy overhead of IMP using CACTI [28]. *IPD* is only accessed on L1 misses, which have a high energy cost; thus, the overhead of accessing *IPD* is negligible. The *PT*, however, is accessed on every L1 access. We model the *PT* as a fully associative cache with 16 entries. Each entry has a 96-bit tag (for address and *PC*) and 120-bit data. Each *PT* access takes less than 3% of the energy of a baseline L1 access. This is a small overhead compared to the performance gain we deliver and this overhead can be further reduced by using a smaller *PT*, less associativity, or applying some throttling. The *GP* is accessed once per indirect access, and the energy is less than 1% of the energy of an L1 access.

7. RELATED WORK

The body of work on hiding memory latency is quite large. We discuss hardware prefetching, software prefetching and other latency hiding techniques. We also discuss related work on bandwidth reduction techniques.

7.1 Hardware Prefetching

Streaming and striding hardware prefetchers are well-established [6, 41] and implemented on commercial processors. These prefetchers capture simple memory access patterns, similar to our baseline design.

Correlation prefetching [13, 31] leverages the observation that in many applications, irregular memory accesses repeat themselves. We can thus store the address pattern (e.g., when we access address A, we then access address B) in a hardware table and use it to drive prefetches for repeated address streams. This works well for short, repeated streams, but requires large storage for capturing long streams, such as in our workloads [12].

7.2 Software prefetching

We may add software prefetches to a program either manually or automatically [25]. Manual insertion is very flexible, but requires significant programmer effort. Automatic insertion requires the compiler to recognize the access pattern.

Previous work studied inserting software prefetching instructions automatically through the compiler for simple access patterns like streaming or striding [27]. Several works have tried to prefetch more complex patterns [20, 23], but these patterns do not correspond to the indirect patterns in this paper. Compiler insertion of indirect prefetches [26] has been proposed, and to the best of our knowledge has only been implemented on the Intel Xeon Phi compiler [17] and is not turned on by default. As shown in Section 6.1.2, software indirect prefetching achieves suboptimal performance partly due to its high instruction count.

Several works proposed prefetching techniques for pointer chasing [23, 35]. This is not the same as the streaming indirect accesses we primarily target, but is similar to our multi-level indirect accesses. A multi-level pattern may be thought of as a very short pointer chain. We believe it is possible to extend IMP to support pointer chasing prefetching, but leave that for future work.

7.3 Other latency hiding techniques

Besides prefetching, many other techniques have been proposed to hide memory access latency. Out-of-Order execution (OoO) [39] hides latency by executing independent instructions while waiting for a memory access. As shown in Section 6.3.1, OoO is not enough to hide all latency and IMP is also effective with OoO.

Simultaneous multithreading (SMT) [15, 40] shares the pipeline among multiple hardware contexts. Multithreading hides latency by allowing other threads to proceed when one thread is stalled on a memory access. Compared to IMP, this incurs significant hardware overhead for extra contexts and requires software to use additional threads.

Run-ahead execution [30] allows a thread to speculatively continue program execution rather than stall on a cache miss. Speculative execution may effectively prefetch data needed by the application in the near future. This technique, however, requires an extra set of register files for checkpointing before the processor enters the run-ahead mode, so it can recover the processor's state after coming back to the normal mode.

In helper threads [8, 14, 22, 47], a separate thread runs a reduced version of the program to reach and execute memory accesses earlier than the primary thread. This prefetches data for the primary thread but requires another hardware context or core for the extra thread, as well as hardware or a software tool to create the helper threads' version of the code.

7.4 Bandwidth reduction

Sector cache was built into very early commercial computers [21]. The special design was motivated by the discrete transistor logic of the time [36] rather than by bandwidth reduction.

Similar to this paper, several previous works have proposed to use sector cache and partial cacheline accessing in CPUs [45, 46, 18] and GPUs [34] to reduce main memory traffic for general applications. Our idea and implementation are different since we only focus on indirect accesses. Also, for a system with IMP, the incremental cost of adding a granularity predictor is relatively low.

8. CONCLUSION

We observe that indirect memory accesses in the form $A[B[i]]$ are both common and performance-critical for applications using sparse data structures. To capture these patterns, we propose an indirect memory prefetcher (IMP). We also observe that indirect memory accesses often have little spatial locality, and thus waste bandwidth. Thus, we consider partial cacheline accessing as an additional feature of IMP, to reduce bandwidth usage in the NoC and DRAM. Evaluated on seven benchmarks, IMP achieves significant performance improvement while reducing the NoC and DRAM traffic significantly.

9. ACKNOWLEDGEMENT

This research was funded by the Intel Science and Technology Center for Big Data. We would like to thank Jongsoo Park for help with the sparse matrix workloads and Sheng Li for help with power modeling.

10. REFERENCES

[1] "Intel launches low-power, high-performance silvermont microarchitecture," http://intel.ly/1pFRQwV, May 2013.

[2] "Cuda programming guide," http://docs.nvidia.com/cuda/, 2015.

[3] D. Abts, N. Enright, J. J. Kim, D. Gibson, M. Lipasti, and G. Inc, "Achieving predictable performance through better memory controller placement in many-core cmps," in *In ISCA âĂŽ09: Proceedings of the 36th annual international symposium on Computer architecture*, 2009.

[4] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on.* IEEE, 2009, pp. 1–12.

[5] J. Canny and H. Zhao, "Big data analytics with small footprint: Squaring the cloud," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2013, pp. 95–103.

[6] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, 1995.

[7] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, Feb. 1985.

[8] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *ACM SIGARCH Computer Architecture News*, vol. 29, no. 2. ACM, 2001, pp. 14–25.

[9] J. Dongarra, "Compressed row storage," http://web.eecs.utk.edu/ dongarra/etemplates/node373.html.

[10] ——, "Toward a new metric for ranking high performance computing systems," 2013.

[11] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98. ACM, 1998, pp. 604–613.

[12] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2013, pp. 247–259.

[13] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 252–263.

[14] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using

migrating helper threads," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 393–404, 2011.

[15] P. Konecny, "Introducing the cray xmt," in *Proc. Cray User Group meeting (CUG 2007). Seattle, WA: CUG Proceedings*, 2007.

[16] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[17] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito, "Compiler-based data prefetching and streaming non-temporal store generation for the intel (r) xeon phi (tm) coprocessor," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International.* IEEE, 2013, pp. 1575–1586.

[18] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3. IEEE Computer Society, 1998, pp. 357–368.

[19] G. Kurian, J. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. Kimerling, and A. Agarwal, "ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network," in *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[20] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "Spaid: Software prefetching in pointer-and call-intensive environments," in *Proceedings of the 28th annual international symposium on Microarchitecture.* IEEE Computer Society Press, 1995, pp. 231–236.

[21] J. S. Liptay, "Structural aspects of the system/360 model 85, ii: The cache," *IBM Systems Journal*, vol. 7, no. 1, pp. 15–21, 1968.

[22] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on.* IEEE, 2001, pp. 40–51.

[23] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5. ACM, 1996, pp. 222–233.

[24] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *International Symposium on High-Performance Computer Architecture*, 2010.

[25] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of parallel and Distributed Computing*, vol. 12, no. 2, pp. 87–106, 1991.

[26] T. C. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 1, pp. 55–92, 1998.

[27] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *ACM Sigplan Notices*, vol. 27, no. 9. ACM, 1992, pp. 62–73.

[28] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep., 2009.

[29] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[30] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on.* IEEE, 2003, pp. 129–140.

[31] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *Micro, IEEE*, vol. 25, no. 1, pp. 90–97, 2005.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." 1999.

[33] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 945–955.

[34] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2013, pp. 86–98.

[35] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5. ACM, 1998, pp. 115–126.

[36] J. B. Rothman and A. J. Smith, "Sector cache design and performance," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on.* IEEE, 2000, pp. 124–133.

[37] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of SIGMOD 2014.* ACM, 2014, pp. 979–990.

[38] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey, "Streaming similarity search over one billion tweets using parallel locality-sensitive hashing," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1930–1941, 2013.

[39] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[40] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[41] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys (CSUR)*, vol. 32, no. 2, pp. 174–199, 2000.

[42] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: distributed machine learning and graph processing with sparse matrices," in *Proceedings of the 8th ACM European Conference on Computer Systems.* ACM, 2013, pp. 197–210.

[43] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "Dramsim: a memory system simulator," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995.

[45] D. H. Yoon, M. K. Jeong, and M. Erez, "Adaptive granularity memory systems: a tradeoff between storage efficiency and throughput," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 295–306, 2011.

[46] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 548–559.

[47] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on.* IEEE, 2001, pp. 2–13.