APT-GET: Profile-Guided *Timely* Software Prefetching

Saba Jamilan
University of California, Santa Cruz
USA
sjamilan@ucsc.edu

Tanvir Ahmed Khan
University of Michigan
USA
takh@umich.edu

Grant Ayers
Google
USA
granta@google.com

Baris Kasikci
University of Michigan
USA
barisk@umich.edu

Heiner Litz
University of California, Santa Cruz
USA
hlitz@ucsc.edu

Abstract

Prefetching which predicts future memory accesses and preloads them from main memory, is a widely-adopted technique to overcome the processor-memory performance gap. Unfortunately, hardware prefetchers implemented in today's processors cannot identify complex and irregular memory access patterns exhibited by modern data-driven applications and hence developers need to rely on software prefetching techniques. We investigate the challenges of enabling effective, automated software data prefetching. Our investigation reveals that the state-of-the-art compiler-based prefetching mechanism falls short in achieving high performance due to its static nature. Based on this insight, we design *APT-GET*, a novel profile-guided technique that ensures prefetch timeliness by leveraging dynamic execution time information. *APT-GET* leverages efficient hardware support such as Intel's Last Branch Record (LBR), for collecting application execution profiles with negligible overhead to characterize the execution time of loads. *APT-GET* then introduces a novel analytical model to find the optimal prefetch-distance and prefetch injection site based on the collected profile to enable timely prefetches. We study *APT-GET* in the context of 10 real-world applications and demonstrate that it achieves a speedup of up to 1.98× and of 1.30× on average. By ensuring prefetch timeliness, *APT-GET* improves the performance by 1.25× over the state-of-the-art software data prefetching mechanism.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Architectures.*

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519583

Keywords: Software Prefetching, Compiler Analysis

ACM Reference Format:
Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. *APT-GET*: Profile-Guided *Timely* Software Prefetching. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519583

1 Introduction

A vast majority of today's software runs on processors inspired by the Von-Neumann architecture. Consequently, the Von-Neumann bottleneck (*i.e.*, the processor-memory speed gap [49, 80, 125]) is the root cause of many performance problems in today's software systems [14, 50, 62, 65, 98]. To make matters worse, the data-driven nature of modern applications (*e.g.*, machine learning [6, 34, 127],mobile applications [22, 24], and data analytics [23, 25, 128, 130]) has increased data footprints significantly, thus limiting the ability of traditional approaches such as deeper cache/memory hierarchies or compile-time data locality optimizations to scale and provide significant performance improvements [14]. Consequently, widely-used modern applications lose more than 60% of all processor cycles due to frequent on-chip cache misses and the subsequently induced high memory access latency [13, 41, 62, 113].

Prefetching—anticipating upcoming memory accesses and loading them before their use—can hide this memory access latency if performed *accurately* and in a *timely* manner [48]. Therefore, a rich body of hardware [18, 30, 40, 59, 68, 88, 95, 108, 114, 121, 122] and software [9, 32, 33, 37, 78, 83, 89–91, 116] data prefetching mechanisms have been proposed in the literature to reduce memory access latency. While there exists an exotic range of irregular data prefetching proposals (*e.g.*, record and replay prefetchers [118] and indirect prefetchers [126]) in the computer architecture literature, only simple prefetchers (*e.g.*, next-line [109] and stride prefetchers [61]) are implemented in today's hardware since complex prefetchers require impractical on-chip

metadata storage along with significant hardware modifications [8, 10].

Consequently, to avoid the memory access latency induced by irregular access patterns (*e.g.*, indirect array access of the form, $A[B[i]]$), developers must rely on software data prefetching mechanisms [65]. Manual software prefetching performed by programmers has shown to be cumbersome and error prone as it is difficult to evaluate the efficacy of a manually inserted prefetch [76]. For irregular accesses, memory addresses are computed by sequences of instructions (the load-slice) rendering manual prefetch injection challenging [9]. Code changes can easily break the prefetch address computation leading to inaccurate prefetches. As software prefetching introduces an instruction overhead, inaccurate prefetches can lead to a performance regression. Recent work on compiler-based automatic prefetch injection schemes [9] has addressed the challenge of generating accurate prefetch-slices, however, it is still unable to address the memory access latency problem satisfactorily.

In this work, we first perform a comprehensive characterization of existing automated software data prefetching mechanisms. Specifically, we investigate why the state-of-the-art software data prefetching mechanism [9] falls significantly short of an ideal (in terms of accuracy, coverage, and timelines) data prefetcher. In our investigation, we find that the existing *static* software-based solutions fail to prefetch memory blocks in a timely manner, thereby missing significant performance opportunities. In particular, prefetches generated too early may be evicted from the processor's caches unused while late prefetches are unable to hide the access latency of demand loads completely. We find that for timely prefetching, *dynamic* information such as the execution time of the optimized code is required. Unfortunately, state-of-the-art software mechanisms only rely on static heuristics to inject prefetch instructions and hence cannot achieve the full performance potential of an ideal data prefetcher.

Driven by our analysis, we propose *APT-GET*[1], a novel profile-guided mechanism to ensure the timeliness of software prefetch operations. *APT-GET* realizes software prefetch timeliness by effectively finding the optimal value for two key parameters: *prefetch-distance* and *prefetch injection site*. We define prefetch-distance as the distance between the current memory access and a future memory access, measured in terms of the number of memory accesses. It defines how far into the future we need to prefetch and it is determined by the program execution time that elapses in between the memory accesses. Prefetch injection site, on the other hand, determines the program location where the prefetch instruction is inserted. Instead of exhaustively searching over all-possible values of prefetch-distance and prefetch injection site, *APT-GET* obtains these values through a new profiling methodology leveraging existing hardware support. In particular, *APT-GET* profiles the elapsed time between two instances of the same memory access instruction to determine near optimal prefetch-distance and prefetch injection site for the corresponding prefetch.

We evaluate *APT-GET* in the context of 10 real-world, memory-latency-bound applications. Across all applications, *APT-GET* achieves an average execution time speedup of 1.30×. By optimizing the prefetching timeliness, *APT-GET* significantly outperforms the state-of-the-art software prefetching mechanism [9] and provides on average 1.25× greater speedup.

Overall, we make the following contributions:

- A thorough investigation of how existing software prefetching techniques fall significantly short of an ideal data prefetcher due to lack of prefetch timeliness
- *APT-GET*: A profile-guided mechanism to ensure software prefetch timeliness by identifying the optimal prefetch-distance and the optimal prefetch injection site.
- An LLVM compiler pass for automatically injecting prefetches that supports variable prefetch-distance and prefetch injection site.
- An evaluation showing *APT-GET*'s effectiveness at ensuring prefetch timeliness for several widely-used memory-bound applications while achieving a significant performance improvement.

As an outline for the rest of this paper, we first characterize the key challenges of automated software prefetching in §2. Next, we describe *APT-GET*'s design in §3. We then describe *APT-GET*'s evaluation on real-world applications and benchmarks in §4. After discussing the related work in §5, we finally conclude in §6.

2 Understanding the Challenges of Automated Software Prefetching

In this section, we investigate the performance of existing automated software prefetching techniques and show why they fall short of providing high performance. In particular, we demonstrate that while existing techniques can achieve high accuracy and coverage, they are often unable to generate timely prefetches. This is because existing approaches utilize static techniques to inject prefetch instructions without incorporating dynamic information such as execution time. We perform this in-depth investigation using a microbenchmark and highlight the challenges of injecting timely prefetch instructions.

2.1 Methodology

We analyze existing automated software prefetching techniques using a microbenchmark with an indirect memory access pattern as shown in Listing 1. The microbenchmark implements a two-nested loop leveraging indirect memory addresses to retrieve a data value from a target array `T`. The inner loop furthermore executes a `do_work()` function

[1] as an appropriate and timely (*APT*) prefetch (*GET*)

```
#define SIZE = ... ;
int BI[SIZE]; // all values are between 0 and SIZE
int BO[SIZE]; // all values are between 0 and SIZE
int T[2*SIZE];

void mbench(int prefetch_distance, int INNER) {
   int OUTER = SIZE/INNER;
   for (int e : OUTER) {
     for (int i : INNER) {
         int val = T[BO[e]+BI[i]];
         //__builtin_prefetch((&T[BO[e]+BI[i+prefetch_distance]]));
         do_work(COMPLEXITY, val);
     }
   }
}
```

Listing 1. Microbenchmark executing indirect memory accesses and a work function. The parameters INNER and COMPLEXITY denote the number of inner loop iterations and the complexity of the work function respectively.

whose work is dependent on the loaded data. We define two parameters to affect the behavior of the microbenchmark. `INNER` determines the trip count [57] of the inner loop, while `COMPLEXITY` defines the time spent in the work function. We perform our microbenchmark analysis on an Intel Xeon Gold 6242R CPU running at 3.10GHz (4.1GHz Turbo) and 768GByte of DDR4-2666 DRAM. Because the microbenchmark is performing indirect memory accesses, Intel's hardware prefetchers [115] are unable to predict the irregular memory addresses, leaving opportunities for software prefetching.

We automatically inject software prefetches utilizing the state-of-the-art software prefetching approach [9] implemented as an LLVM compiler pass. The pass operates at the intermediate representation (IR) level of LLVM and determines software prefetching opportunities through static code analysis. The pass identifies indirect memory accesses (loads) and then performs a backward data dependency analysis utilizing depth-first search until it finds the first loop induction variable, while keeping track of all the instructions that are encountered during this search. Since load instructions are located inside the loop, their memory addresses are dependent on the induction variable of the loop. Therefore, we need to know the value of the induction variable in each loop iteration to calculate the next memory addresses of the load that we want to prefetch. We can calculate the addresses for the next iterations by adding the prefetch-distance to the current induction variable. Additionally, the key difference between irregular access patterns such as in pointer chases and indirect memory accesses inside loops is, that all indirect accesses depend on the induction variable. Therefore, we need to know the induction variable value to generate prefetch instructions.

The identified instructions, which we refer to as a *load-slice*, are duplicated for prefetching. This duplicated load-slice is then transformed by replacing the original

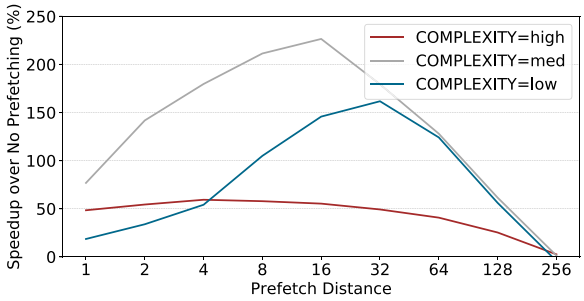

Figure 1. Performance impact of prefetching various distances for indirect memory accesses with 256 inner loop iterations and varying work function complexity

load instruction with a prefetch instruction. The prefetch instruction's address is computed by adding a constant *prefetch-distance* to the address of the original memory access. This allows prefetching memory blocks that will be accessed in subsequent loop iterations. We note that the technique described above relies on static code transformation, and it also depends on programmer-specified flags (*e.g.*,`-DFETCHDIST=32`) to ensure the timeliness of prefetches by tuning the prefetch-distance.

Next, we show that static approaches do not generalize well across a large variety of application use cases. Moreover, we also demonstrate that static techniques are unable to realize a significant amount of the performance benefits offered by the optimal software prefetching mechanism that prefetches memory blocks with near-perfect accuracy, i.e., by covering all potential data cache misses in a timely manner.

2.2 Prefetching Timeliness

For the first experiment, we configure the microbenchmark to utilize a loop trip count of $INNER = 256$. We choose three different work functions with low, medium, and high complexity. Figure 1 shows the speedup obtained by injecting prefetches for different prefetch-distance. There are a number of interesting observations. First, the potential performance gains delivered by prefetching are significant, exceeding 200% for a prefetch-distance of 16 and a medium complexity work function. Second, choosing the optimal prefetching distance has a significant performance impact. Third, the optimal prefetch-distance varies between configurations and depends on the complexity of the work function. In particular, for the low, medium, and high complexity work functions, the optimal prefetch-distance is 32, 16, and 4 respectively. Existing techniques [9] utilize a static prefetch-distance and hence do not provide optimal and generalized performance for different applications.

2.3 PMU Counter Study

To provide additional insight, we perform a performance analysis with the tool, `perf stat` [43, 71]. We analyze

Table 1. Prefetch accuracy and timeliness depending on the prefetch-distance

Prefetch	IPC	Prefetch Accuracy	Late Prefetch
None	0.33	0%	0%
Dist-1	0.42	72%	95%
Dist-64	0.73	70%	1%
Dist-1024	0.29	3%	0%

the performance of the microbenchmark choosing a low work complexity and $INNER = 256$ while varying the prefetching distance between 0, 1, 64, and 1024. Table 1 shows the instructions per cycle (IPC) performance as well as the *prefetch accuracy* which is defined as the number of prefetches (`offcore_requests.all_data_rd`-`offcore_requests.demand_data_rd`) divided by the number of all loads (`offcore_requests.all_data_rd`). As can be seen, as soon as prefetching is enabled with a prefetch-distance of 1 or 64, 70% of all demand loads are effectively prefetched, proving that automated injection passes are effective in determining the correct addresses to prefetch. However, when the distance (1024) exceeds the loop trip count, prefetches are no longer accurate, since most of them are too early prefetches. Therefore, prefetches which are generated by using a very large prefetch-distance, can be evicted from the cache before they are used while displacing other useful data.

Observation: PMU counters reveal that for a range of prefetch-distance, a significant fraction of demand loads can be correctly prefetched using automatic prefetch injection.
Insight: *Static prefetch injection is sufficient to enable high prefetching coverage and accuracy.*

Column 4 of Table 1 shows the *late prefetch* ratio which is defined as the number of demand loads hitting a prefetch residing in the fill buffer (FB) of the processor (`LOAD_HIT_PRE.SW_PF`). Processors utilize fill buffers or miss status hold registers to coalesce multiple loads to the same cache line into a single memory access. The occurrence of this event means that a demand load was correctly prefetched, however, that the prefetch was issued too late. As the prefetched cache line has not yet been retrieved from memory, the demand load needs to stall until the prefetch has been completed.

Observation: For small prefetching distances, processors exhibit many events where a demand load hits a corresponding prefetch in the fill buffer.
Insight: *Static prefetching techniques are unable to consistently achieve timely prefetches, thus dynamic profiling information is required.*

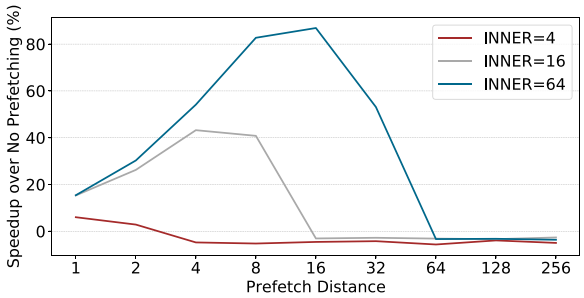

Figure 2. Performance impact of prefetch-distance for indirect memory access kernel with low work function complexity and varying inner loop trip count

2.4 Prefetch Injection Site

Table 1 showed that for a too-large prefetch-distance, prefetches are no longer accurate (nor timely) and hence may lead to a performance regression. Figure 2 shows the prefetching performance for the microbenchmark using low work function complexity while varying the loop trip count between 4, 16, and 64. It can be seen that for a loop trip count of 4, prefetching is no longer beneficial while for trip counts of 16 and 64 improvements are moderate and furthermore require a small prefetch-distance. Existing static prefetch injection techniques offer no flexibility besides injecting prefetches in the inner loop as they possess no information about the optimal *prefetch injection site*. To enable significant prefetching performance gains in cases where the loop trip count is small, a prefetching mechanism should also be able to evaluate additional prefetch injection sites such as the outer loop based on dynamic profiling information.

Observation: Choosing the prefetch injection site statically for example by always injecting prefetches into the inner loop does not provide significant performance gains for loops with low trip counts.
Insight: *Dynamic techniques are required to determine the optimal injection site of a prefetch.*

2.5 Static Techniques to Infer Execution Time

Static compile-time techniques can, in principle, predict the execution time of input-independent loops by counting the number of instructions and by leveraging cost models [27, 44, 123] to infer the cost of each instruction. However, due to the complexity of contemporary microprocessors, cost models show limited accuracy [36]. The state-of-the-art static techniques [4, 29, 58, 73, 74, 87, 103] incur 9-36% average errors while predicting the execution time of basic blocks even under the assumption that all memory access times are constant and well-known [36]. Moreover, these cost models have to be well maintained and frequently updated when the hardware changes [58]. For instance, as modern superscalar processors are deeply pipelined executing multiple

instructions simultaneously, the cycle-per-instruction (CPI) of a particular instruction is not fixed, but instead, it depends on its data and control flow dependencies. Furthermore, the average memory access time of a load is significantly affected by its locality and cache-ability which is generally unknown at compile time. Lastly, in the presence of input-dependent code, static techniques cannot predict the execution time. For these reasons, we propose a dynamic profile-guided technique to predict the execution time of loops enabling us to infer the elapsed time between two instances of the same load instruction.

3 Design of *APT-GET*

Our analysis shows that the prefetch *timeliness*, both in terms of the prefetch-distance and the prefetch injection site, significantly affects the effectiveness of automated software prefetching mechanisms in achieving predictable high-performance gains across different applications. While a complete design space exploration can identify the best configuration, performing such an exhaustive search over all prefetch-distances and prefetch injection sites is infeasible in large-scale real-world software systems. Hence, we propose *APT-GET*, a novel profile-guided mechanism to identify the optimal prefetch-distance and the optimal prefetch injection site using only a single profiling run to capture the dynamic behavior of an application. Specifically, *APT-GET* employs efficient hardware support (Intel's LBR [70]) to collect the application profile with negligible overhead (§3.1). As widely-used profile-guided code layout optimization techniques [31, 52, 66, 96, 97] already collect similar program execution profiles in production, *APT-GET* can be seamlessly integrated into existing systems. Based on this profile, *APT-GET* applies a novel analytical technique to find both the optimal prefetch-distance (§3.2) and the optimal prefetch injection site (§3.3). Without LBR, *APT-GET* lacks the dynamic information needed to improve prefetch timeliness. While it may be possible to estimate loop execution times with software techniques [86], LBR provides the most accurate results with the lowest overheads. Apart from Intel processors, AMD processors support branch sampling [47] and future ARM processors will implement the Branch Record Buffer Extension (BRBE) [99], which can be used as an alternative to LBR. Finally, *APT-GET* revises existing compiler-based prefetching mechanisms to incorporate these optimal prefetch configurations to ensure the timeliness of prefetch operations (§3.5).

3.1 Profile Collection

Enabling timely prefetches requires a detailed characterization of the corresponding demand loads. This characterization includes the hit/miss ratio and the performance impact of the load, the trip count of the loop containing the load instruction, and the execution time of a single loop iteration. For instance, as shown in the analysis section (§2), we need to determine the elapsed time between two instances of the same load instruction for computing the optimal prefetch-distance. The loop latency (execution time) is hereby defined by two components: the *instruction component (IC)* and the *memory component (MC)*. The instruction component includes all (non-load) instructions implementing the loop. The latency of this component, IC_latency depends on the number of instructions and their data and control flow dependencies. However, frequently in practice, IC_latency does not differ significantly across different loop iterations. Moreover, this latency is constant even in the presence of optimal prefetching. The memory component, on the other hand, includes the loads causing frequent cache misses. Therefore, the latency of this component, MC_latency is determined by the level within the memory hierarchy that serves the load. As the L1 cache access latency (4 cycles) differs significantly from the DRAM access latency (hundreds of cycles), MC_latency is highly variable. Our prefetching technique captures this variance to identify the optimal prefetch-distance. To determine the optimal prefetch-distance, we need to learn the latency of both the instruction and memory component, so that:

$$IC\_latency \times prefetch\_distance = MC\_latency \quad (1)$$

If Equation (1) holds then the MC_latency can be hidden with prefetching. To separate the IC_latency from the MC_latency, it is insufficient to measure the *average* time between two instances of a load. Instead, we need to predict the execution time of a loop in the absence of cache misses deriving the optimal prefetch distance.

To enable the load characterization outlined above, we leverage the Last Branch Record (LBR) feature offered by Intel CPUs [70]. The LBR is a buffer that holds several key pieces of information about the last 32 basic blocks (BBL) executed by the CPU. A basic block is defined as a sequence of consecutive instructions that was terminated by a *taken* branch. Hence, when *APT-GET* collects hardware performance event samples with the LBR feature enabled, the collected profile includes LBR entries for the last 32 taken branches immediately preceding the instruction that triggers the performance event. We show an example schematic view of the multiple LBR entries in Figure 3.

PC	OE	IE	IE	OE	IE	IE	IE	OE
Target	OS	IS	IS	OS	IS	IS	IS	OS
Time	4	9	11	13	19	21	24	26

Figure 3. Schematic view of the Intel CPU's Last Branch Record (LBR) highlighting the outer loop branches in blue, the inner loop branches in orange and the cycle times of each branch in green

As we show in Figure 3, each LBR entry contains the program counter (PC) of a taken branch, the target of the branch, and the CPU cycle when the branch was executed. By finding two instances of the same branch PC implementing a loop and subtracting their cycle counts, we can compute the execution time of a loop iteration. As a specific demand load instruction exists exactly once for a single loop iteration, this enables us to compute the elapsed time between two instances of the same load instruction. In the case of a nested loop, if we know the branch PC corresponding to the outer loop and the branch PC corresponding to the inner loop, we can count the number of inner branch PCs within two outer branch PCs in the LBR to compute the number of inner loop iterations. For instance, in Figure 3, the average loop execution time of `I` is 2.2 and the average loop trip count of `I` is 2.5.

3.2 Determining the Optimal Prefetch Distance

As we describe in §3.1, measuring the average loop iteration time is insufficient for determining the optimal prefetch-distance. In particular, we need to predict the loop's instruction component (IC) execution time under the assumption that memory blocks corresponding to all memory accesses have already been prefetched and that they can be served with low latency. During the profiling step of our technique, we have not injected any prefetches yet and hence this information is unavailable. To address this challenge, we profile the *latency distribution* of delinquent loads (loads that cause frequent LLC misses) [39], instead of solely measuring their average memory access time, resulting in the following application profiling technique.

First, we capture delinquent load PCs that induce frequent Last Level Cache (LLC) misses utilizing precise event-based sampling (PEBS) [71, 119]. Second, we capture LBR samples at the default frequency of once per millisecond while executing the application. Third, we search all LBR samples that contain a delinquent load PC. In particular, the load PC must be greater or equal to the start PC of a BBL and smaller than the terminating branch PC of the BBL (as provided by PC and target information of two consecutive branch entries in the LBR). Fourth, for all LBR samples that contain at least two instances of the BBL containing the delinquent load, we measure the loop execution time by subtracting the cycle counts of the two subsequent branches. Fifth, we analyze the latency distribution of the loop's execution time to predict the latency in the case that the load is served from the L1 or L2 cache.

Figure 4 shows a distribution plot of the execution time of a loop containing the delinquent load PC as used in the graph benchmarks evaluated in (§4). The plot shows four peaks at around 80, 230, 400, and 650 cycles. As the execution time of non-load instructions is relatively stable across loop iterations, we posit that these peaks are caused by loads being served from different levels of the memory hierarchy such as the L1, L2, LLC, and DRAM. From this data we derive that $IC\_latency = 80$ cycles and $MC\_latency = 650 - IC\_latency = 570$ cycles. According to Equation (1) this allows us to compute the $prefetch\_distance = 570/80 \approx 7$.

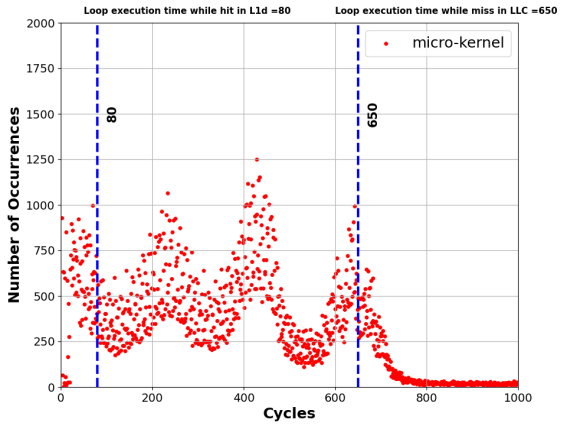

Figure 4. Distribution of a loop's execution time containing a delinquent load [39] in terms of CPU cycles measured using LBR samples

3.3 Finding the Optimal Prefetch Injection Site

As we demonstrate in §2.4, when loops have low trip counts and low execution time per iteration, prefetch instructions inserted in the inner loop could not provide any performance benefit. We extend *APT-GET*'s LBR-based analytical technique described in §3.2 to identify such inner loops and inject prefetch instructions into the outer loop instead of the inner loop. This optimization enables *APT-GET* to prefetch ahead and improve prefetch timeliness. To determine whether to inject prefetches in the outer or inner loop, *APT-GET* analyzes the recorded LBR samples and determines the average trip count of the inner loop (*e.g.*, 2.2 in the example shown in Figure 3). Then, *APT-GET* injects prefetch instructions into the outer loop instead of in the inner loop only if the following equation holds.

$$loop\_trip\_count \times k < prefetch\_distance \quad (2)$$

In Equation 2, k represents a constant and *APT-GET* determines its value based on the loop characteristics. Every loop, where prefetch instructions are injected, contains a prologue and epilogue of size prefetch-distance (in iterations) in which prefetching does not occur. No prefetches are executed for the loads in the prologue and the prefetches performed in the epilogue will not match any corresponding demand load. As a result, if we want to prefetch 80% of all demand loads, the value of k needs to be 5. If *APT-GET* determines to inject prefetches in the outer loop, the prefetch-distance will be computed on the execution latency distribution of the *outer*

loop as described in Section 3.2. To implement prefetching in the outer loop, we extend our LLVM pass to extract the load-slice in the inner loop and replicate it into the outer loop. Furthermore, the induction variable of the outer loop which is considered a constant from the perspective of the inner loop (and the extracted load-slice), needs to be multiplied with the prefetch-distance to form the final prefetching instruction sequence. Next, we provide *APT-GET*'s further implementation details.

3.4 Automated Profiling Methodology

APT-GET performs a fully automated approach consisting of the following steps to generate timely software prefetch instructions. First, *APT-GET* utilizes `perf record` [43, 71] to detect frequent cache miss inducing loads and derives the start PCs of their basic blocks. Second, *APT-GET* captures application's LBR profiles and filters them for the PCs determined in the previous step. To compute prefetch-distance and prefetch injection site, *APT-GET* extracts the average loop trip count by counting the number of consecutive inner loop PCs in the LBR record. It furthermore, computes the average iteration execution time from the cycle counts in the LBR record. Therefore, *APT-GET* obtains the peaks in the scatter plot as they represent the execution time of the BBL when the delinquent load PC is served from a level in the memory hierarchy. For detecting the peaks inside the plot automatically, *APT-GET* uses `find_peaks_cwt` [2, 45] of `scipy.signal` [1], which performs a continuous wavelet transform algorithm to find the location of the peaks. The result of our automated approach is a list of delinquent load PCs with their corresponding prefetch-distance and prefetch injection site which can be consumed by the LLVM software prefetching pass.

3.5 LLVM Prefetch Injection Pass

We implement a function level LLVM pass that detects indirect memory access patterns inside the IR of an application and injects prefetching kernels based on the generated list of delinquent loads. To convert a delinquent load PC to an instruction in the IR, *APT-GET* utilizes AutoFDO's [31] capability of converting arbitrary PCs into lines of code in the IR. Algorithm 2 describes an overview of our implemented LLVM pass for software prefetching of indirect memory access patterns. During the initialization of our algorithm, Lines 3-7, it scans through every function in the module and checks whether there are samples related to functions inside the application IR or not. If it finds at least one sample, we can use the input profile to find precisely the delinquent load PC inside the IR. In this case, the algorithm sets `AutoFDOMapping` variable to True. If the algorithm doesn't find any sample during initialization, Lines 35-38, it performs the same static searching scheme as proposed by Ainsworth & Jones [9] to traverse through all BBLs inside each function

Algorithm 2. An overview of proposed profile-guided LLVM pass for software prefetching

```
1  // REQUIRE input profiling file
2  bool AutoFDOMapping;
3  doInitialization(Module &M){
4    for (F: M):
5      if(SamplesFound):
6        AutoFDOMapping=true;
7  }
8  runOnFunction(Function &F) {
9    modified = false;
10   //vectors to keep candidate loads and their
11   //prefetch-distance for emitting prefetches
12   SmallVector<Instruction*,30> prefetches;
13   SmallVector<Instruction*,30> prefetchDists;
14   if(AutoFDOMapping):
15      //Map the PC to the correct load instruction
16      //inside the IR
17      for(curLoad: BBL):
18         HintsFound =FoundHints(curLoad,SamplesFound);
19         if(HintsFound):
20            for(S: HintsFound)
21              prefechDist = S.second;
22              if(SearchAlgorithm(curLoad, SetOfPhiNodes,
                      SetOfPhiLoads, SetOfInstrs)):
23                prefetches.push_back(curLoad);
24                prefetchDists.push_back(prefechDist);
25      for(p: prefetches):
26         if(SetOfPhiNodes[p].size()>1):
27            //The load is located inside a nested
28            //loops
29             if(InjectPrefechesMorePhis(p, prefechDists[p])):
30                 modified =true;
31         else:
32            //The load is located inside a single
33             //loop
34             if(InjectPrefechesOnePhi(p, prefechDists[p])):
35                 modified =true;
36    else:
37     //Search BBLs statically to emit prefetches
38     //for all indirect memory patterns
39    return modified;
40 }
```

to capture all load instructions with indirect memory access patterns for generating prefetch-slices.

If there exists an AutoFDO mapping for a function containing a delinquent load PC (line 14-35), the algorithm traverses the BBLs inside the function to capture the delinquent load PCs identified inside the LBR sample. In Line 18, the `FoundHints` function compares the debugging location of each load instruction inside the BBLs with the profiling information in the sample to find the precise delinquent Load PC's location. When a delinquent load PC is found inside the IR the algorithm reads the sample to capture the corresponding calculated prefetch-distance from LBR analysis for the load instruction (lines 18-20).

In Lines 22-24, the algorithm calls the `SearchAlgorithm` function, which is a load slice search function similar to `Depth-First Search (DFS)` algorithm proposed by Ainsworth & Jones [9]. This function extracts the load slices of the load instructions by performing a backward data dependency analysis while tracking all instructions that form the load slice. The search terminates when all loop induction variables (`PHINode`), that the load is dependent have

been found. We extend Ainsworth & Jones's algorithm by continuing to search for backward-dependent instructions after the first induction variable is found for the purpose of enabling outer-loop prefetch injection. In particular, we explore the previous BBL and if the two BBLs implement a nested loop, we determine the induction variable of the outer loop by extending the prefetch slice to contain both induction variables enabling injection into both the inner and/or outer loop.

After capturing load slices determined by the `PHINodes` of delinquent load instructions (lines 25-35), the algorithm executes the corresponding prefetching function to inject the prefetch-slice into the IR of the application. If the number of captured `PHINodes` for a delinquent load PC is more than one, it means that the load is located inside a nested loop and algorithm calls the `InjectPrefechesMorePhis` function, otherwise, the algorithm calls the `InjectPrefechesOnePhi` function for inserting the slice. For generating the prefetch slice, the captured load slice is replicated while replacing the delinquent load instruction with a prefetch instruction and adding the calculated prefetch-distance from LBR analysis to the BBL induction variable.

Listing 3 illustrates the simplified IR representation of the nested loop with an indirect memory access pattern as shown in the microbenchmark code 1.The indirect pattern load is located in line 13 and is dependent on the instructions in lines 7-12 as well as on the inner loop induction variable %iv2 in line 6. It is also dependent on the outer loop induction variable %iv1 in line 2 and the instruction in line3. For this code, we can inject the prefetch instructions inside the inner loop for the indirect pattern load by replicating lines 2-7. In this case the outer loop induction variable `%iv1` is considered a constant for all executions of the inner loop. The highlighted lines, line 13-21, in Listing 4 illustrates the IR representation of the microbenchmark 1 after injecting the prefetch slice for the indirect pattern load inside the inner loop. In line 9, the calculated prefetch-distance value by *APT-GET* is added to the inner loop induction variable, `%iv2`, to generate timely prefetch instructions. If we want to inject prefetches into the outer loop, `%iv1` is no longer considered a constant, hence, we need to extend our prefetch-slice by following dependencies across the inner BBL including `%iv1`. Note that, from the outer loop perspective, the inner loop induction variable `%iv2` is unknown (it depends on the inner loop iteration). As a result, we assign `%iv2` to 0, line 6, so that only the first inner loop iteration is prefetched in the outer loop. To improve coverage, we can emit multiple prefetches where `%iv2` is swept from 0 to the average number of inner loop iterations as observed in §2 using LBR-based profile.

To further generalize our technique, our pass introduces the capability for detecting indirect pattern loads that have non-canonical-type induction variables. In particular, our pass supports arbitrary computation on the loop induction variable such as `i*=2` instead of just allowing `i++`. We also add support for multiple and complex exit conditions to break out of a loop such as `for(i:K){if(cond(i)) break;}`.

```
1  for.body1:
2    %iv1 = phi i64
3    %1 = getelementptr inbounds i32, i32* %BO, i64 %iv1
4    [....]
5  for.body2:  ; preds = %for.body1
6    %iv2 = phi i64
7    %2 = load i32, i32* %1
8    %3 = getelementptr inbounds i32, i32* %BI, i64 %v2
9    %4 = load i32, i32* %3
10   %5 = add i32 %2, %4
11   %6 = sext i32 %5 to i64
12   %7 = getelementptr inbounds i32, i32* %T, i64 %6
13   %8 = load i32, i32* %7
14   [....]
```

Listing 3. The simplified LLVM's IR-level representation of the microbenchmark 1 before injecting the prefetch slice

```
1  for.body1:
2    %iv1 = phi i64
3    %1 = getelementptr inbounds i32, i32* %BO, i64 %iv1
4    [....]
5  for.body2:  ; preds = %for.body1
6    %iv2 = phi i64
7    %2 = load i32, i32* %1
8    %3 = getelementptr inbounds i32, i32* %BI, i64 %v2
9    %4 = load i32, i32* %3
10   %5 = add i32 %2, %4
11   %6 = sext i32 %5 to i64
12   %7 = getelementptr inbounds i32, i32* %T, i64 %6
13   %9 = add i64 %iv2, prefetch_distance
14   %10 = icmp slt i64 %INNER, %9
15   %11 = select %10, i64 %INNER, i64 %9
16   %12 = getelementptr inbounds i32, i32* %BI, i64 %11
17   %13 = load i32, i32* %12
18   %14 = add i32 %2, %13
19   %15 = sext i32 %14 to i64
20   %16 = getelementptr inbounds i32, i32* %T, i64 %15
21   %17 = bitcast i32* %16 to i8*
22   call void @llvm.prefetch.p0i8(i8* %17, i32 0, i32 3, i32 1)
23   %8 = load i32, i32* %7
24   [....]
```

Listing 4. The simplified LLVM's IR-level representation of the microbenchmark 1 after injecting the prefetch slice inside the inner loop

3.6 Limitations of *APT-GET*

Our proposed technique has a number of limitations. However, none of these limitations has shown to be a significant issue in practice. The first two limitations are due to the limited size of the LBR containing only 32 entries on our system. In the case of a two-nested loop, where the inner loop containing the delinquent load has a high loop trip count, LBR samples will only contain the branch PC implementing the inner loop. As a result, we cannot measure the outer loop latency. This is not a real problem because with high loop trip counts we can always prefetch in the inner loop and do not have to revert to outer loop prefetch injection.

Another potential scenario limiting *APT-GET*'s effectiveness can occur when the inner loop containing the delinquent load also contain 32 other taken branches. Consequently, LBR samples contain the inner loop branch PC only once prohibiting latency measurements. In this case, the loop execution time is generally high enough so that a default prefetch-distance of one is sufficient.

Lastly, if the execution time of a loop is input data-dependent, we need to re-profile the application for each input. This means that in contrast to static compile-time techniques, *APT-GET* also allows optimizing input-dependent code. We believe re-profiling is feasible in this case, especially in the data center setting where profile-guided optimization techniques have been most successful [15, 64–67, 79, 96, 112] and where applications are compiled and released at high cadence. Furthermore, AutoFDO [31] has shown that even stale profiles enable PGO techniques to provide good performance as data inputs tend to change slowly over the course of multiple weeks.

4 Evaluation

In this section, we describe the software infrastructure, test setups, real world benchmarks and data sets that we use to evaluate *APT-GET*.

4.1 Experimental Setup

The techniques described in Section 3 are implemented as a function level LLVM pass [72] that is available at [5]. We utilize the Clang compiler, version 10.0, on Ubuntu Linux 20.04 with kernel version 5.4 to apply the pass on the IR representation of the applications. We enable the highest compiler-level optimizations (`-O3`). We use (`-gmlt`) and (`-fdebug-info-for-profiling`) to emit debugging information for identifying delinquent load PCs inside the pass [3]. We compare *APT-GET* against a no-prefetching baseline and against the static prefetch injection technique Ainsworth & Jones [9]. We execute each experiment three times and utilize `perf stat` [43, 71] to obtain the results presented in this section. The machine configuration of the evaluated system is described in Table 2.

Table 2. The Machine Configuration

Component	Parameters
Core	Intel(R) Xeon(R) Gold 5218 CPU @2.30GHz (3.9GHz Turbo)
L1 I/D Cache	64KiB/core
L2 Cache	1MiB/core
LLC	22MiB shared
Main Memory	DIMM DDR4 capacity: 32GiB, channels: 6, @2666MHz

Table 3. The list of real-applications

App	Description
BFS	Searches a target vertex given a start node in a graph
DFS	Searches a target vertex by performing a depth-first traversal given a start node
PR	Computes ranking of web-pages
BC	A measure of centrality computed by finding all the shortest paths between all vertices
SSSP	Computes the shortest path to all vertices given a source vertex
IS	Bucket sorting of random integers
CG	sparse matrix multiplications
RandAcc	Measuring memory system performance
HJ2/HJ8	Represents a database application
Graph500	Breadth-first search on an undirected graph

Table 4. Graph data-sets properties

Data-set Name	#Vertices	#Edges
web-Google (WG)	875,713	5,105,039
p2p-Gnutella31 (P2P)	62,586	147,892
roadNet-CA (CA)	1,965,206	2,766,607
roadNet-PA (PA)	1,088,092	1,541,898
loc-Brightkite (LBE)	58,228	214,078
web-BerkStan (WB)	685,230	7,600,595
web-NotreDame (WN)	325,729	1,497,134
web-Stanford (WS)	281,903	2,312,497

4.2 Methodology

Our proposed technique is generic and automated; hence it can in principle be applied to any application that is performance limited by LLC cache misses. For evaluating *APT-GET*, we examine ten real-world, memory-bound applications as described in Table 3. In accordance with prior work [9], applications were selected for exhibiting indirect memory accesses located inside loops (including nested loops) as hardware prefetchers do not handle those. Figure 5 shows how much the selected applications are bounded by the L3 cache and DRAM. In particular, we evaluated graph full programs including **breadth-first search (BFS)**, **depth-first search (DFS)**, **pageRank (PR)**, **betweenness centrality (BC)**, and **single-source shortest path algorithm (SSSP)** from the CRONO benchmark suite [7]. Traversing graph data structures frequently requires executing indirect memory access patterns inside a nested loop where the loop trip counts depends on the number of graph vertices or edges. For running these applications, we use both real-world graph data-sets from the Stanford Network Analysis Platform (SNAP) [77] as well as synthetic graphs that exhibit

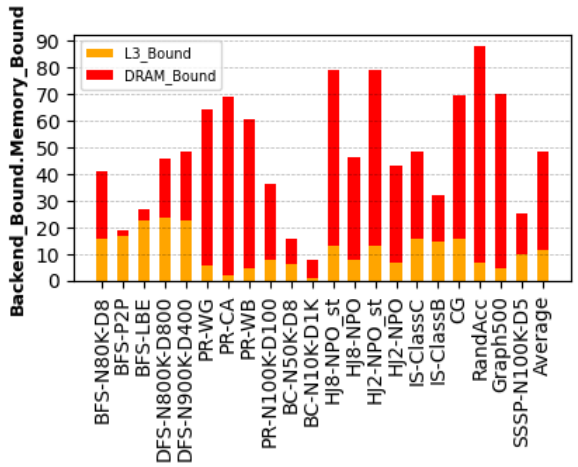

Figure 5. The percentage of L3/DRAM stalls is obtained for each application's non-prefetching baseline. On average, the performance of selected applications is 49.37% bounded by the memory system.

a sufficiently large number of vertices and edges per vertex to induce high Misses Per Kilo Instructions (MPKI) in the Last Level Cache (LLC) for the non-prefetching baseline configuration. The utilized graph data sets are shown in Table 4.

We evaluate additional applications including **Integer Sort (IS)** and **Conjugate Gradient (CG)** applications from the NAS parallel benchmark suite [16]. The integer sort algorithm induces indirect memory access patterns during the bucket sorting process of random integers inside an array. For evaluating IS we set the number of the iterations to 25 using two different problem sizes, Class B and Class C. The CG benchmark executes sparse matrix multiplications generating irregular indirect memory accesses for traversing vectors in the compressed sparse row (CSR) format. We also evaluate the **RandomAccess (randAccess)** benchmark from HPC Challenge Benchmark Suite[85] which randomly updates elements of a large table to compute the giga updates per second (GUPS) memory system performance. We utilize a table size of 1GiB for the RandomAccess benchmark suite.

We also use two different forms of the Hash join benchmark [19], **Hash Join 2EPB (HJ2)** and **Hash Join 8EPB (HJ8)**, to evaluate the performance of *APT-GET*. Both Hash Join benchmarks implement hashing to lookup target values based on some key distribution, where HJ8 utilizes hash buckets of 8 elements while HJ2 just utilizes 2 elements per bucket. The size of hash table is 970 MiB and we run both of these benchmarks with two hasing algorithms (NPO and NPO_st). The **Graph500** benchmark [92] executes breadth-first search on an undirected graph that has an average degree of 16. We utilize a scale factor of 22 and an edge-factor of 10.

4.3 Performance Improvement

Figure 6 shows the execution time speedup of *APT-GET* and Ainsworth & Jones [9] across all benchmarks over the no-prefetching baseline. Ainsworth & Jones utilizes a static prefetch-distance and, furthermore, is limited to injecting prefetches into the inner loop. As can be seen, *APT-GET* provides a maximum speedup of 1.98× for HJ8 and BFS and an average speedup of 1.30× over the baseline while Ainsworth & Jones only provides gains for few applications resulting in an average speedup of 1.04×. We calculate the average speedup values by using the geometric mean. Except for the CG benchmark, *APT-GET* improves performance for all applications whereas Ainsworth & Jones shows a performance regression for BC. This shows that injecting prefetches with non-optimal prefetch-distance and prefetch injection site can reduce performance due to the injected instruction overhead.

4.4 Cache Miss Reduction

In Figure 7 we show the cache miss reduction enabled by *APT-GET* measured in misses per kilo instructions (MPKI). We measure misses utilizing the `offcore_requests.demand_data_rd` PMU counter and compare against the non-prefetching baseline. Note that demand loads that hit a prefetch to the same address in the fill buffer are counted as a cache miss. In average, *APT-GET* reduces cache misses by 65.4%, while Ainsworth & Jones reduces cache misses by 48.3% and MPKI improvements are most significant where *APT-GET* also provides the highest execution time benefits. We leave analyzing the interplay of hardware and software prefetches for future work. While for BC with 50K nodes and degree of 8, Ainsworth & Jones reduces cache misses over *APT-GET* it also injects significantly more instructions as we will show in Figure 11 explaining the higher execution time improvement of *APT-GET*. To provide further insights, in the following sections, we evaluate the individual techniques of *APT-GET*.

4.5 Effectiveness of the LBR Profiling

We first evaluate the effectiveness of our LBR profiling technique to determine the optimal prefetch-distance. Therefore, we execute all benchmarks with the following 8 different prefetch distances $D = \{1, 2, 4, 8, 16, 32, 64, 128\}$. We then take the prefetch-distance that performed best and compare the achieved performance against *APT-GET*. As shown in Figure 8, the proposed LBR approach is able to determine a near optimal prefetch-distance for all applications.

4.6 Effectiveness of the Prefetch Distance Optimization

In figure 9, we evaluate the effectiveness of our proposed LBR sampling technique against using a static prefetch-distance approach. We compare the speedup results for *APT-GET* against 3 different static prefetch-distance values of 4, 16, 64,

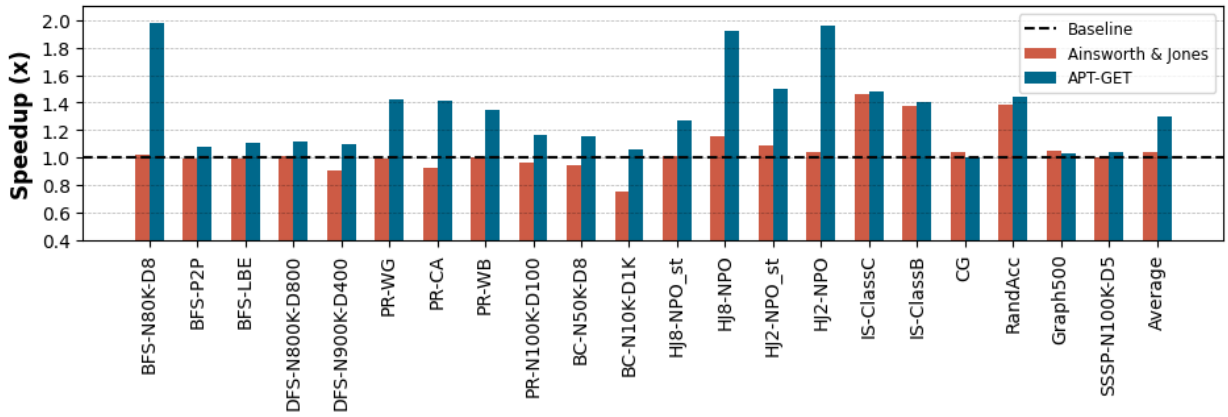

Figure 6. Execution time speedup provided by *APT-GET* over the non-prefetching baseline: *APT-GET* achieves 1.30× average speedup on average, compared to the 1.04× speedup provided by the state of the art (Ainsworth & Jones).

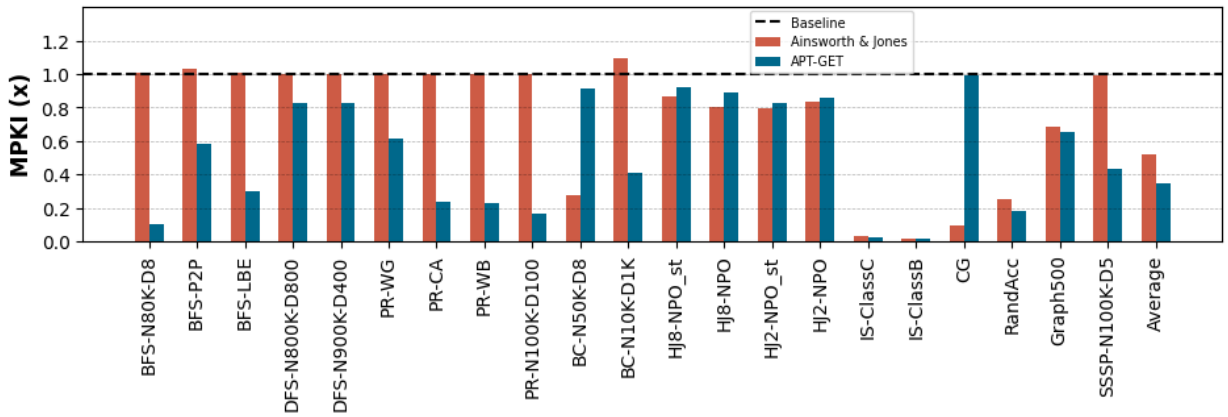

Figure 7. *APT-GET*'s LLC MPKI, misses per 1000-instructions, reduction over the non-prefetching baseline (lower is better): on average, *APT-GET* provides 1.35× greater MPKI reduction than the state of the art (Ainsworth & Jones).

and the calculated prefetch-distance value from LBR samples for each workload. As we can see, for most of the applications *APT-GET* achieves a higher performance gain by using the calculated prefetch-distance from LBR samples than utilizing a static prefetch-distance of 4, 16, 64, highlighting the efficacy of our approach. While a static prefetch-distance of 64 also performs well it is outperformed by *APT-GET* by 1.06× for RandomAccess and by 1.09× for HJ2-NPO.

4.7 Effectiveness of the Prefetch Injection Site Optimization

Figure 10 shows the effect of optimizing the prefetch injection site for all applications. In this experiment, we measure the speedup of *APT-GET* by prefetching either in the outer or the inner loop for all applications that contain nested loops. The goal of these experiments is to evaluate the effectiveness of our proposed approach for detecting the appropriate prefetch injection site. For all the workloads except for DFS, injecting prefetch instructions inside the inner loop decreases the performance over the non-prefetching baseline. Therefore, it is crucial to enable outer-loop prefetching but also to detect the appropriate prefetch injection site for each prefetch individually. We can see that based on the number of edges and vertices the input graph contains, the achieved performance gain from inner or outer loop prefetching can be significantly different. For example, if we compare the achieved speedup for the BFS application by considering two different inputs, loc-Brightkite with 58K nodes and an average edge degree of 3, as well as graph with 80K nodes and an average edge degree of 8, we can see that the achieved speedup gain from outer loop prefetching differs significantly.

4.8 Instruction Overhead

Injecting prefetch slices introduces overheads in terms of additional instructions that need to be executed by the

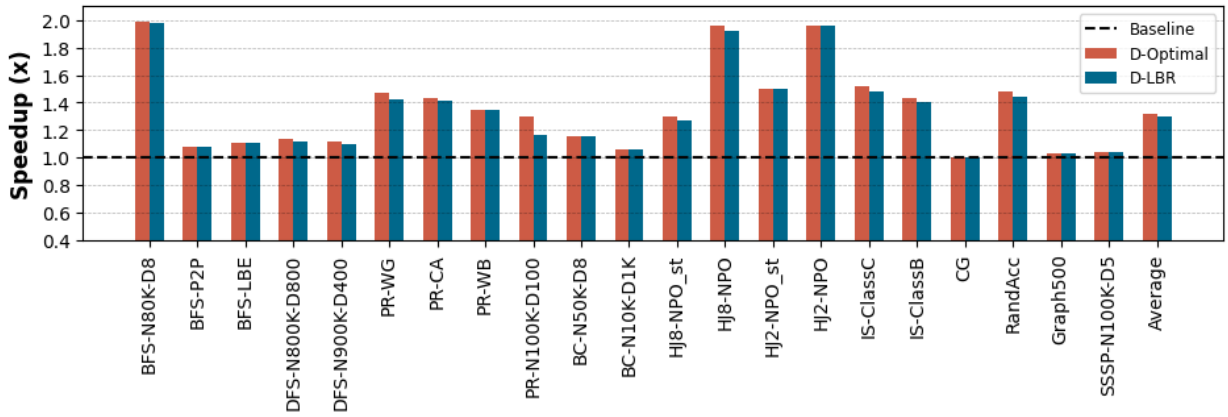

Figure 8. Speedup of prefetch-distance from LBR sampling technique and optimal prefetch-distance over non-prefetching baseline: LBR sampling technique achieves 1.30× overall speedup in average, compared to 1.32× speedup of optimal prefetch-distance, over non-prefetching baseline.

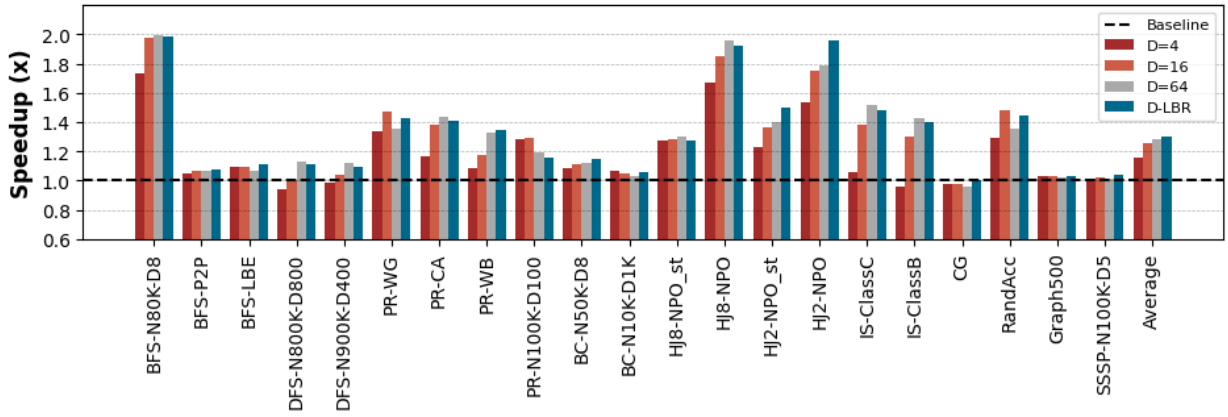

Figure 9. Speedup for different static offset values and LBR over non-prefetching baseline: prefetch-distance of 4, 16, 64, and LBR sampling technique achieve 1.16×, 1.26×, 1.28×, and 1.30× speedup in average over non-prefetching baseline, respectively.

processor. In particular, while prefetching almost always improves the instructions per cycle (IPC) performance of an application it increases the instruction count potentially offsetting the performance gains. While we have already shown that *APT-GET* provides a net performance gain, Figure 11 provides insight about the number of instructions injected by *APT-GET* and Ainsworth & Jones. While for most applications the instruction overhead is negligible, for IS and RandomAccess, it is significant which limits the performance gains provided for these applications. We believe there exist future research opportunities in considering the instruction overhead for conditional prefetch slice injection.

4.9 Inputs for profiling and testing

We use different realistic data sets (such as graphs) for evaluating the input sensitivity of *APT-GET*. In particular, Figure 12 shows the performance of training and evaluating *APT-GET* on the same input (TRAIN-DATA) vs. evaluating on a different input (TEST-DATA). The obtained results indicate that there are no significant performance differences between the two inputs showing that *APT-GET* can generalize across inputs.

4.10 Profiling overhead

In Google's data centers, all applications are already continuously profiled [15, 102] and recompiled before deployment. Our technique does not introduce additional overheads here. Our approach only requires a single profiling run. The average total overhead is less than 15-20 seconds. Note that optimization is performed only once in data centers while the application is executed on 1000s of nodes.

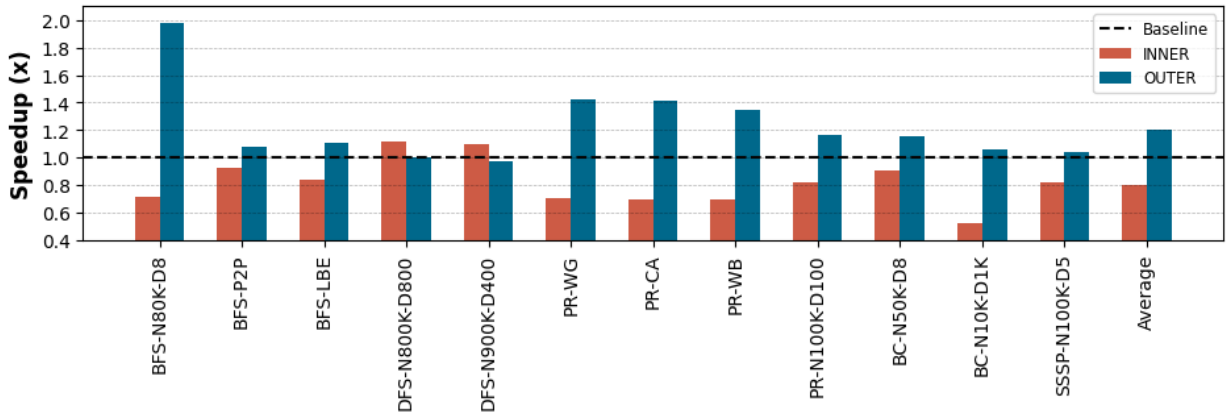

Figure 10. Speedup of injecting prefetches inside the outer or inner loops over non-prefetching baseline: For most of the applications, injecting prefeches inside the outer loop achieves 1.20× overall speedup in average, while injecting prefetches inside the inner loop improves speedup for DFS up to 1.11× over non-prefethcing baseline.

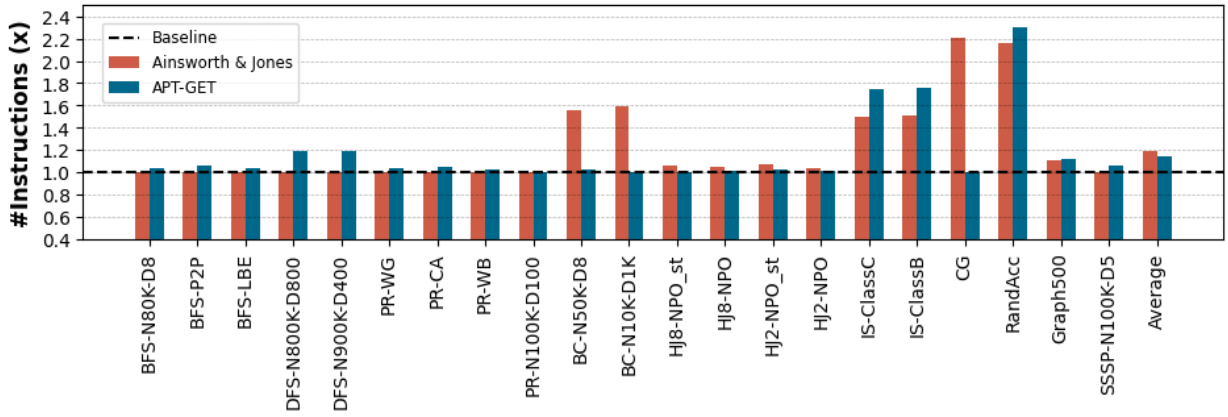

Figure 11. The overhead of injected prefetching instructions over non-prefetching baseline: *APT-GET* increases the total number of instructions 1.14× in average, compared to 1.19× in average of Ainsworth & Jones, over non-prefetching baseline.

5 Related Work

Prefetching is a well-studied and yet widely open area that spans many types of access patterns and implementations. We classify prior works in three categories. **Software prefetching.** Traditional software prefetching [26] techniques utilize compilers to perform static code analysis to generate fixed prefetch targets [11, 35, 53, 111]. These approaches are limited by lacking knowledge about which memory accesses actually cause performance degradation and which prefetch distances should be used. Furthermore, they are limited by practical constraints such as the ability to only detect simple patterns such as Singly-Nested Loop Nests [120] or strides [63, 91, 124]. Other approaches can analyze more complex behaviors like linked list traversals and insert jump pointers into source code at compile time [38, 83, 104, 105]. However, these require source code modification and result in additional run-time storage costs whenever a pointer is inserted into a data structure. In contrast, *APT-GET* does not rely on source code modification, it can handle arbitrarily complex indirect and direct access patterns, and uses profiling information to identify performance-critical loads and tune prefetch distances to improve prefetch timeliness.

Improving upon static methods, some prior works utilize dynamic profiling to better identify prefetch candidates [81, 84], but they don't utilize profiling information to improve timeliness, or introduce overheads by requiring to executed a separate prefetching thread in parallel to the workload [131].

Hardware prefetching. Stream prefetchers [56, 106] and pattern-based prefetchers [42, 59, 68, 69, 88, 95, 100, 107, 109] can be implemented with low to moderate hardware complexity and are capable of prefetching strides and other simple access patterns. Spatial [18, 21, 51, 110] and temporal [17, 28, 55, 60, 117] prefetchers can learn and replay more

Figure 12. Execution time speedup provided by *APT-GET* over the non-prefetching baseline for different inputs as train/test data: *APT-GET* achieves 1.36× average speedup on average for test data sets compared to the 1.39× average speedup obtained for train data sets

complex memory access patterns, but they require costly on-chip storage and rely upon highly-recurrent access patterns. None of these prefetchers is well-suited for large instruction footprint applications exhibiting many irregular and indirect memory patterns such as pointer-based traversals.

Several hardware mechanisms have been proposed to prefetch complex memory access patterns that are based on the data and control flow of an application [12, 39, 46, 54, 75, 82, 93, 94, 101, 132]. While general-purpose in nature, they require fast and complex hardware resources such as helper threads to run ahead of the application and prefetch upcoming memory accesses. Unlike *APT-GET*, these techniques cannot easily filter relevant address calculations from the main application and their high cost and complexity is often better spent on additional CPU cores.

Hybrid hardware-software prefetching. Hybrid hardware-software prefetching mechanisms [10, 20, 66, 114, 129] attempts to combine the best of both worlds while also addressing the limitations of hardware-only and software-only mechanisms. However, these techniques require both hardware prefetching support and software programming model, neither of which exists in today's processor. In contrast, *APT-GET* employs hardware and software interfaces already available in today's processors, identifies the key limitation of software-managed prefetching in terms of prefetch timeliness, and proposes a profile-guided low-overhead mechanism to ensure the timeliness. Hence, *APT-GET* can be readily be employed on existing processors to optimize the performance of real-world applications.

6 Conclusion

We propose a novel automated prefetch injection technique for reducing stall cycles in memory latency-bound programs. Our approach provides performance gains of up to 1.98× and 1.30× in average over state-of-the-art mechanisms. To motivate our technique, we first study prior software-based prefetching mechanisms for indirect memory access patterns that cannot be handled by existing hardware prefetchers. This analysis reveals that, while existing techniques provide high accuracy and coverage, due to their *static* nature, they are unable to generate timely prefetches. To address this challenge, we propose a profile-guided optimization technique utilizing the last branch record capability of contemporary microprocessors. Our approach, implemented as an LLVM compiler pass, enables a detailed characterization of memory latency-bound loops, enabling timely prefetching of the performance limiting loads. We believe that this work can establish *dynamic* prefetch injection as a generic, efficient, low-overhead compiler technique.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback and suggestions. This work was supported by Google and NSF grants #1942754 and #2010810, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We thank Anant Nori and Ahmad Yasin from Intel for their helpful discussions and feedback.

References

[1] 2008. scipy.signal. https://docs.scipy.org/doc/scipy/reference/signal.html

[2] 2008. scipy.signal.find_peaks_cwt. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks_cwt.html

[3] 2018. Support for inserting profile-directed cache prefetches. https://reviews.llvm.org/D54052

[4] 2021. llvm-mca - LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html. [Online; accessed 9-October-2021].

[5] 2022. Profile Guided Software Prefetching. https://github.com/SabaJamilan/Profile-Guided-Software-Prefetching

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[7] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 44–55.

[8] Sam Ainsworth and Timothy M Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–11.

[9] Sam Ainsworth and Timothy M Jones. 2017. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 305–317.

[10] Sam Ainsworth and Timothy M Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. *ACM Sigplan Notices* 53, 2 (2018), 578–592.

[11] Hassan Al-Sukhni, Ian Bratt, and Daniel A Connors. 2003. Compiler-directed content-aware prefetching for dynamic data structures. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 91–100.

[12] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. 2001. Data prefetching by dependence graph precomputation. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 52–61.

[13] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 643–656.

[14] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.

[15] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 462–473.

[16] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0*. Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[17] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 131–142.

[18] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 399–411.

[19] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

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

[21] Rahul Bera, Anant V Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. Dspatch: Dual spatial pattern prefetcher. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 531–544.

[22] Ketan Bhardwaj, Matt Saunders, Nikita Juneja, and Ada Gavrilovska. 2019. Serving mobile apps: A slice at a time. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[23] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[24] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. 2018. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 316–331.

[25] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[26] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. ACM, New York, NY, USA, 40–52. https://doi.org/10.1145/106972.106979

[27] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality. *ACM SIGPLAN Notices* 29, 11 (1994), 252–262.

[28] Chandranil Chakraborttii and Heiner Litz. 2020. Learning i/o access patterns to improve prefetching in ssds. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 427–443.

[29] Andres S Charif-Rubial, Emmanuel Oseret, José Noudohouenou, William Jalby, and Ghislain Lartigue. 2014. CQA: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 1–10.

[30] Mark J Charney and Anthony P Reeves. 1995. *Generalized correlation-based hardware prefetching*. Technical Report. Technical Report EE-CEG-95-1, Cornell University.

[31] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 12–23.

[32] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2004. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering*. 116.

[33] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. 2001. Improving index performance through prefetching. *ACM SIGMOD Record* 30, 2 (2001), 235–246.

[34] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[35] William Y Chen, Scott A Mahlke, Pohua P Chang, and Wen-mei W Hwu. 1991. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *MICRO*, Vol. 24. 69–73.

[36] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sýkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 167–177.

[37] Trishul M Chilimbi and Martin Hirzel. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 199–209.

[38] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M Tullsen. 2002. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 62–73.

[39] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 14–25.

[40] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A stateless, content-directed data prefetching mechanism. *ACM SIGPLAN Notices* 37, 10 (2002), 279–290.

[41] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197.

[42] Fredrik Dahlgren and Per Stenström. 1995. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors.. In *hpca*. 68–77.

[43] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[44] Jialin Dou and Marcelo Cintra. 2007. A compiler cost model for speculative parallelization. *ACM Transactions on Architecture and Code Optimization (TACO)* 4, 2 (2007), 12–es.

[45] Pan Du, Warren A Kibbe, and Simon M Lin. 2006. Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *bioinformatics* 22, 17 (2006), 2059–2065.

[46] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *International Conference on Supercomputing*. Citeseer, 68–75.

[47] Stephane Eranian. 2021. Add AMD Fam19h Branch Sampling support. https://lwn.net/Articles/875869/.

[48] Babak Falsafi and Thomas F Wenisch. 2014. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture* 9, 1 (2014), 1–67.

[49] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2019. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[50] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[51] Michael Ferdman, Stephen Somogyi, and Babak Falsafi. 2009. Spatial memory streaming with rotated patterns. *1st JILP Data Prefetching Championship* 29 (2009).

[52] Google. 2020. Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller.

[53] Edward H Gornish, Elana D Granston, and Alexander V Veidenbaum. 2014. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 128–142.

[54] Milad Hashemi, Onur Mutlu, and Yale N Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 61.

[55] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *arXiv preprint arXiv:1803.02329* (2018).

[56] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 397–408.

[57] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2003. Stride prefetching by dynamically inspecting objects. *ACM SIGPLAN Notices* 38, 5 (2003), 269–277.

[58] Intel Corporation. 2019. Intel Architecture Code Analyzer. https://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer.html. [Online; accessed 9-October-2021].

[59] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13, 2011 (2011), 1–24.

[60] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 247–259.

[61] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.

[62] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[63] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. 2014. A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*. IEEE, 101–110.

[64] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 816–829.

[65] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. DMon: Efficient Detection and Correction of Data Locality Problems using Selective Profiling. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (OSDI 2021)*. USENIX Association.

[66] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[67] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA) (ISCA 2021)*.

[68] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[69] Jinchun Kim, Elvira Teran, Paul V Gratz, Daniel A Jiménez, Seth H Pugsley, and Chris Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices* 52, 4 (2017), 737–749.

[70] Andi Kleen. 2016. An Introduction to Last Branch Records. https://lwn.net/Articles/680985/

[71] Andi Kleen. 2022. GitHub - andikleen/pmu-tools: Intel PMU profiling tools. https://github.com/andikleen/pmu-tools

[72] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[73] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. 2019. Automatic throughput and critical path analysis of x86 and ARM assembly kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 1–6.

[74] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*. IEEE, 121–131.

[75] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. 2009. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems* 20, 9 (2009), 1309–1324.

[76] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 1 (2012), 1–29.

[77] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.

[78] Chuanpeng Li, Kai Shen, and Athanasios E Papathanasiou. 2007. Competitive prefetching for concurrent sequential I/O. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 189–202.

[79] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: Critical Slice Prefetching. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[80] Christianto C Liu, Ilya Ganusov, Martin Burtscher, and Sandip Tiwari. 2005. Bridging the processor-memory performance gap with 3D IC technology. *IEEE Design & Test of Computers* 22, 6 (2005), 556–564.

[81] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. 2003. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 180.

[82] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G Abraham. 2005. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 93–104.

[83] Chi-Keung Luk and Todd C Mowry. 1996. Compiler-based prefetching for recursive data structures. In *ACM SIGOPS Operating Systems Review*, Vol. 30. ACM, 222–233.

[84] Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P Geoffrey Lowney, and Robert Cohn. 2002. Profile-guided post-link stride prefetching. In *Proceedings of the 16th international conference on Supercomputing*. ACM, 167–178.

[85] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Vol. 213. 1188455–1188677.

[86] Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. 2021. Break dancing: low overhead, architecture neutral software branch tracing. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 122–133.

[87] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*. PMLR, 4505–4515.

[88] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 469–480.

[89] Todd Mowry and Anoop Gupta. 1991. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of parallel and Distributed Computing* 12, 2 (1991), 87–106.

[90] Todd C. Mowry, Angela K. Demke, and Orran Krieger. 1996. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*, Karin Petersen and Willy Zwaenepoel (Eds.). ACM, 3–17. https://doi.org/10.1145/238721.238734

[91] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/143365.143488

[92] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[93] Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2005. Techniques for efficient processing in runahead execution engines. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 370–381.

[94] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 129–140.

[95] Kyle J Nesbit and James E Smith. 2004. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 96–96.

[96] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2–14.

[97] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 119–130.

[98] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 335–348.

[99] Lucas Prates. 2020. Add support for the Branch Record Buffer extension. https://reviews.llvm.org/D92389.

[100] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 626–637.

[101] Tanausu Ramirez, Alex Pajuelo, Oliverio J Santana, and Mateo Valero. 2008. Runahead threads to improve SMT performance. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 149–158.

[102] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro* 30, 4 (2010), 65–79.

[103] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. 2020. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 442–455.

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

[105] Amir Roth and Gurindar S Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *ACM SIGARCH Computer Architecture News*, Vol. 27. IEEE Computer Society, 111–121.

[106] Suleyman Sair, Timothy Sherwood, and Brad Calder. 2003. A decoupled predictor-directed stream prefetching architecture. *IEEE Trans. Comput.* 52, 3 (2003), 260–276.

[107] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 141–152.

[108] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 861–873.

[109] Alan Jay Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* 12 (1978), 7–21.

[110] Stephen Somogyi, Thomas F Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 252–263.

[111] Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. 2009. A compiler-directed data prefetching scheme for chip multiprocessors. In *ACM Sigplan Notices*, Vol. 44. ACM, 209–218.

[112] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: Profile-Guided BTB Replacement for Data Center Applications.. In *49th Annual International Symposium on Computer Architecture (ISCA)*.

[113] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[114] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.

[115] Vish Viswanathan. 2014. Disclosure of hardware prefetcher control on some Intel processors. *Intel SW Developer Zone* (2014).

[116] Zhenlin Wang, Doug Burger, Kathryn S McKinley, Steven K Reinhardt, and Charles C Weems. 2003. Guided region prefetching: A cooperative hardware/software approach. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings*. IEEE, 388–398.

[117] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 79–90.

[118] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2005. Temporal streaming of shared memory. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 222–233.

[119] Thomas Willhalm and Roman Dementiev. 2012. Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor#abstracting.

[120] Michael Joseph Wolfe and Michael Wolfe. 1996. *High performance compilers for parallel computing*. Vol. 102. Addison-Wesley Reading.

[121] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 996–1008.

[122] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient metadata management for irregular data prefetching. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–13.

[123] Peng Wu, Arun Kejariwal, and Călin Caşcaval. 2008. Compiler-driven dependence profiling to guide program parallelization. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 232–248.

[124] Youfeng Wu. 2002. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 210–221o.

[125] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[126] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.

[127] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. 2018. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[128] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* (2010).

[129] Chao Zhang, Yuan Zeng, John Shalf, and Xiaochen Guo. 2020. RnR: A software-assisted record-and-replay hardware prefetcher. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 609–621.

[130] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. 2018. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[131] Weifeng Zhang, Brad Calder, and Dean M Tullsen. 2006. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 50–64.

[132] Weifeng Zhang, Dean M Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for efficient prefetching. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 85–95.