**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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

**EuroSys ’22, April 5–8, 2022, RENNES, France**

© 2022 Copyright held by the owner/author(s).


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.

\(^1\)as an appropriate and timely (APT) prefetch (GET)

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.
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 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

<table>
<thead>
<tr>
<th>Prefetch</th>
<th>IPC</th>
<th>Prefetch Accuracy</th>
<th>Late Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0.33</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Dist-1</td>
<td>0.42</td>
<td>72%</td>
<td>95%</td>
</tr>
<tr>
<td>Dist-64</td>
<td>0.73</td>
<td>70%</td>
<td>1%</td>
</tr>
<tr>
<td>Dist-1024</td>
<td>0.29</td>
<td>3%</td>
<td>0%</td>
</tr>
</tbody>
</table>

the performance of the microbenchmark choosing a low work complexity and $\text{INNER} = 256$ 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 ($\text{offcore_requests.all_data_rd}$ - $\text{offcore_requests.demand_data_rd}$) divided by the number of all loads ($\text{offcore_requests.all_data_rd}$). 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 ($\text{LOAD_HIT_PF}$). 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 load demand 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:

\[ IC\_latency \times \text{prefetch\_distance} = MC\_latency \]  

(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.
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 ≈ 7$.

### 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.

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

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 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

```c
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> prefetchDist;
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 prefetchDist = S.second;
22 if(SearchAlgorithm(curload, SetOfPhiNodes, SetOfPhiLoads, SetOfInsts)):
23 prefetches.push_back(curload);
24 prefetchDist.push_back(prefetchDist);
25 for(p: prefetches): prefetches
26 if(SetOfPhiNodes[p].size() > 1):
27 //The load is located inside a nested
28 //loops
29 if(InjectPrefechesMorePhis(p, prefetchDist[p])):
30 modified = true;
31 else:
32 //The load is located inside a single
33 //loop
34 if(InjectPrefechesOnePhi(p, prefetchDist[p])):
35 modified = true;
36 else:
37 //Search BBLs statically to emit prefetches
38 //For all indirect memory patterns
39 return modified;
40 }
```
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 line 3. 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:n){if(cond(i)) break;}.  

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

Table 2. The Machine Configuration
The size of hash table is 970 MiB and we run both of these benchmarks with two hasing algorithms (NPO and NPO_st). HJ2 just utilizes 2 elements per bucket. HJ8 utilizes hash buck-}


table size of 1GiB for the RandomAccess benchmark suite.


text 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 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 1.98x for HJ8 and BFS and an average speedup of 1.30x over the baseline while Ainsworth & Jones only provides gains for few applications resulting in an average speedup of 1.04x. 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 $1.30 \times$ average speedup on average, compared to the $1.04 \times$ 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 $1.35 \times$ 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 $1.06 \times$ for RandomAccess and by $1.09 \times$ 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 $1.30 \times$ overall speedup in average, compared to $1.32 \times$ 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 $1.16 \times$, $1.26 \times$, $1.28 \times$, and $1.30 \times$ speedup in average over non-prefetching baseline, respectively.

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 prefetches inside the outer loop achieves $1.20\times$ overall speedup in average, while injecting prefetches inside the inner loop improves speedup for DFS up to $1.11\times$ over non-prefetching baseline.

Figure 11. The overhead of injected prefetching instructions over non-prefetching baseline: APT-GET increases the total number of instructions $1.14\times$ in average, compared to $1.19\times$ 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 execute 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
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


