Classifying Memory Access Patterns for Prefetching

Grant Ayers*
Stanford University

Heiner Litz*
UC Santa Cruz

Christos Kozyrakis
Stanford University, Google

Parthasarathy Ranganathan
Google

Abstract

Prefetching is a well-studied technique for addressing the memory access stall time of contemporary microprocessors. However, despite a large body of related work, the memory access behavior of applications is not well understood, and it remains difficult to predict whether a particular application will benefit from a given prefetcher technique. In this work we propose a novel methodology to classify the memory access patterns of applications, enabling well-informed reasoning about the applicability of a certain prefetcher. Our approach leverages instruction dataflow information to uncover a wide range of access patterns, including arbitrary combinations of offsets and indirection. These combinations–or *prefetch kernels*–represent reuse, strides, reference locality, and complex address generation. By determining the complexity and frequency of these access patterns, we enable reasoning about prefetcher timeliness and criticality, exposing the limitations of existing prefetchers today. Moreover, using these kernels, we are able to compute the next address for the majority of top-missing instructions, and we propose a software prefetch injection methodology that is able to outperform state-of-the-art hardware prefetchers.

ACM Reference Format:
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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378498

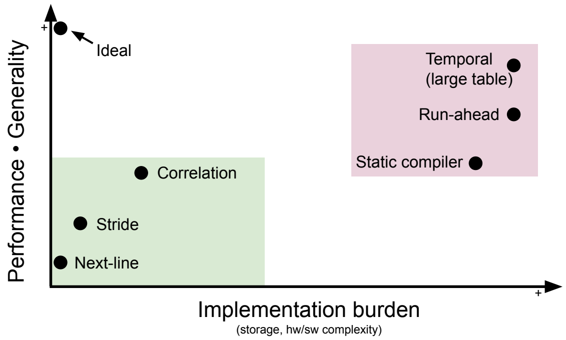


Figure 1. The landscape of prefetcher designs which weighs heavily toward low performance or high cost

1 Introduction

Von-Neuman architectures suffer from the well-known processor-memory performance gap: While processors have enjoyed exponential increases in performance, memory has scaled in terms of bandwidth and capacity but not in access latency. As a result, the cycle time of modern processors is now two orders of magnitude smaller than the access latency of DRAM. One way computer architects have addressed this problem is by employing deep memory hierarchies with small, low-latency caches. However, given that data set sizes are increasing [28] and transistor scaling is slowing down [17, 20], we cannot rely solely on cache capacity scaling. Prefetching works around limited cache capacities by speculatively moving data from slow memory into fast caches in advance, so that later demand loads can access the data from those caches with low latency. This can be an efficient technique as it often requires very few hardware resources. However, it relies on an accurate prediction mechanism and sufficient time to prefetch the correct data elements.

There exists a large body of prior research on prefetchers including stream prefetchers [13, 23, 25, 42, 44], correlation prefetchers [27, 38, 45], and execution-based prefetchers [12, 15, 15, 21, 30, 32, 36, 37, 39, 54] which have shown significant performance gains. Surprisingly however, the prefetchers deployed by contemporary microprocessors such as Intel Xeon, are limited to simple, conservative designs such as next-line and stride prefetchers [48]. This is because, as

*Work performed while these authors were at Google

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03.
https://doi.org/10.1145/3373376.3378498

a whole, the prefetcher proposals in the literature do not capture the trifecta of performance, generality, and cost as shown in Figure 1. Low-cost designs, such as stride prefetchers, are easy to implement but only improve a very small and specific set of access patterns, leaving lots of performance potential on the table. Highly general designs, such as run-ahead prefetchers, promise high performance but come with prohibitive costs such as additional hardware threads or cores. Neither extreme tries to understand the underlying access patterns of the application. Thus, it is still an open problem to design a prefetcher that yields high performance, is general to a wide range of access patterns, and is implementable. As a result, current caches and prefetchers still leave up to 60% of performance on the table in the data center [4, 28].

One longstanding challenge for prefetcher design is that computer architects and software developers generally have only a limited understanding of applications' memory access behaviors. This results in a potpourri of prefetcher designs that are excellent for certain types of workloads or kernels but not for others. As such, it is often unclear whether or not an application is amenable to a given prefetching technique, and in many cases a mismatched prefetcher can even reduce performance by creating useless memory traffic and cache evictions. A successful prefetch design that greatly improves generality while minimizing implementation burden must be made aware of the diverse access patterns of complex applications. Even hardware-based prefetchers must move towards this approach.

To address the challenges above, this paper takes a new approach: Instead of designing a new prefetcher that improves performance for a specific type of access behavior, we develop a novel methodology to understand and classify all of the memory access patterns of applications. Our methodology can be used to evaluate and optimize existing prefetchers, and, crucially, develop new prefetcher techniques. We offer insights and answers to key questions such as the following:

- What percentage of application cache misses can be handled by a particular prefetcher?
- What are the upper bounds for application miss coverage and performance improvement that a given prefetcher can provide?
- What type of prefetcher capabilities are required to prefetch a certain memory access pattern, or a given percentage of all misses in an application?
- How much opportunity is there for a prefetcher to run ahead and emit timely prefetches for a given cache miss?

We base our methodology on the observation that memory access patterns are concretely encoded in the application binaries themselves. By extracting these patterns directly, we can avoid the inaccuracy of guessing the patterns a priori, or the overhead of relearning the patterns indirectly. With this dataflow-based approach, we can classify every important miss in an application and reason not only about what type of computation is required to compute the next address, but also how much time there is to prefetch each next miss. Then, taking the application as a whole, we can reason about what type of prefetching techniques would improve performance, and whether they could be implemented in hardware or as custom injected software prefetches.

In order to focus only on the dataflow paths that are relevant to the application, we leverage both static binary analysis and dynamic profiling information. In particular, we perform a data dependency analysis for all load instructions that cause a significant number of cache misses to determine the sequence of instructions required to compute the delinquent (missed) load address. These instruction sequences are compacted to form *prefetch kernels* which we can then classify. Such an automated technique for classification and analysis of applications' memory access behaviors provides significant value for entities that run diverse and complex warehouse-scale computer (WSC) applications: First, our technique is scalable and performed entirely offline which enables a comprehensive, automated prefetch analysis for a large number of applications. Second, this approach can accurately predict the utility of prefetching on a per-application basis and hence can filter out applications that are unlikely to benefit from prefetching. Third, our technique can be leveraged to tune exposed prefetcher knobs, such as aggressiveness, without performing a comprehensive search of the configuration space. Finally, dataflow-based memory analysis informs the capabilities that are required for new prefetcher designs to be effective and gives us bounds on what we can expect from them.

We applied our methodology to a suite of memory intensive SPEC 2006 [22], PARSEC [6] and WSC applications to show the benefits of dataflow-based classification. In contrast to prior work which has shown the benefits of different prefetching techniques, our approach enables automated *reasoning* about why a particular prefetcher works or doesn't work for an application. Section 5 shows that some applications cannot benefit, even from an oracle prefetcher, and quantifies the expected gains of a particular type of prefetcher. We also provide insights about the complexity of WSC applications and show why optimizing WSC applications is challenging. Finally, focusing on commercially-viable implementations, Section 6 introduces three software based prefetcher designs that leverage dataflow analysis. Our evaluation shows that there exists no simple "one size fits all" prefetcher design and that prefetcher programmability is required to achieve high performance. In particular, by leveraging application-specific prefetcher configurations to account for timeliness and memory access types, we show that dataflow-informed software-based prefetching can reclaim up to 100% of performance lost to stalls in small benchmarks, and up to 44% of lost performance in large WSC workloads.

Pattern	Recurrence relation $\mathscr{R}$	Example	Notes
Constant	$A_n = A_{n-1}$	*ptr	
Delta	$A_n = A_{n-1} + d$	streaming, array traversal	d = stride
Pointer Chase	$A_n = Ld(A_{n-1})$	next = current->next	Load address is derived from the value of previous load
Indirect Delta	$A_n = Ld(B_{n-1} + d)$	$*(M[i])$	
Indirect Index	$A_n = Ld(B_{n-1+c} + d)$	$M[N[i]]$	c = base address of M, d = stride
Constant plus offset reference	$A_n = B_n + c_1$, $B_n = Ld(B_{n-1} + c_2)$	Linked list traversal	c_1 = data offset c_2 = next pointer offset B_n = address of the n$^{\text{th}}$ struct

Table 1. Classification of Memory Access Patterns

2 Background

Prefetchers track information about past memory accesses in order to predict future accesses. The amount of state being tracked as well as the prefetcher design itself determines the memory access patterns that can be learned by the prefetcher. Efficient prefetchers preload cache lines from slow memory into processor caches before they are used, thus reducing the average memory access time (AMAT). The performance of a prefetcher is determined by two main metrics, *coverage* and *accuracy*. Coverage defines the percentage of future memory accesses (or cache misses) a prefetcher can determine, while accuracy is defined as the ratio of prefetched memory elements to those used by future demand loads. Low coverage generally only bounds the potential performance improvement of prefetching, while low accuracy can lead to a slowdown if useless data evicts useful data from a cache or increases memory latency. Other important metrics are prefetch timeliness and cache size. The cache size determines the time period a prefetched memory element remains accessible with low latency before it is evicted. Timeliness refers to the time between a prefetch and the demand load of a cache line. If data is prefetched too late, the memory access latency of a demand load cannot be hidden, while prefetching too early can lead to an eviction of an otherwise accurately-prefetched data element.

Prior work has shown that the performance improvement delivered by prefetching is highly workload-dependent. For instance, the `Canneal` application from the PARSEC [6] benchmark suite suffers from low instructions per cycle (IPC) due to a high AMAT. While `Canneal` seems to be a perfect candidate to benefit from prefetching, prior work [3] shows that even complex prefetching schemes hardly improve performance. This paper addresses the question of why specific prefetchers are unable to address a particular workload. Therefore, we first provide a taxonomy of memory access patterns that exist in applications. We then define the operations that are required to prefetch a particular pattern and then develop the methodology to automatically classify the misses of an application.

3 Memory Access Classification

Applications exhibit a wide variety of memory access patterns, ranging from simple reuse and strides to complex calculations with memory indirection. A prefetcher's performance is thus proportional to the number of cache-miss-causing access patterns it can predict. In order to understand the access patterns that impede performance the most, we focus on the load instructions that contribute to the most cache misses. We make the observation that there exists a recurrence relation that defines the difference between two subsequent load addresses of the same program counter (PC). We define a recurrence relation $\mathscr{R}$ as $A_n = f(A_{n-1})$, where A is a memory address, n represents the n$^{\text{th}}$ execution of a particular load instruction and $f(x)$ is an arbitrary function. The complexity of $f(x)$ determines the capabilities a prefetcher requires to predict and prefetch a certain future cache miss. Table 1 shows some of the common patterns we observed while analyzing a wide set of applications, ordered by increasing complexity.

Note that while Table 1 cannot capture all potential access patterns, most accesses we have observed fit into one of these categories or a composition or nesting of them. For instance, double-linked list or tree traversals can be classified as *constant plus offset reference*.

Classifying memory accesses according to Table 1 provides the following insights: Given an application and its memory access classification, we can determine the capabilities required by a prefetcher to address a certain percentage of misses. For instance, delta patterns do not contain a load (Ld) and can be predicted by performing an arithmetic operation on address A_n to get A_{n+1}. For architectures that support prefetching certain patterns, we can then quantify the efficacy of their prefetchers. In particular, we can determine the percentage of successful prefetches for each particular class. This classification also enables a better understanding of prefetcher timeliness, since, while achieving timelines for some prefetch patterns is simple, for others it can be complex to impossible. For instance, for the delta pattern we can prefetch $A_n = A_{n-k} + d * k$ where k is a multiplicative factor that determines prefetcher timeliness (The higher the value of k, the further we prefetch into the future). The same applies to the *Indirect Index* pattern where future i values are easily predictable and can be used to prefetch arbitrary addresses stored in N. However, for *pointer chase*, running ahead is difficult as a load needs to be resolved first to predict the next address. If the memory latency given by the load chain is higher than the independent work executed between subsequent delinquent loads, hiding the memory latency is impossible, even with infinite run-ahead. Our approach enables these types of analyses and gives insight about the potential and the actually-achieved performance of a prefetcher technique.

The classification scheme of Table 1 maps memory accesses to design patterns and data structures and is useful for the human observer. However, these patterns are challenging to leverage for automated processing. Complex applications will often use compositions or variations of these patterns, for instance, a variation of *constant plus offset reference* is a tree traversal where both children are visited. To address this issue, we formalize our approach to express prefetch patterns as follows: For a given load instruction and its function to compute the next address, $f(x)$ can be expressed as a *prefetch kernel* which consists of a number of *data sources* and *operations* on those sources which generate the delinquent load address. There are three types of data sources: Constants, registers and memory locations. An operation might be any instruction (or micro-op) specified by the ISA of the architecture being analyzed. We list the most-frequently-used operations that our approach leverages for classification in Table 2. By binning multiple patterns into the same class (e.g. multiple loads are classified as *load*) this approach becomes general for handling arbitrarily-complex patterns while providing a rich set of analytical capabilities. For instance, it allows us to determine whether a pattern is indirect (uses the value returned by a load) or whether the data sources that feed into loads are constant or depend on a prior execution of the kernel. Furthermore, by analyzing prefetch kernels we can determine the instructions (capabilities) that a prefetcher needs to support to achieve a certain miss coverage of the application. We can also obtain insights about timeliness by analyzing the kernel's complexity and depth of the load chain. In the next section, we will explain how to obtain prefetch kernels from applications in an automated way.

4 Prefetch Kernel Extraction

Our proposed methodology performs offline analysis of application binaries in combination with traces that contain their instruction sequences and memory load and store addresses. Utilizing dynamic information from traces allows us to focus only on execution paths and misses that matter, and is critical for reducing kernels to their most basic behaviors.

Machine Classification	Corresponding Pattern
Constant	Constant
Add	Delta
Add, Multiply	Complex
Load, Add	Linked List, Indirect Index, ...
Load, Add, Multiply	Complex

Table 2. Machine Operations and Patterns

4.1 Dataflow Extraction Overview

Each miss in the application is caused by an address that was calculated by one or more instructions in the program binary (or libraries). The goal of dataflow analysis is to extract and then classify these address-generating instructions for each miss. As a side effect, the kernels also fully describe how to compute the next memory address accessed by the instruction. Because there is typically an intractable number of dataflow paths in most programs, we leverage dynamic profile information to prune our work to the paths that matter most for prefetching. We start by collecting application traces with Memtrace [7]. Each trace contains a sequence of instruction program counters (PCs) as well as the memory addresses of all load and store instructions. We also collect a cache miss profile comprised of a ranked list of PCs that cause cache misses. Such a profile can be generated by feeding the traces through a cache simulator, or via performance counters such as Intel's Precise Event-Based Sampling (PEBS) [16]. It is important to note that a miss profile is specific not only to individual workloads, but also to the machines on which they are run: If we were to classify and then run an application on two machines with different memory hierarchies, the classification might focus on unimportant dataflow graphs. As such, we assume that all dataflow analyses (and any resulting optimizations) are made on a per-application, per-architecture basis.

The combination of the program and library binaries, a program trace, and a miss profile allows us to rebuild the dataflow graphs for each miss-causing instruction in the application. These graphs can be used for classification as described in Section 5. Furthermore, our kernel-based prefetcher, described in Section 6, is based on these kernels.

A data dependency graph, or prefetch kernel, fully describes the data (e.g., constants) and computations that are required to form a miss address. The vertices of the graph represent operands, which can be constants, registers, or memory locations. The edges of the graph encode the operations (add, load, assign, etc.) between vertices which form the data dependencies. The graph is directed where the root (sink) vertex is the miss-causing instruction address, and there are one or more source vertices. The depth of the graph defines the critical path length of the kernel.

To perform kernel extraction, we search through the instruction trace in execution order for occurrences of miss-causing PCs. Each time an important miss-causing PC is found, we form a new dataflow graph that begins at that PC and searches backward in time until the last occurrence of the same PC. This window of execution history can be as small as a few instructions (e.g., a miss PC within a tight loop), or as large as millions of instructions in scale-out WSC workloads like web search. The root node of the graph is the memory address of the miss PC, which may be as simple as a fixed constant, or, in the case of x86, a combination of base, index, and segment registers as well as a scale and displacement. Whatever the case, each component of the address is added to the graph and then itself searched for its own data dependencies by looking even further back in the execution sequence. If a vertex is part of a load instruction, we check if there is a prior store to the same address. If so, we can connect the data dependence from the load to the store since, within a single thread, this is guaranteed to be the source. Load-to-store dependencies are very common, especially in x86 because of frequent register spilling. Our ability to follow dependencies through memory distinguishes our approach from prior work on prefetch-slicing and static-only analysis. For instance, the pointer analysis performed by static approaches to determine data dependencies can be challenging if not impossible whereas our approach can determine dependencies through memory natively.

We continue building the kernel recursively until all of the source vertices in the graph are *terminal*. A vertex is terminal if it is a constant (e.g., an immediate, displacement, or scale value), a register that cannot be traced back further (e.g., from a random number source), or any operand that has reached the edge of the instruction sequence (i.e., it has reached the prior occurrence of the miss PC in question). Thus each terminal vertex is a dataflow source, and can be a constant, a register, or a location in memory. Finally, for dataflow sources that change after use (at any point up until the graph root), we add the paths that describe these changes (See Section 4.3).

We note that our technique determines data dependencies within a sequence, ignoring control flow dependencies. We leverage the fact that the captured dynamic traces already resolve all branches for us and hence we do not need to follow different paths as would be the case for static analysis techniques. Instead, our tool analyses all instruction sequences in between pairs of the same miss PC, and as a result, the tool might discover different data dependency graphs for the same load PC. We track the frequency of occurrence of these different graphs and only utilize those for further processing that are executed frequently. As we will show in the next section, omitting divergent control flow paths enables us to compact prefetch kernels significantly.

4.2 Compaction

At this point, the extracted instruction graph represents the subset of the original application's execution history that is required to compute the load address of a delinquent load. A raw dataflow kernel typically includes a lot of extra and unnecessary operations that make classification difficult. For example, a kernel for a stride access pattern should be as simple as adding a constant to the prior address. However, it may actually contain memory loads and stores caused by register spilling and function calls (e.g., if the base address is passed as a stack parameter). Proper classification of memory access patterns relies on our ability to reduce these graphs to their minimal form, otherwise we would not be able to recognize even simple patterns like strides. To enable compaction, we developed the following techniques for removing dataflow artifacts introduced by the ISA and compiler.

Store-Load Bypassing In x86, register spilling leads to frequent data movement between registers and the stack in memory, consisting of matching push/pop, or more generally, store/load instruction pairs. While static analysis tools can match push/pop pairs (because the relative memory addresses of stack operations are well-defined), they cannot match general store/load pairs as the memory addresses are unknown at compile time. Our technique leverages dynamic memory traces, matching all store/load pairs and bypassing memory operations by directly connecting store source registers to load destination registers.

Arithmetic Distribution and Compaction Many dataflow graphs contain operations that ultimately cancel out and should be pruned for proper classification. For example, an immediate value added to a known-zero register (e.g., $0 in MIPS or a register XOR'ed with itself) does not actually require addition as it represents a constant value. Much more complex scenarios arise when, for instance, a fixed base address for an array is passed over the stack. In this case the stack pointer would be identified as a dataflow source, and we have to show that the base address is fixed even though the stack pointer may change frequently in the graph. We can do this by following the pushes and pops of the stack through the dataflow graph showing that they are balanced between subsequent executions of the graph kernel. If they are balanced, the stack pointer is fixed between executions of the kernel and we can then conclude that the base address is also fixed.

In order to facilitate arithmetic compaction and reduction, we flatten the dataflow graphs by distributing and reducing nested arithmetic operations. For example, if a graph represented the formula $(3 + 4) - (6 + 1)$, it would be flattened to $3+4-6-1$ and then simplified to zero and pruned. Arithmetic distribution and compaction is expensive computationally, but we've observed that it can reduce graphs by a factor of $1,000\times$, especially for kernels that span wide call stacks.

Assignment and Zero Pruning Assignments between registers (e.g., `mov %rcx, %rdx` in x86) occur frequently but are not relevant for classification. All trivial assignments, as well as those revealed by other reductions such as store-load bypassing or distribution, are optimized out of the dataflow graphs. Additionally, many instances of zero values occur either as constants or as computational patterns (such as `xor %rax, %rax` to clear registers in x86). Removing these vertices often causes large subpaths of the graphs to be pruned.

4.3 Kernel Extraction Example

This section provides an example execution of our offline trace analysis tool to generate a prefetch kernel. Figure 2 shows a code snippet of a linked list traversal that calls a trivial work function at each node. Since each node is allocated on the heap, its memory address is not required to have any relationship to other nodes in the list. In other words, there is no inherent spatial locality between dynamically-allocated heap objects. However, linked list nodes are related to each other through their next pointers, a case of *reference locality*. Thus a dataflow graph should relate the access of one node to the next with indirection via a load instruction.

In this program, profiling shows that a cache miss occurs in `do_work` when the 5th field of the node is accessed. We can make an important observation here: Cache misses are triggered by the first access to a cache line, the `do_work` function in this case, and not by the primary pointer chasing instruction `node=node→next` in `simple_chase`. As a result, our tool needs to be able to observe data flow dependencies through function calls and memory. Figure 3 shows the execution trace of one iteration of the `simple_chase` loop, where the last instruction (`0x4013fe` at the bottom) is the cache miss. The assembly code is in the AT&T syntax of `instruction source_operand, destination_operand`.

```
void simple_chase(Node *node, int count) {
  for (int i = 0; i < count; i++) {
    do_work(node);
    node = node->next;
  }
}
void do_work(Node *node) {
  node->field[6] = node->field[5] + node->field[4];
}
```

Figure 2. Linked List Pseudocode

To start the analysis, consider line 35 of the dynamic trace shown in Figure 3 which shows the miss-causing load instruction `0x4013fe`. The tool starts with this instruction and then traverses the trace in backwards direction until either all of the dataflow sources are terminal or it finds the same PC again. The resulting graph is the raw, non-compacted prefetch kernel. We can see that `0x4013fe` utilizes the `rax` register as a source operand to obtain the target memory address. We then search for the next line in the upward direction utilizing `rax` as a destination register which is the previous line 34. In line 34, we can observe that `rax` is read from main memory, potentially because of register spilling. Tracing the data flow through memory is challenging as we need to know the effective address used to read from memory. Fortunately, our dynamic traces not only contain instructions but also the effective memory address of each load and store (`mov` with indirect addressing). Using that additional effective address information (not shown in Figure 3) we can also follow dependency chains through memory. Note that register spills will be removed from the prefetch-kernel within the optimization pass, however, for now we need to observe dependencies through memory to continue the traversal. As shown by the arrows in Figure 3, the tool traverses the trace in a backwards direction, tracking all data flow dependencies until it reaches line 12, `pop %rbp`. Although the tool continues traversal until it hits line 1 (the boundary of the prior loop iteration), it cannot find an earlier instruction that stores the data for `rbp`, and thus `rbp` is a dataflow source (as well as all of the constants that were found along the path).

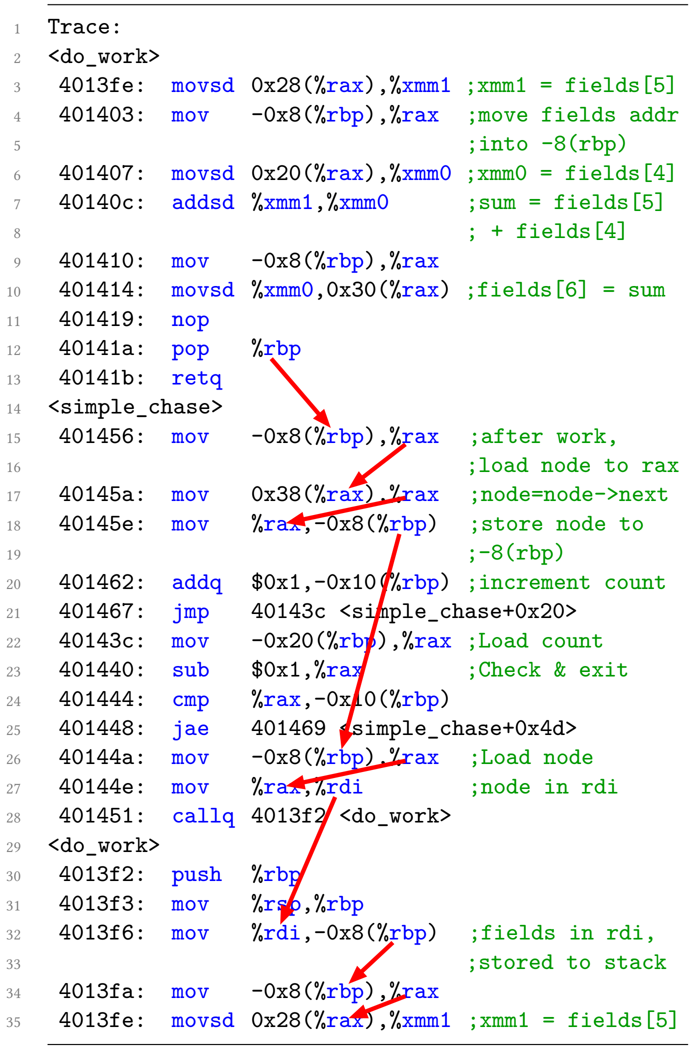
Even after simplifying the graph with assignment and store/load bypassing, the resulting dataflow kernel graph is still overly complex for a simple linked list access pattern:

$$A_n = 0x28 + \text{load}(0x38 + \text{load}(-0x8 + \text{load}(rsp))) \qquad (1)$$

This is because simple backwards traversal is insufficient for graph reduction: We also need to analyze if and how source operands are changed by instructions that are not contained in the backwards pass in order to classify the graph. In particular, we now need to perform a forward pass for each discovered source operand. Taking the source instruction `pop %rbp` from line 12, we determine that the load address (the stack pointer) is constant since the number of `push`/`call` instructions is equal to the number of `pop`/`ret` instructions and thus the source data register `rbp` will always be loaded from the same memory location. Next, we determine that the value of `rbp` does not change from the time it is loaded (line 12) until it is stored (line 30). Then, observing that the source data in `rbp` is constant, we can eliminate the two inner loads of Equation 1 and arrive at

$$A_n = 0x28 + \text{load}(0x38 + rax) \qquad (2)$$

where `rax` is the value of `rax` at line 16. Next, we observe that `rax` is loaded in line 15, modified in line 17, and stored in line 18, creating the recurrence relation $\text{rax}_n = \text{load}(\text{rax}_{n-1} + 0x38)$. Substituting this for the second term of Equation 2 yields the recurrence relation for a linked list described earlier in Table 1, and the dataflow graph kernel can be labeled as a linked list traversal.



```
Trace:
<do_work>
 4013fe:  movsd 0x28(%rax),%xmm1 ;xmm1 = fields[5]
 401403:  mov   -0x8(%rbp),%rax  ;move fields addr
                                 ;into -8(rbp)
 401407:  movsd 0x20(%rax),%xmm0 ;xmm0 = fields[4]
 40140c:  addsd %xmm1,%xmm0      ;sum = fields[5]
                                 ; + fields[4]
 401410:  mov   -0x8(%rbp),%rax
 401414:  movsd %xmm0,0x30(%rax) ;fields[6] = sum
 401419:  nop
 40141a:  pop   %rbp
 40141b:  retq
<simple_chase>
 401456:  mov   -0x8(%rbp),%rax   ;after work,
                                  ;load node to rax
 40145a:  mov   0x38(%rax),%rax   ;node=node->next
 40145e:  mov   %rax,-0x8(%rbp)   ;store node to
                                  ;-8(rbp)
 401462:  addq  $0x1,-0x10(%rbp) ;increment count
 401467:  jmp   40143c <simple_chase+0x20>
 40143c:  mov   -0x20(%rbp),%rax ;Load count
 401440:  sub   $0x1,%rax        ;Check & exit
 401444:  cmp   %rax,-0x10(%rbp)
 401448:  jae   401469 <simple_chase+0x4d>
 40144a:  mov   -0x8(%rbp),%rax   ;Load node
 40144e:  mov   %rax,%rdi         ;node in rdi
 401451:  callq 4013f2 <do_work>
<do_work>
 4013f2:  push  %rbp
 4013f3:  mov   %rsp,%rbp
 4013f6:  mov   %rdi,-0x8(%rbp)   ;fields in rdi,
                                  ;stored to stack
 4013fa:  mov   -0x8(%rbp),%rax
 4013fe:  movsd 0x28(%rax),%xmm1 ;xmm1 = fields[5]
```

Figure 3. Linked List Dynamic Trace

In summary, the dataflow process extracts and reduces dataflow graphs, analyzes source operands for zero pruning, and finally classifies each graph.

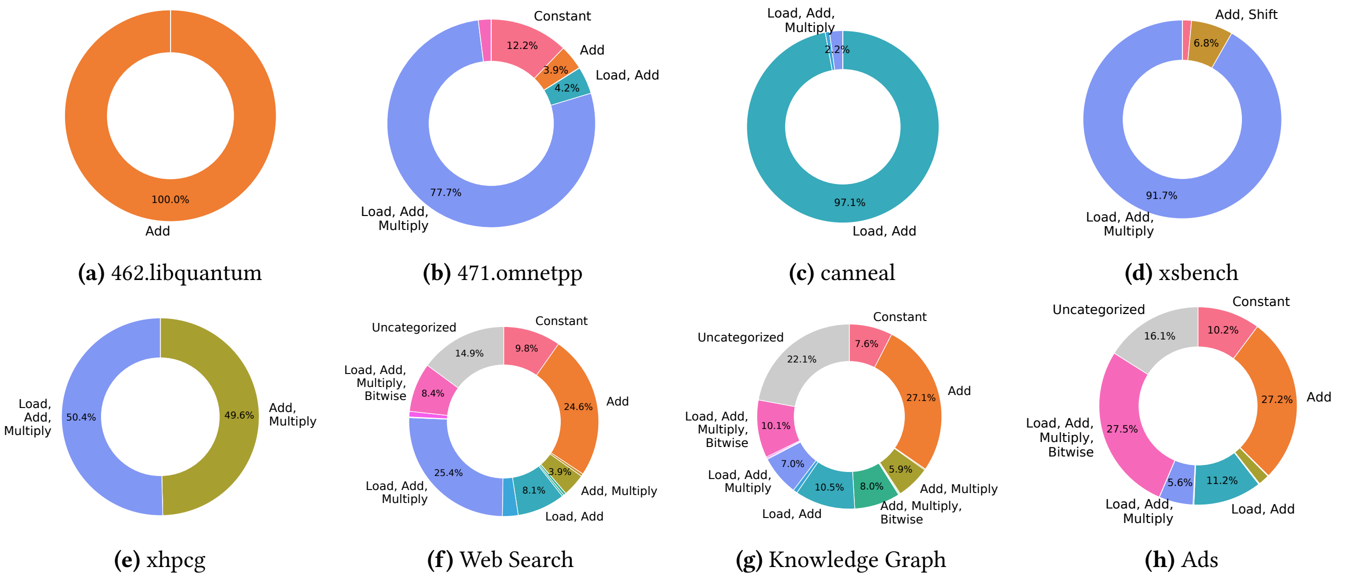
4.4 Implementation

We implemented our dataflow methodology in C++ in approximately 15,000 source lines of code. It is a stand-alone program that accepts an application binary (and libraries), execution trace, and miss profile as inputs, and outputs data classification information as well as software prefetch injection information which we use in Section 6. The graph analysis engine is ISA-agnostic, as all instructions are broken down into very basic micro-ops (so, for example, an x86 push instruction would be decomposed into a register decrement and memory store). However, the tool requires a front-end to convert application binary instructions into the micro-op-like graph nodes. Despite its enormous complexity, we target the x86 instruction set due to its prevalent use in large WSC environments. However, additional ISAs including ARM, POWER, or RISC-V could be supported with considerably less effort. We utilize Intel's XED [24] library to parse x86 op code classes and source and destination operands. Our tool supports the intricate details of x86 including different register sizes (a 32 bit register can be consumed by a 64 bit instruction in x86), REP-prefixed (repetitive) instructions and conditional instructions, as well as over 100 instruction classes.

For non-trivial dataflow applications (such as large WSC workloads like web search), we need to be able to process data dependency graphs of millions of vertices and edges and thousands of prefetch kernel instances. Computational efficiency, therefore, is of significant importance in both time and space. We employed a number of algorithmic improvements to reduce the runtime complexity of our tool, including multithreading (using a large shared instruction window pool), decoded instruction caching, hybrid data structures to support efficient allocation layout and insertion/deletion, and other cache and data structure optimizations to reduce memory fragmentation in the heap. With these and other optimizations in place, we can currently process 10 million trace instructions per second (MIPS) for simple applications and 0.1 - 5 MIPS for complex workloads, depending on the graph sizes and depths of the applications. Our initial naïve implementation was over hundred times slower and quickly ran out of memory on commodity server machines.

5 Analysis

In this section we utilize our tool to analyze eight memory-bound applications. We chose 471.omnetpp and 462.libquantum from SPEC CPU2006 [22], Canneal from PARSEC [6], the monte carlo neutron transport algorithm XSBench [47], and the high-performance conjugate gradient benchmark XHPCG [14]. Furthermore, we evaluate three WSC applications [5] from the Google fleet; A web search leaf node, knowledge graph backend, and an ads matching service. For each of the WSC applications, we collect traces with a representative single-machine loadtest. As in a profile-guided compilation approach, we execute the applications once, instrumented with DynamoRIO's Memtrace tool to generate application traces. The traces are then post-processed with our dataflow tool described in Section 4 to generate the following prefetch kernel analysis. For small applications, we analyze enough miss instructions to provide at least 95% miss coverage. For complex WSC applications with long miss tails, we currently limit our analysis to the top 200 miss PCs, which provides between 64%-86% miss coverage.



(a) 462.libquantum **(b)** 471.omnetpp **(c)** canneal **(d)** xsbench

(e) xhpcg **(f)** Web Search **(g)** Knowledge Graph **(h)** Ads

Figure 4. Memory Access Pattern Classification

5.1 Prefetch Kernel Classification

The access behavior of each miss-causing instruction is characterized by the complexity and type of calculations generating its addresses. Figure 4 shows the dataflow kernel classification for each of our applications. Each kernel is binned into classes similar to those described in Table 2, and the size of each classification bucket corresponds to the number of kernels that matched it. Because of control flow deviation, multiple kernels may exist for a single delinquent load PC, and each is classified separately.

We can draw several insights from Figure 4. For instance, for `462.libquantum` we see that virtually all cache misses can be computed with addition (in this case, a fixed delta), and, therefore, can be prefetched by a simple stride prefetcher (or with software prefetching). The opposite holds for `canneal`, `xsbench`, and `xhpcg` where fewer than 20% of misses correspond to the *add* or $add, shift$ categories prefetchable by stride prefetchers. For these, we can now bound the expected gains of a stride prefetcher in terms of miss coverage.

We also see that for most applications, computing the next address involves loading data from memory. This is not surprising, since many data structures rely on indirection, but it also reveals quantitatively why traditional hardware prefetchers are ill-equipped for these applications: Indirection typically breaks simple spatial correlations between addresses and these kernel addresses appear random in the access stream. As such, as few as 10% of misses are likely to be covered in apps like `xsbench` by stride-based prefetching.

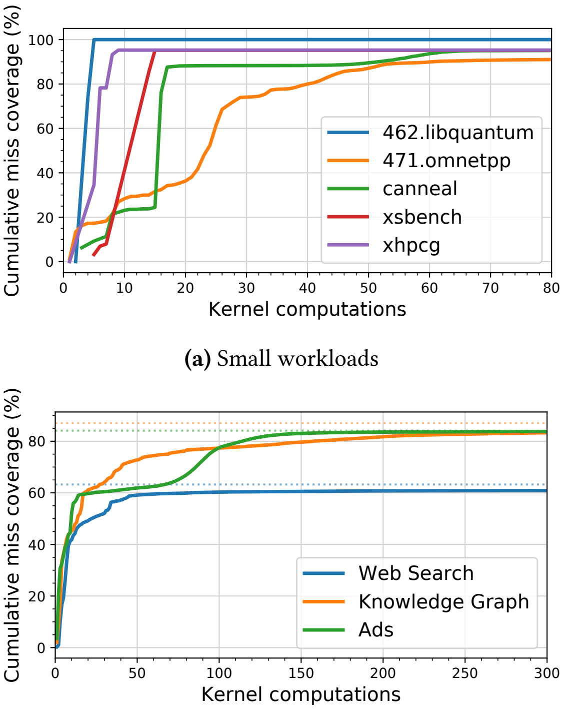
Interestingly, in our large WSC applications, a significant portion of misses (>30%) can be computed without indirection. However, many of these misses have large reuse distances which cause their data to be evicted and thus they aren't amenable to high-degree stride prefetching. Also significant, many of these "simple" kernels still require complex calculations involving many operations as we'll see next. Lastly, the large number of different patterns in WSC applications exceeds typical prefetcher resources such as the number of streams a stride prefetcher can observe.

In the WSC workloads, some kernels are shown as uncategorized. This is not a limitation of dataflow-based analysis, but rather a consequence of not implementing the long tail of hundreds of x86 instruction encodings and variations needed to fully analyze every kernel. A more complete implementation (or simpler ISA) would properly categorize these slices.

5.2 Prefetch Kernel Complexity

Another dimension of access pattern characterization is the number of calculations required to form an address. Figure 5 shows a cumulative distribution function (CDF) of the number of calculations (*add*, $shift$, *multiply*, *load*, etc.) a prefetcher must support (per kernel) to achieve a certain percentage of miss coverage. For simple applications like `462.libquantum`, five calculations per kernel covers all misses in the program. On the other hand, nearly 40% of misses in `Ads` have kernels of more than 10 operations, with some more than 100.

It is valuable to understand the computational complexity of an application's access patterns, since it informs us about the difficulty of learning patterns indirectly as well as about the storage and computational resources of a hardware prefetcher. Our large WSC application analysis suggests that prefetchers with simple pattern detection will be incapable of covering a majority of cache misses.



(a) Small workloads

(b) Large workloads

Figure 5. Kernel ops vs. coverage CDF

5.3 Prefetch Kernel Timeliness

Dataflow analysis gives us accurate address computations, but even accurate prefetches are useless if they can't be completed faster than the program needs them. We can leverage our dynamic profiling information with our compacted prefetch kernels to reason about how much time slack is available for prefetching each miss. Figure 6 shows a prefetch kernel scatter plot, presenting the prefetch kernel latency on the x-axis and the program latency (time between subsequent loads of the miss PC) on the y-axis. Each dot corresponds to one unique prefetch kernel, and the color of the kernel corresponds to its execution frequency. Dots near the blue line (with slope 1) represent prefetch kernels with approximately the same latency as the program. These kernels are challenging to prefetch, sometimes because they are executed in a tight loop (with no unrelated program calculations), because the kernel computations could not be simplified, or because memory loads dominate the latency (e.g., a pure pointer chase). If there is any run-ahead slack at all, then a high *prefetch degree* can help, where we define prefetch degree d as the number of recursive executions of a prefetch kernel. Applying a prefetch kernel multiple times enables further runahead by prefetching the next d misses of a delinquent load. Otherwise (and especially if these kernels contain chained loads as well), it becomes impossible to run ahead of the main program rendering prefetching useless.

On the other hand, kernels with larger distances from the blue line are capable of running ahead of the program and are likely to be prefetched in time. In this case, low prefetch degrees are sufficient and, to avoid cache pollution, it can be beneficial to delay injection of the prefetch. By leveraging this data, the prefetch degree, timeliness, and injection sites can be optimized for every prefetched load individually. We show the performance benefit achieved by adjusting per-load parameters in Section 7.

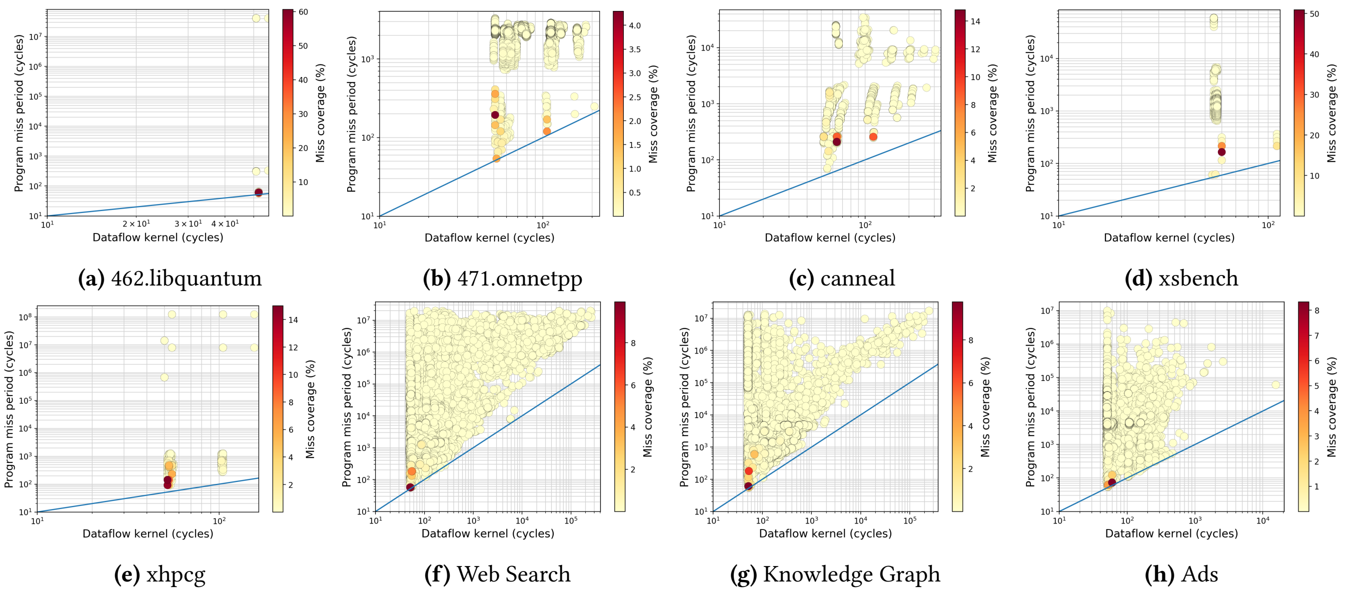
6 Prefetcher Design

We propose a new software prefetcher design based on dataflow analysis. Our approach executes the target binary once to obtain execution traces and miss profiles, performs dataflow analysis, and finally injects prefetches into a new binary, as in automatic feedback-driven optimization (FDO) [9]. In other words, prefetches are automatically injected by the compiler at the injection sites we specify, and the compiler re-links a new optimized binary. Inserting useful prefetches requires that we address the following questions:

- What address should be prefetched?
- Where should prefetch instructions be inserted?
- How aggressively should we prefetch?

Prior works on compiler assisted prefetching [1, 8, 10, 18, 33, 35, 46, 52] struggle to address these questions. In particular, they either require manual annotation or are significantly limited in the access patterns they can extract from source code (e.g., they can only prefetch simple loops with regular stride accesses). These prior approaches also have limited knowledge about the timeliness of a prefetch and hence have often been ignored in favor of hardware techniques.

Our dataflow methodology offers new opportunities for software-based prefetching. First, prefetch kernels are capable of expressing arbitrarily-complex formulas to compute the prefetch address, and can utilize multiple source registers and even memory as inputs. Second, by analyzing the distance in instructions between recurrent delinquent load PCs in the trace, we can determine optimal insertion sites for the prefetch instructions. In particular, by leveraging dynamic information such as IPC as well as microarchitectural knowledge of the cache and DRAM latencies, we can compute favorable insertion sites that are timely. We adopt the prefetch dynamic window injection technique [5] to minimize fan-in and fan-out of our insertion sites. Third, by leveraging additional profiling information contained in the trace, we can tune the prefetch degree (aggressiveness) for each individual delinquent load PC. In particular, for dense loops that contain few instructions unrelated to the delinquent load, high-degree prefetching is generally useful. However, if the loop count is small, then prefetching too far ahead will only pollute the cache. Our prefetcher utilizes Formula (3) below to compute the prefetch degree (D), where *LoopCnt* is the average number of iterations of a loop that resembles a



(a) 462.libquantum **(b)** 471.omnetpp **(c)** canneal **(d)** xsbench

(e) xhpcg **(f)** Web Search **(g)** Knowledge Graph **(h)** Ads

Figure 6. Prefetch Kernel Timeliness

recurrent access, *MissLat* represents the average DRAM to core latency and *KernelOps* is the number of instructions of a compacted prefetch kernel.

$$D = max(LoopCnt, (IPC * MissLat)/KernelOps) \quad (3)$$

For kernels without loads, the address for a higher-degree prefetch can be computed by applying the kernel repeatedly. However, for complex kernels, loads need to be resolved before reapplying the formula, thus limiting the effectiveness of high-degree prefetching.

While our technique can also be implemented in hardware, we opted against it because of its high complexity and cost and lower flexibility. Prior works, including run-ahead execution [21, 36, 37], precomputation threads [12, 54] or helper threads [32, 39], require complex hardware or utilize entire SMT cores for the purpose of prefetching, introducing unrealistic hardware overheads. Due to its complexity, we perform the dataflow analysis offline, once per application. Executing the prefetch kernels at runtime requires the capability to execute a wide range of instructions and hence at least a simple core would be required. As the main program never depends on prefetch kernel instructions, there is significant ILP among the two streams which can be leveraged by superscalar out-of-order processors for efficient execution of the kernels. By injecting kernels in software, we avoid large dedicated on-chip memory which would be required to hold the kernels in hardware. Furthermore, this enables greater flexibility such as configuring the trigger point for the prefetch or the aggressiveness in a kernel-specific way.

Our technique can be implemented with existing processors, however, there arises one issue. If the prefetch kernels contain load instructions, executing those speculatively can lead to illegal memory accesses causing a segmentation fault. Therefore, we propose one hardware change to existing processors which is the support of a speculative load instruction `specmov`. The `specmov` instruction tries to access the effective address in memory and is dropped instead of leading to an exception if the address is unmapped or the page access checks have failed. The dropped load also sets a flag in the condition code register which needs to be checked by a subsequent branch instruction to exit the prefetch kernel prematurely in the case of a misspeculation. As an alternative, prefetch kernels could be executed as part of a transaction such as Intel's TSX [19], however, serialization before and after the transaction would significantly reduce ILP between the main program and the independent prefetch kernel without applying additional hardware modifications.

Our methodology extracts prefetch kernels from execution traces in which all branches have been resolved and hence prefetch kernels do not contain control flow instructions. This enables efficient compaction of the kernels, however, it means that our tool may find multiple kernels for a given delinquent load. For instance, for every loop, the dataflow analysis will at least determine two kernels: One kernel reflects the regular iterations of the loop and another reflects the last iteration where the loop is exited. The dataflow analysis will find both kernels and count the number of executions of each kernel. We only utilize a prefetch kernel if it is executed significantly more frequently than the other kernels of the same PC. If two kernels have the same frequency, we prefetch both. If more than two frequent kernels exist, we ignore them entirely for prefetching. Our experiments have shown that prefetching more than two kernels often leads to cache pollution, offsetting any benefits of prefetching.

Parameter	Value
CPU	Intel Haswell E5
L1 Instruction cache	32 KiB, 8-way
L2 Unified cache	256 KiB, 8-way
L3 Unified cache	Shared 2.5 MiB/core 22-way
All-core turbo frequency	2.5 GHz
L3 cache latency	60 cycles (average)
Memory	DDR4 10GB/s/core 200 cycles

Table 3. System Configuration

7 Evaluation

We evaluate our proposed prefetcher technique via simulation and compare it against several baselines. The first baseline reflects a contemporary Intel processor with a simple stream prefetcher that can detect regular strides. Based on this architecture we evaluate compiler-assisted techniques such as prefetching constant prefetch kernels with a fixed degree as well as variable per load degree. Finally, we evaluate a prefetcher that leverages the `specmov` instruction to also prefetch complex memory access patterns.

7.1 Methodology

We implemented our kernel prefetcher as part of the zsim [43] simulator which we modified to include a trace-driven execution mode. We collect traces with DynamoRIO's [7] memtrace client, and limit traces to two billion instructions during steady-state execution. We inject the prefetch kernels at the insertion sites determined by dataflow analysis. If the insertion site is part of an inlined function, we insert the kernel at each location. To perform high-degree prefetching, we execute the kernel in a loop to compute the next *degree* prefetch addresses. In case there exist two frequent kernels for a delinquent load, we insert both kernels as described in Section 6. All instructions executed as part of prefetch kernels are modeled as overhead and not included in IPC improvements.

The baseline stride prefetcher observes L3 misses and prefetches into the L1 cache. Our software prefetches also target the L1 cache. As zsim implements an inclusive memory hierarchy, this guarantees that lines are also prefetched into the L3 cache. Prefetching is memory-bandwidth-limited and both hardware and software prefetches are dropped whenever the processor consumes more than 90% of the peak memory bandwidth. For the processor we utilize an Intel Haswell-like configuration with the properties shown in Table 3. We limit DRAM bandwidth to 10GB/s per core. We use the same set of applications as used in the study performed in Section 5: `462.libquantum` and `471.omnetpp` from SPEC 2006 [22], `Canneal` from PARSEC [6], `XSBench` [47], `XHPCG` [14] and the WSC applications `Web Search`, `Knowledge Graph`, and `Ads` from Google. To show the upper-bound of performance gains that can be obtained with prefetching, we compare each approach against a perfect memory hierarchy in which memory accesses are always served from the L1 cache, including cold misses.

7.2 Results

Figure 7 shows the performance improvement that we measured with our prefetching methodology. We compare a stride prefetcher (red, left) with dataflow-informed prefetchers of simple, load-less kernels and fixed prefetch degrees (df_1, df_2, df_4, df_8) as well as a variable-degree (df_var) dataflow prefetcher that can utilize prefetch kernels with memory loads (df_var_load). Furthermore, we show the theoretical upper bound performance (perfect) where every access hits the L1 cache (gray, right).

As we showed in Figure 4, `462.libquantum` performs predictable accesses with a constant stride. This causes frequent L3 cache misses that are easy to predict by all prefetchers, however, a high prefetching degree is required to achieve optimal performance. This is challenging for both software and hardware approaches that generally need to be configured to prefetch a fixed degree, but is addressed by our software prefetch techniques that leverage a variable per-load degree. `471.omnetpp`, `canneal` and `xsbench` exhibit a large fraction of complex kernels that include chained loads. As a result, the stride prefetcher and the software techniques that are limited to prefetching non-load kernels are unable to improve performance significantly, while the prefetching technique leveraging `specmov` delivers a speedup of 1.38×, 1.14× and 1.9×, respectively.

As shown in Figure 6, `471.omnetpp`, `canneal` and `xsbench` utilize relatively large dataflow kernels, and hence, the gains achieved by prefetching load chains is limited by timeliness. According to Figure 4, 50% of `xhpcg`'s dataflow kernels are composed of add/multiply operations that do not require loads. The extracted address computation formulas only leverage simple arithmetic, but nevertheless, cannot be learned by the stride prefetcher which can only handle kernels limited to `add` operators with constant addends. For `xhpcg`, our tool fails to determine the optimal prefetch degree as a more aggressive degree of 8 delivers higher performance. Nevertheless, for the other applications, variable prefetch degree always performs better than using a fixed degree.

All three WSC applications showed a wide range of dataflow kernel classes. They also show an order of magnitude higher number of performance-relevant loads and hence are challenging to prefetch. Nevertheless, for `Web Search` our prefetcher shows 9% IPC gain and for `Knowledge Graph` an improvement of 6%. These applications have been performance optimized for years and hence limited gains are expected, but nevertheless a 9% IPC gain can save millions of dollars for large WSC providers.

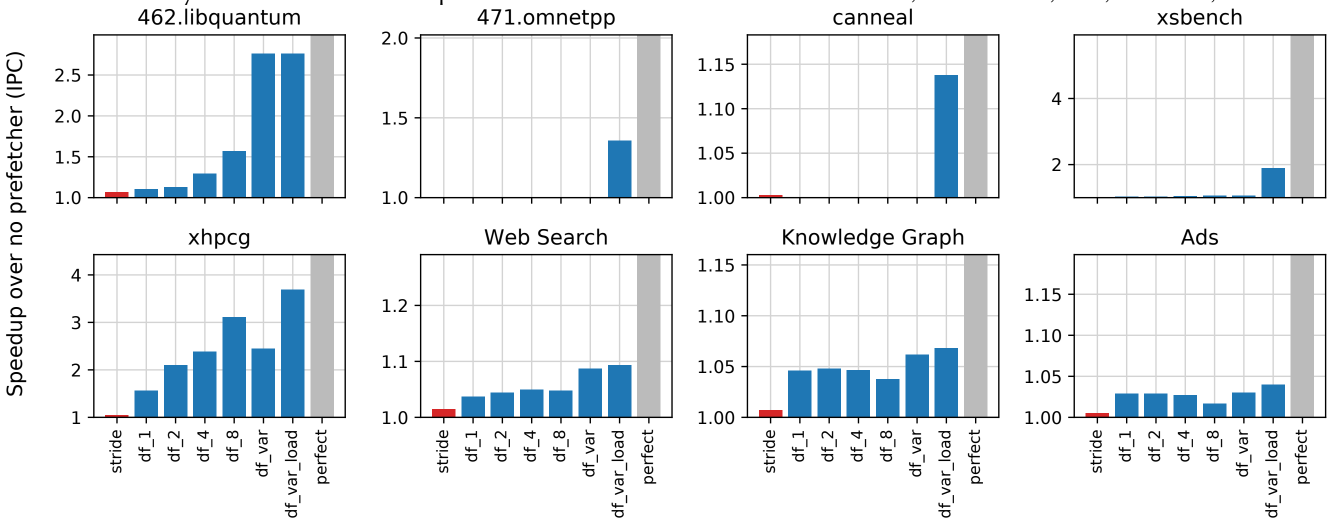


Figure 7. Software Prefetcher IPC Speedup

8 Related Work

Hardware Techniques: Several hardware mechanisms have been proposed [2, 12, 15, 21, 30, 32, 36, 37, 39, 54] to prefetch complex memory access patterns that depend on the data and control flow of the executed program. These approaches leverage additional helper threads, running on separate cores or custom hardware to cause cache misses in advance of the main executable, thus prefetching data into caches. These dynamic techniques generally suffer from two disadvantages: First, they introduce high hardware complexity and cost as they require powerful cores to process the helper threads. For timeliness, these helper threads need to be fast and hence they cannot rely on small wimpy cores. Second, dynamic techniques have limited ability to reduce the code footprint of the helper threads using compaction. Performing data dependency analysis in hardware requires substantial storage space and hence is limited to short instruction sequences. For example, continuous run-ahead [21] limits its dependency chains to 32 operations. Furthermore, while dependency analysis can filter out independent instructions, implementing the compaction techniques described in Section 4.2 in hardware is challenging. In contrast, our technique performs the dependency analysis offline, avoiding hardware overheads and enabling it to process and compact dependency chains of millions of instructions. Temporal prefetchers including GHB [38, 49], ISB [26] and MISB [51] can potentially prefetch arbitrary memory access patterns by storing long sequences of past accesses. However, to provide high performance they require megabytes of expensive on-chip memory. Additionally, they can only prefetch previously-seen patterns, whereas dataflow kernels compute the next address based on the current state of the system.

Software Techniques: Software prefetching [8] techniques have been well-studied in prior work. Compiler-based techniques [1, 10, 18, 46] perform static code analysis to generate prefetch targets. The performance benefits provided by static approaches are limited, as only simple structures such as Singly-Nested Loop Nests (SNLNs) [50] or regular strides [29, 35, 52] can be learned. In contrast, our approach can handle complex dataflows of generic software algorithms and data structures. There exist several works that analyze more complex recursive data structures such as linked-lists at compile time to insert jump pointers [11, 33, 40, 41] pointing to an element within the same data structure at some distance e.g. several elements ahead in a linked-list. These jump pointers are inserted into the original data structure (e.g. a linked list node) and hence require additional storage as well as source code modifications for initializing jump pointer references whenever an element is inserted into a data structure. Prior works [31, 34] leverage dynamic profiling for determining the most useful prefetch candidates, but do not leverage the capabilities of trace-based dataflow analysis to explore timeliness and load classification. Zhang [53] performs dynamic prefetch optimization based on profiling, however, this work requires an extra thread while running the main application.

9 Conclusion

In this paper we introduced a new methodology for analyzing memory access patterns of applications for prefetching. Our offline trace-based dataflow analysis provides detailed insights about the memory access types that applications exhibit and how they affect execution performance. Our techniques enable automated classification of memory patterns, and allow us to reason about the effectiveness of a prefetcher for a given application. We applied our approach to propose new software-based prefetcher designs that leverage per-load configuration knobs such as prefetcher aggressiveness to achieve performance benefits many times greater than a stride-based baseline with trivial implementation cost. Finally, we anticipate that our technique will be leveraged in future work to develop new software- and hardware-based prefetcher designs. We expect that these new designs will have to be highly configurable, preferably by software, to cover the wide range of access patterns that we quantify in this work.

Acknowledgements

We would like to thank anonymous reviewers for their insightful comments. We would also like to thank our collaborators at Google, the Stanford Platform Lab, and the SRC Center for Research on Intelligent Storage and Processing-in-memory (CRISP) for their support. Heiner Litz's research was supported by the NSF grant CCF-1823559.

References

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

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

[3] Islam Atta, Xin Tong, Vijayalakshmi Srinivasan, Ioana Baldini, and Andreas Moshovos. Self-contained, accurate precomputation prefetching. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 153–165. ACM, 2015.

[4] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *High Performance Computer Architecture (HPCA)*, 2018.

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

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[8] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 40–52. ACM, 1991.

[9] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Code Generation and Optimization (CGO)*, 2016.

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

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

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

[13] Fredrik Dahlgren and Per Stenström. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *hpca*, pages 68–77, 1995.

[14] Jack Dongarra, Piotr Luszczek, and M Heroux. Hpcg technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

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

[16] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *Workshop on memory systems performance and correctness*, 2008.

[17] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 2011.

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

[19] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[20] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

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

[22] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

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

[24] Intel. Intel x86 encoder decoder (xed). https://github.com/intelxed/xed, 2019.

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

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

[27] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[28] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA)*, 2015.

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

[30] Jaejin Lee, Changhee Jung, Daeseob Lim, and Yan Solihin. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(9):1309–1324, 2008.

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

[32] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G Abraham. Dynamic helper threaded prefetching on the sun ultrasparc

cmp processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104. IEEE Computer Society, 2005.

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

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

[35] Todd C Mowry, Monica S Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *memory*, 100(110):120, 1992.

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

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

[38] Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*, pages 96–96. IEEE, 2004.

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

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

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

[42] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, 2003.

[43] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *Computer Architecture (ISCA)*, 2013.

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

[45] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 171–182. IEEE, 2002.

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

[47] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[48] Vish Viswanathan. Disclosure of hardware prefetcher control on some intel® processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors. Accessed: 2019-08-09.

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

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

[51] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient metadata management for irregular data prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 449–461. ACM, 2019.

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

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

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