2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Division of Labor: A More Effective Approach to Prefetching

Sushant Kondguli and Michael Huang

Dept. of Electrical and Computer Engineering
University of Rochester
Rochester, NY 14620, USA
{sushant.kondguli, michael.huang}@rochester.edu

Abstract

Prefetching is a central component in most microarchitectures. Many different algorithms have been proposed with varying degrees of complexity and effectiveness. There are inherent tradeoffs among various metrics especially when we try to exploit both simpler access patterns and more complex ones simultaneously. Hypothetically, therefore, it is better to have collaboration of sub-components each specialized in exploiting a different access pattern than to have a monolithic design trying to have a similar prefetching scope. In this paper, we present some empirical evidence. We use a few components dedicated for some simple patterns such as canonical strided accesses. We show that a composite prefetcher with these components can significantly out-perform state-of-the-art prefetchers. But more importantly, the composite prefetcher achieves better performance through a more limited prefetching scope while attaining a much higher accuracy. This suggests that the design can be more readily expanded with additional components targeting other patterns.

***Keywords*-Prefetching**

I. Introduction

Prefetching is a tried-and-true mechanism to hide long latencies and thus to reduce the chance of costly pipeline stalls. Over time, innumerable prefetching algorithms have been proposed. Fundamentally, they all exploit some predictability of the memory access patterns, but they differ quite a bit in almost every aspect: the specific patterns targeted, the amount of resources required, accuracy of prefetches, the percentage of misses eliminated etc. There are usually some inherent tradeoffs in the design. For instance, simpler patterns (such as constant-stride accesses) lead to good prefetch accuracies but may have limited scope[1]. A prefetcher is ultimately evaluated by its contribution to performance. Implicitly, this puts pressure on designers to broaden prefetch targets.

Prefetching scope can be broadened to include variations of the pattern targeted or more directly by combining different types of prefetchers. Whatever the approach, increasing scope often leads to reduced accuracies. Inaccurate prefetches not only slow down other prefetches and demand accesses but can also pollute the cache and can thus actively hurt performance. We need to carefully analyze the *marginal* effect of expanding scope. As we will show later – and to our surprise – many state-of-the-art prefetchers that cover variations of the strided access pattern have poor marginal benefits on these variations. In general, a monolithic prefetcher with a single target pattern will unlikely achieve broad scope while maintaining high accuracies. We believe that it is better in practice to design more than one component within a *composite* prefetcher, each specialized for its own focused prefetching targets. With this division of labor, scope and accuracy are decoupled. The former is achieved with better combinations, while the latter, improvement of components. We will present some analyses in this paper to support this view.

Designing composite prefetchers requires a different approach to the analysis of prefetcher performance. Ultimately, the figure of merit of a product prefetcher is some kind of cost-benefit ratio. The benefits include cycles saved and the concomitant energy savings. The costs involve static investments (logic and storage) and dynamic energy expenditures. Since the vast majority of prefetchers use relatively simple automata, the energy cost is almost always outweighed by the energy savings resulting from successful prefetches and thus commonly ignored. Consequently, prefetchers are mostly evaluated and compared by their performance benefits (with some attention to the storage cost induced if it is non-trivial). While this comparison can show how individual prefetchers fare relative to each other in their ultimate figure of merit, it is insufficient to evaluate the merit of the algorithm as a component or to guide the selection of components. For example, if a modification to a prefetcher sacrifices its scope but vastly improves the accuracy, the modification will fare poorly in the traditional merit analysis but the design can be an excellent component in a composite prefetcher. Indeed, in this paper, we make a case for designing such components with limited prefetching

This work is supported in part by NSF under grants 1514433 and 1533842.

[1]By scope we mean what is being targeted or attempted by the prefetcher. We will have a more precise definition in Sec. III

scope.

Rather than broadening the scope via more sophisticated designs to accommodate variations of a basic pattern, we opt for more focused components that each only handles a more limited case, but does so with a high accuracy and efficiency. When used together, these components complement each other and form a composite prefetcher whose scope is largely the sum of that of the components' and an accuracy that is not intrinsically affected by the expanding scope. With this division of labor approach, we can lower the barrier to innovation. Improving the accuracy of existing components or inventing additional components that expand scope will slowly but surely increase the performance of future prefetchers.

II. Related Work

Various prefetching approaches have been proposed over the years, targeting different types of memory access patterns. Characterizing patterns can be an imprecise endeavor. An observed pattern can be the result of a simpler underlying pattern obfuscated due to a number of factors such as out of order execution, non-linear transformation of the address and so on. A state machine designed to capture certain pattern may capture other patterns or false positive instances. Nevertheless, we can still divide these access patterns being targeted roughly into four different categories: regular strided patterns, pointer patterns, irregular patterns, and region patterns.

Prefetchers targeting regular strided patterns can be as simple as prefetching the next cache line following the access of one [15] or could involve identifying unique stride that separates addresses in a memory stream based on the PC of the instructions that access them [18] or based on global order [23]. Using a global history buffer (GHB) [21] helps in identifying multiple unique strides in an address stream. GHB, however, requires a lot of storage and variations with smaller storage costs have been studied [12], [34], [36].

Pointer chasing prefetchers try to predict future accesses by using hardware/software approaches to predict the address being pointed to by the pointers. This can be done by inserting prefetch instructions via compiler optimizations [25] or by correlating the data in the data cache with its address and predicting its likelihood of being a pointer [7]. It has been observed that pointer based prefetching has poor timeliness [7], [13], [26]. By chaining PC localized streams Diaz et al. [8] attempt at improving timeliness of such prefetches.

Irregular memory access pattern based prefetchers target harder to prefetch addresses by identifying certain key characteristics of the memory stream [4], [5], [24], [31]. Markov prefetchers [6], [14], [35] predict from an observed memory sequence, the sets of unique addresses that are likely to occur in the future. Markov prefetchers require a lot of storage space, ISB [13] uses translation buffers to implement Markov prefetchers in a reduced space. SMS [30] takes a different approach to identifying irregular streams by storing the accesses pattern of a page and speculating that the future reference to another page by the same instruction can result in a similar access pattern.

VLDP [29] and SPP [17] target both regular and irregular access patterns. Unlike a regular access pattern prefetcher, both of these prefetchers try to identify a pattern amongst the strides of memory accesses. KPC [16] extends the design of SPP to synergistically integrate cache replacement policies with prefetching. BOP [20] identifies a best possible offset that matches the strides of most of the memory accesses in a phase.

Region prefetchers [9], [33] try to predict a region of memory that is most likely to have a future access and prefetch the entire region. Such a prefetcher suffers from excessive amount of inaccurate prefetches. Controlling the aggressiveness using different heuristics has been shown to improve prefetch accuracy and performance [11], [32].

III. Building Composite Prefetchers

Programs have many different data structures and their memory access patterns change depending on the data structure and the code. They thus exhibit a variety of memory access patterns. Prefetchers detect some of these patterns and make predictions about future accesses. The design of a prefetcher involves tradeoffs. For instance, targeting more access patterns may lower the accuracy of the prefetches and creates more pollutions. To understand the effect of these tradeoffs in more concrete terms, we take some prefetchers and quantify two aspects of the design choices: how ambitious the goal is and how well it is achieved.

The first metric measures how much of the miss stream a prefetcher *attempts* to cover, or its "ambition". We purposely want to separate attempt from how well it is achieved. Thus we are only concerned with what addresses are being prefetched and completely ignore whether the prefetch helps or hurts. We call this metric prefetching scope (or scope for short), which measures the fraction of footprint at least attempted by a prefetcher at some point as follows.

Let $FP = \{A_i\}$ be the footprint (of a particular cache level, the same below), *i.e.*, the set of unique cache line addresses of misses[2] in the baseline system without a prefetcher. Since not every line is equally important, the weight factor W_i (for cache line A_i) is the number of misses of address A_i (over a particular window of observation). Let PFP be the prefetching footprint (addresses being prefetched) over the same observation window. The scope of a prefetcher P over that observation window is defined as the fraction of FP covered by PFP, weighted by the

[2]In all cases, we ignore secondary misses – those with a pending fetch to the same cache line.

weight factors as shown below.

$$S(P) \equiv \frac{\sum_{j|A_j \in FP \cap PFP} W_j}{\sum_{i|A_i \in FP} W_i}$$

Note here that a line is considered covered as long as the prefetcher has attempted to prefetch the line (within the observation window), without regard to the frequency or utility of the prefetch. In other words, $1 - S(P)$ shows how much of the target address space the prefetcher did not attempt to cover.

The second metric, effective accuracy, measures how useful each of the issued prefetch is. We define it as the number of misses that are avoided (as a result of engaging a prefetcher) divided by the number of prefetches issued.[3]

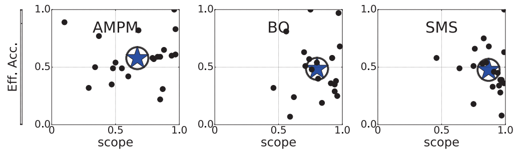


Figure 1. Accuracy vs scope for four different prefetchers. Each dot in the plots represent one application. Details of the observation windows are discussed in Sec. V-A. The per-prefetcher global average (stared circle) is calculated as the result of one large observation window strung together from those of the individual applications.

In Figure 1, we show the scope and (effective) accuracy of three prefetchers (AMPM [12], BOP [20] and SMS [31]) over the entire suite of SPEC 2006 applications.[4] For a broad-brushed characterization, we can look at the overall average. As we go from AMPM to BOP and then to SMS, the scope increases from 67% to 76% and then to 87%; but accuracy reduces from 58% to 49% and then to 48%. The inherent tradeoff between scope and accuracy is clearly reflected in these statistics. Unfortunately, improving scope at the cost of reducing accuracy may not yield much ultimate performance benefit – and it certainly increases the cost of prefetching.

Given these observations, our intuition is that prefetching may be better achieved through division of labor, where multiple components are used (Figure 2). In such a composite prefetcher, scope can be improved by finding *specialized* prefetcher components aiming at those prefetch targets not already covered by existing components. Under such a system, the figure of merit for a prefetcher *component* is no longer large scope and accuracy, but just high accuracy (with a meaningful scope). This is analogous to the human society: with division of labor, a successful member no longer needs to excel in a lot of areas but can be specialized in a narrower domain and still be valuable.

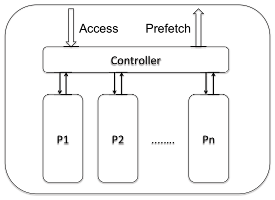


Figure 2. A logical overview of composite prefetchers.

A number of potential benefits may derive from a composite prefetcher compared to a monolithic design. Some benefits can already be seen from composite prefetchers using existing algorithms as components, which will be analyzed later. Other benefits are conjectural and may materialize over time as designers adopt the approach and come up with more component designs. We discuss some benefits below.

1) **Efficiency:** Most prefetchers need storage and energy to memorize and analyze addresses to recognized patterns. With specialized patterns, each component can maximize storage efficiency. For instance, some prefetchers memorize spatial patterns. Strided accesses would be an inefficient use of such storage. In a composite prefetcher, with appropriate coordination, each component can ignore addresses known to be associated with other components. This also minimizes the chance a prefetcher recognizing false positive patterns.
2) **Clarity:** Different access patterns have different probability of prediction success. Separately tracking each pattern allows better decision. For instance, given limited resource, the system may preferentially handle requests from some components over others. Given different success probability, it is also easier to decide the appropriate prefetch destination.
3) **Flexibility/configurability:** Different applications may have different likelihood of showing certain patterns. Based on feedback, programs may be able to adjust the parameters of different prefetch components, disable certain components, or even bring their own ad-hoc prefetch logic.

In addition to higher production efficiencies, division of labor in human society brought increasing specialization and deep expertise. We hope a similar benefit may follow over time where component design becomes more sophisticated, highly accurate, and comprehensive. We believe a future composite prefetcher will be significantly more capable than what we can present in this paper. Next, we will discuss a specific implementation which should be considered only as

[3]There is a commonly used metric of accuracy which measures the fraction of prefetches issued that are accurate – defined as accessed before being evicted from the cache. This definition is somewhat optimistic as it does not take into account misses induced by prefetching. The accuracy of a prefetcher can be no worse than 0, where the effective accuracy of a prefetcher can be negative. In our analysis, we found that effective accuracy is between 62% and 100% of accuracy.

[4]Note that neither scope nor effective accuracy is a figure of merit. They are intended to help understand the tradeoffs in prefetcher design in a more concrete manner.

a proof-of-concept prototype.

IV. Example Implementations

Our composite prefetcher consists of a few custom components. They target patterns such as strided streams and pointer traversal. They will be combined with a simple coordinator to control their runtime action. We will also experiment with existing (monolithic) prefetchers as components.

A. Targeting Strided Streams

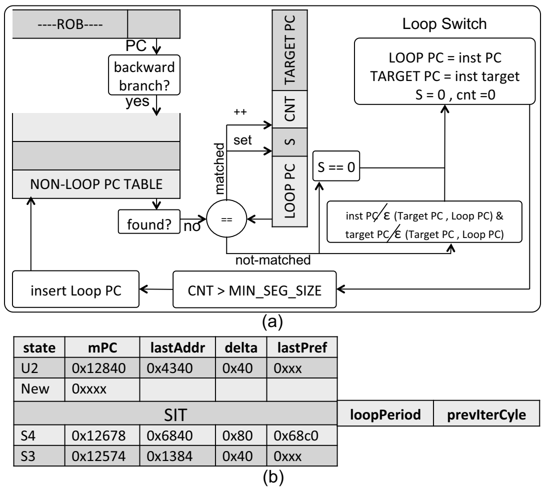
Among all the prefetching targets, the canonical strided streams are perhaps the easiest from the hardware standpoint. Many prefetchers target these streams either explicitly or implicitly. However, most of these prefetchers target variations of the basic pattern. Our component, which we call T2, only targets canonical streams, by which we mean those generated from the repetition of a single instruction within the inner loop. The implementation can be viewed as going through three high-level steps: ❶ identifying loops, ❷ detecting streams associated with the loop, and ❸ prefetching.

1) Identifying loops: When execution is broken down into iterations of loops, canonical strided streams are both relatively easy to detect and to prefetch. The loop hardware, therefore, tries to identify inner loops. The general idea is to capture the "loop branches" which manifest as back-to-back instances of the same backward branch, with no other backward branches occurring in between. These branches then serve to mark the boundaries between iterations of the same code.

Certain complications prevent these branches from occurring back-to-back. For example, a backward branch within the loop body can intersperse among instances of the loop branch. To filter these branches out, we use a loop-branch register to keep track of both the PC and the target of a backward branch (Figure 3-a). When a newly encountered backward branch matches that stored in the loop-branch register, the loop is identified. This simple heuristic does not cover all possibilities but works well in typical cases. Some backward branches are not loop branches. Once the hardware realizes that, it remembers them in a table to help reduce the time it takes to identify a stable loop. In our evaluation we use a Non-Loop PC Table (NLPCT) of 20 entries to store the PC of such branches. If a branch is found in this table, it is skipped by the loop marker.

2) Strided stream detection: T2 only targets the most common type of canonical streams: those generated from the same static instruction inside the innermost loop. This is mainly achieved using the stride identifier table (SIT). The basic idea is to keep track of every memory accessing instruction (using its PC), the address of its last execution instance (`LastAddr` in Figure 3-b), and the delta between the addresses in two consecutive accesses (`Delta`). When the detected delta is stable, we mark the instruction in the I-cache, so that the issue logic will notify T2 every time it issues the instruction and T2 can prefetch accordingly.

From this basic idea, there are two modifications. First, it is common for a number of different memory instructions to be accessing the same stream. This could be the result of loop unrolling for example. Instead of tracking all such instructions, we activate tracking only when an instruction triggers a primary cache miss (in L1). This is achieved by labeling a memory instruction in one of four states in the I-cache: When an I-cache line-fill occurs, all memory instructions start in state 0 (unknown). Until the instruction encounters a primary cache miss, the system ignores instructions in state 0. When it triggers a primary miss, it transitions into state 1 (observation). In this state, every instance of the instruction will cause an update in SIT. If we see the delta is stable, we move the instruction to state 2 (strided), otherwise, we label it state 3 (non-strided). We find that the system is not sensitive to the criterion of labeling an instruction strided. In this paper, when we see sixteen consecutive instances of the same delta, we label the instruction strided. Conversely, when we see four consecutive instances of changing delta we label the instruction non-strided. We begin issuing prefetches in state 1 if we have seen four consecutive instances of the same delta if T2 is not already issuing too many prefetches.



state	mPC	lastAddr	delta	lastPref		
U2	0x12840	0x4340	0x40	0xxx		
New	0xxxx					
SIT					loopPeriod	prevIterCyle
S4	0x12678	0x6840	0x80	0x68c0		
S3	0x12574	0x1384	0x40	0xxx		

Figure 3. (a)Loop hardware. (b)Stride identifier table.

The second modification is to disambiguate between different call sites. In some loops, two different strided streams are being accessed. However, both accesses may be done through function calls. This is especially common when the code is written in an object-oriented language. Using simple PC, we can not distinguish between multiple call sites and would fail to detect strides. A simple fix is to take the PC and xor it with the top entry of the return address stack (RAS). We use this modified PC (`mPC` in Figure 3-b) in the design.

3) Prefetching: Once a memory instruction is labeled as strided, the act of calculating future addresses is straightforward. Moreover, given the loop hardware, we have a good estimate on the execution time per iteration. If we track average memory access time, we can control prefetch distance. Specifically, the appropriate prefetch distance is $d = \frac{AMAT+m}{T_{iter}}$ where $AMAT$, m, and T_{iter} are average memory access time, margin constant, and average execution time per iteration, respectively.

B. Targeting Pointer Chains

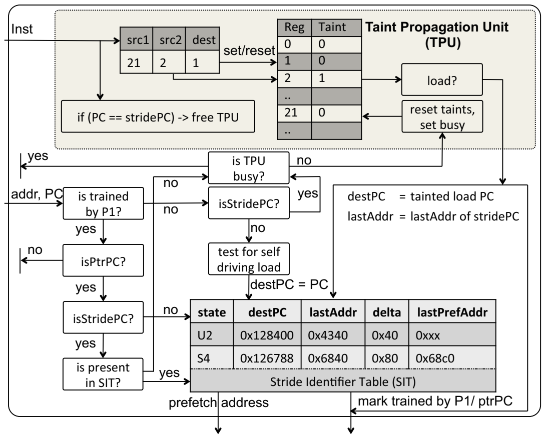
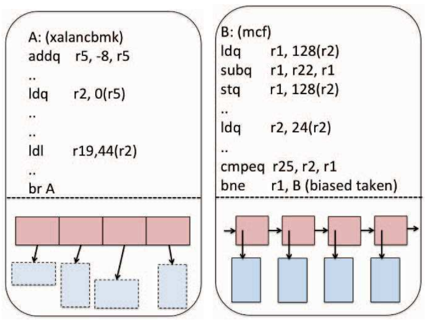


Figure 4. P1 Prefetcher.

A second type of access pattern that we target involves pointer, that is, the address of a later access depends on the outcome of an earlier one. Earlier prefetcher designs have targeted such accesses (*e.g.*, [7]), but one challenge is to achieve timely prefetches. There are two special patterns of pointer accesses that lend themselves to timely prefetching with relatively simple finite state machines. For notational convenience, we call this component P1 (Figure 4).

1) Array of pointers: The first pattern is an offshoot of the strided access pattern: the address of a later access is the value from a strided access stream (plus a constant offset). Figure 5-a shows a real code snippet that generates this pattern and an illustration of the logical data structure. The implementation of the prefetcher is straightforward, especially in the presence of T2.

In the detection phase, we search for load instructions whose address (transitively) depends on a strided load. To find out dependent loads of a particular strided memory instruction i, we use a simple taint propagation circuit at the decoder. A bit vector corresponding to all logical registers will first be cleared. Then a single bit corresponding to the destination register of instruction i will be set. From then on, if an instruction has a source register that is tainted/set in the bit vector, the corresponding bit of its destination register will be set. Otherwise, the destination register bit will be cleared. This process will stop when instruction i is encountered again. During this process, any load instruction that is tainted will be a candidate for our pointer access pattern. Suppose instruction j is such a candidate. We then check to see if j's address is always a constant offset from the value of i. This checking process is similar to that used in detecting constant stride in T2. Specifically, every iteration, we keep the *value* of instruction i and calculate the delta between it and the address of instruction j. If the delta remains constant for a number of iterations (4 in all experiments in this paper), we mark i as special strided pointer instruction in the (expanded) stride identifier table (SIT) and keep the corresponding delta found (between i's value and j's address).



```
A: (xalancbmk)
addq   r5, -8, r5
..
ldq    r2, 0(r5)
..
..
ldl    r19,44(r2)
..
br A
```

```
B: (mcf)
ldq    r1, 128(r2)
subq   r1, r22, r1
stq    r1, 128(r2)
..
ldq    r2, 24(r2)
..
cmpeq  r25, r2, r1
bne    r1, B (biased taken)
```

(a) (b)

Figure 5. P1 target access patterns.

In the steady state, when instruction i executes, in addition to the stride prefetching, the value of instruction i will be delivered to P1, which then adds the delta and issues another prefetch. Note that once instruction i is identified as a strided pointer instruction, not just a plain strided instruction any more, its prefetch distance will be doubled to compensate for the back-to-back nature of i and j.

2) Pointer chains: The second pattern P1 targets is a more classic pointer-chain pattern as shown in Figure 5-b. Identifying this pattern is very similar to the tainting-based approach just discussed: If memory instruction i's address register transitively depends on its own destination register (from the previous iteration) then it forms the pointer-chain pattern. Prefetching the main list (accessed by instruction i) can be thought of as a variation of prefetching for the strided access: in addition to adding a delta ($A_{n+1} = A_n + \Delta$), we need to access the memory ($A_{n+1} = M[A_n + \Delta]$). This creates two main differences in the design of the finite state machine (FSM). First, unlike prefetching for strided accesses, in pointer-chain prefetching, the FSM can only issue the next prefetch after the previous prefetch returns the value. So in the initial "catch-up" stage (before reaching the proper prefetching distance), the FSM for strided access pattern simply issues, say, one prefetch every cycle. The FSM for pointer chain will have to wait for the previous prefetch to return. In the steady state (after reaching the proper prefetching distance), the value from the previous prefetch will be stored. The next prefetch will be issued when the next instance of the triggering instruction executes.

A second difference concerns the correction mechanism. It is possible that the address stream deviates from the actual access stream after some iterations. This could be the result of control flow inside the iteration. For strided access pattern, the design is largely self-correcting as in the steady state, the FSM takes the current address and adds proper adjustments. For pointer-chain, once we are on the wrong track, it is possible that the FSM will continue to prefetch along the wrong track and generate only pollution. We see no such situations in our experiments. We believe this is an unlikely situation in the real world and any reasonable solution to prevent continuous pollution is perhaps sufficient. One solution is to keep one prefetch address in the SIT and compare it to the actual addresses from upcoming iterations of the corresponding memory instructions. If no match is found in a time-out period (say, after m iterations), the state of the memory instruction can be reset to test for the pointer-chain pattern again.

C. Targeting High Spatial Locality Streams

Some regions of memory demonstrate sufficiently high spatial locality that any number of patterns may show good matching merely due to coincidence. Clearly a high-quality prefetcher component that matches the underlying access pattern with high accuracy is still superior to one with lower accuracy. Until such a high-quality component is invented, these regions can be targeted by a more simple-minded design that brings in every line in the vicinity. We call our implementation of such a “carpet bombing” prefetcher component C1 (Figure 6).

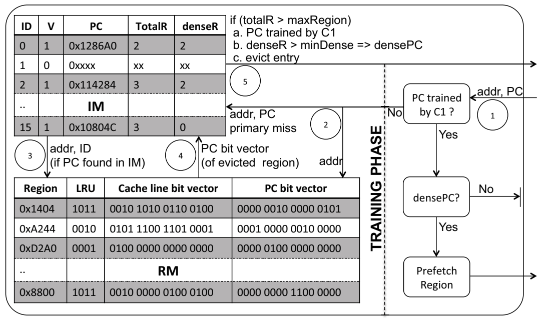


Figure 6. C1 Prefetcher.

Fetching an entire region to the cache is effectively making the cache line longer. In this paper, a region is thus just a super cache line with 16 constituent cache lines in them. To track the spatial locality in the region, we use a Region Monitor (RM), which contains multiple entries (16 in this paper), each tracking a different region. The entry contains a tag and a cache line bit vector to track each cache line within the region. On every cache access, if the region is present in RM, the corresponding bit is set. If the region is not monitored in RM, an invalid or a victim entry is selected.

In our design we try to associate a high locality access stream with instructions. This is facilitated by another structure called the Instruction Monitor (IM). When we start to monitor a candidate instruction, a new entry in IM will be allocated. The entry will remain until a decision is made about the candidate instruction. Thus there is no eviction of entries in IM. To tie a monitored instruction with the region it touches, each entry in RM contains a PC bit vector, wherein each bit corresponds to one entry in IM. When a monitored instruction accesses region r, we find out its IM entry ID (say k); go to the entry corresponding to r in RM; and set the k^{th} bit in the instruction vector of the entry. This way, the instruction vector of an entry in RM tells us which instructions (currently being monitored by IM) have accessed this region. When the region entry is evicted, we will update every instruction that has touched this region as follows. Each entry in IM has two counters: $TotalRegions$ and $DenseRegions$. The former is always incremented; and the latter only if the evicted region is dense (more than six bits in its cache line vector are set). When $TotalRegions$ reaches a certain threshold value (4 in this paper), we make a decision about the instruction (and vacate the entry in IM for another candidate). If the instruction is found to access a dense region with a high probability ($> 3/4$), it is marked as such. When such instructions execute in the future, C1 will trigger region prefetch.

D. Coordinator Design

In an ideal world, experts possess not only specific expertise, but also the recognition of its boundaries. If prefetcher components also clearly recognize their boundary of effectiveness, the coordinator only needs to aggregate this knowledge and stratify accesses for each component. In reality, the design of the coordinator depends a lot on the availability and idiosyncrasies of the (non-ideal) components. A thorough exploration of the topic is premature at this moment. We discuss two conjectures and how they lead to a design instance in our current example.

- Expertise can be measured: Even with potentially overlapping expertise from different components, we can measure the effective accuracy of each component and pick the best performing component for each pattern.
- Patterns are tied to static instructions: If so, at least we can empirically characterize the prefetch accuracy of a component for the subset of accesses generated by one static instruction. This will allow us to empirically and probabilistically establish a reasonable division of labor based on static instructions.

Note that these conjectural principles only suggest that a first-effort coordinator can be constructed. There are most certainly issues that need investigations. At the same time, the coordinator also present new optimization opportunities. For instance, different components may demonstrate different prefetching accuracies and cache pollution. In a shared-

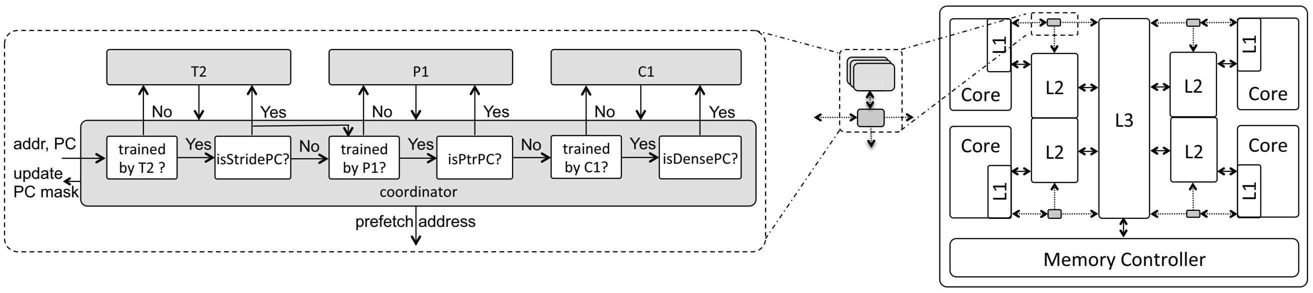


Figure 7. A schematic of the proposed composite prefetcher.

cache environment, the competition for shared resources expands to include prefetches from different threads. The coordinators should take all these factors into account and adjust various parameters of prefetching such as the aggressiveness of prefetching and the destination cache level of the prefetches.

In our example implementation, the coordinator of our three components is straightforward. Both T2 and P1 are already instructions based. Generally speaking, they only identify instructions they can handle and achieve good results. In other words, they recognize the boundary of their expertise, which makes division of labor easy. Thus our controller is a hardwired decision logic that presents a memory instruction to each component in turn. Since T2 targets more cases, we start with T2; and try P1 when T2 does not handle it; and finally try C1. In other words, this particular controller uses no additional storage and just combinational logic to steer accesses to different components as shown in Figure 7. Finally, for both T2 and P1, the high accuracy warrants prefetching all the way to L1 while the lower accuracy of C1 suggests that L2 is a more appropriate target. This is the policy the coordinator implements.

E. Existing Prefetchers as Components

The ultimate merit of a composite prefetcher clearly depends on the quality of its underlying components. While it takes further explorations to develop these components, existing prefetchers can also serve as ready-made components. However, as components, they are far from ideal, especially the typical pattern-based prefetchers. The reason is two-fold.

First, a typical pattern-based prefetcher derives significant results from prefetching canonical streams. As we will show later, we find T2 to be a much more compelling component for canonical streams. Thus, when we use existing prefetchers alongside T2, we are really using them beyond the core competency.

Second, their prefetching beyond canonical strided streams is usually not carefully optimized (if at all). Indeed, there may not even be a simple, coherent pattern of accesses (beyond canonical strided streams) that the prefetcher captures well. When we use a number of existing prefetchers simultaneously, the challenge at the coordinator level is significant: it is hard to tell who should get the job. If we blindly allow all to try, there is a higher chance that some prefetchers will get the right prefetch, but there is perhaps a much higher degree of pollution to offset and even negate the benefit.

It is worth repeating the emphasis: we advocate a different way of building prefetchers and existing monolithic prefetchers are decidedly poor examples of prefetcher components. Using them as a component is in general inefficient and brings in extra design challenges. On the other hand, they point to possible future target patterns.

In our experiments making use of existing prefetchers as components, we use the following heuristics for the coordinator. First, we want to identify the component most suitable to a particular access pattern to handle it. Second, once a component is identified, relevant accesses are filtered out from other components so as to minimize erroneous prefetches. In our experimental setup, we find our three components to have higher accuracies than monolithic prefetchers we experimented with. The coordinator thus only pass on accesses from instructions not recognized by T2, P1, or C1 to other components. When there are more than one other components, we first figure out the appropriate component to handle the access pattern from a particular memory instruction as follows. We distribute the accesses in a round robin fashion to each component. The prefetched lines will be tagged by the identity of the component issuing the prefetch. When a demand access hits a prefetched line, we will use the component that brought in the line to handle the instruction going forward.

V. Experimental Analysis

A. Experimental Setup

To quantitatively analyze the behavior of various prefetchers, we use an execution-driven simulator gem5 [3]. Table I summarizes the configuration for the tested systems. We perform experimental analyses on a diverse set of workloads, including SPEC CPU2006 benchmark suite (reference input), a graph application suite (CRONO [1], using graph input data structures from google, amazon, twitter, math-

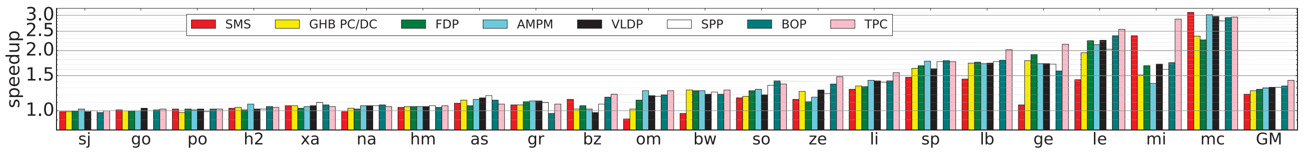


Figure 8. Comparison of speedup of individual prefetchers. Applications are sorted in increasing average performance gain from all prefetchers. Note that the vertical axis is in log scale.

overflow, and california road-networks), embedded applications (STARBENCH [2] with large inputs), and scientific workloads (NPB using C class workloads). We use individual applications in a single-core environment and 4-thread mixes randomly drawn from the above suites for a 4-core environment. To reduce the simulation time we generate five simpoints [28] per benchmark, each with an interval of 10M instructions. All the results reported are obtained from these simpoints.

Core Parameters: 1-4 Cores, OoO, 4-wide, 3.0GHz, 192 ROB, 96 LSQ, 128INT/128FP PRF, 4INT/ 2MEM/ 4FP FUs, L-Tage (1+12 Components + 256-Entry Loop Predictor [27]), 4K Entry BTB, 32-entry RAS, 15 cycle branch miss penalty
Private L1: Split I/D, 64KB, 4-way, 64B blocks, 3ports, 1ns, 32MSHRs, LRU
Private L2: 256kB, 8-way, 2 ports, 3ns, 32MSHRs, LRU
Shared L3: 2MB/Core, 16-way, 12ns, LRU
Main Memory: 4GB, DDR3 1600MHz, 2 channels, 2 ranks/channel, 8 banks/rank, t$_{RCD}$=13.75ns, t$_{RAS}$=35ns, t$_{FAW}$=30ns, t$_{WTR}$=7.5ns, t$_{RP}$=13.75ns

Table I. Processor Configuration.

GHB-PC/DC [22]	4KB	256 Entry GHB, 256 Entry Index Table
SPP [17]	5KB	256 Entry ST, 512 Entry PT, 1024 Entry PF, 8 Entry GHR
VLDP [29]	3.25KB	64 Entry DHB, 128 Entry DPT, 128 Entry OPT
BOP [20]	4KB	1K Entry RR Table, 1Kb Prefetch bits
FDP [32]	2.5KB	1Kb Tag Array, 8Kb Bloom Filter, 64 streams
SMS [31]	12KB	64 Entry AT, 32 Entry FR, 512 Entry PHT, 1 PR
AMPM [12]	4KB	128 Access Maps, 256b per Map
T2	2.3KB	32 Entry SIT, 2KB state bits in I-cache and LH (1 Entry LR and 16 Entry NLPCT)
P1	1.07KB	1Entry PtrPC, 8 Entry SIT, TPU(64bits), 1KB state bits
C1	1.2KB	16Entry IM (640 bits), 16 Entry RM (1248 bits), 1KB state bits
TPC	4.57KB	T2 + P1 + C1

Table II. Storage cost of evaluated prefetchers.

B. Overall Effect of the Example Implementation

The main conjecture of this paper is that through proper division of labor among prefetching components, we can build composite prefetchers that are more efficient and more effective than conventional monolithic prefetchers. We have yet to fully explore the design space of composite prefetchers. The design used in this paper is thus but an example of a potentially very diverse design space.

We first discuss the bottom-line result: can composite prefetchers be more effective and efficient. This configuration takes the three components discussed in Sec. IV: T2, P1, and C1. For notational convenience, we will refer to this prefetcher as TPC for short. For comparison, a number of commonly used or recent prefetchers (GHB-PC/DC [22], FDP [32], VLDP [29], SPP [17], BOP [20], AMPM [12], and SMS [30]) are also included. The configurations of all these are specified in Table II. Note that with the exception of SMS, the tested prefetchers all use a small amount of storage and their performance are insensitive to additional storage. We first focus on SPEC result as it is the most commonly used suite to evaluate prefetcher designs. We can see that our simulated performance of the comparison prefetchers show broad agreement with published results.

Effectiveness: Figure 8 shows the speedups of TPC and monolithic prefetchers over a baseline microarchitecture (without prefetching) for the entire suite of benchmarks. The first thing to note from the figure is that TPC is noticeably more effective than the state-of-the-art monolithic prefetchers. The geometric mean speedup of TPC is 1.41, compared to 1.21 to 1.33 for monolithic designs. In other words, TPC is 6% faster than its nearest competitor. Secondly, TPC is broadly effective. It is the best-performing prefetcher in 11 out of 21 benchmarks and performs within 5% of the best performing prefetcher for the rest of the benchmarks.

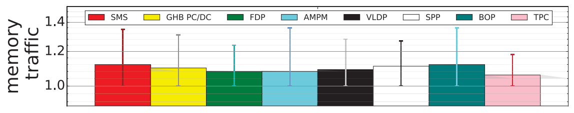


Figure 9. Comparison of normalized memory traffic of individual prefetchers. We show the suite-wide geometric mean as bars and the range of all applications as "I-beams" on top of the bar.

Efficiency: Figure 9 compares the total memory traffic of the system under different prefetchers. For each benchmark, we normalize the traffic to that of the baseline system without prefetching. On average, memory traffic overhead under TPC is 6%, the least among tested hardware prefetchers. The next best-performing prefetcher (BOP) has an overhead of 12%. To put this into perspectives, even in a decoupled look-ahead system with a dedicated, full-custom look-ahead thread, there is a traffic overhead of about

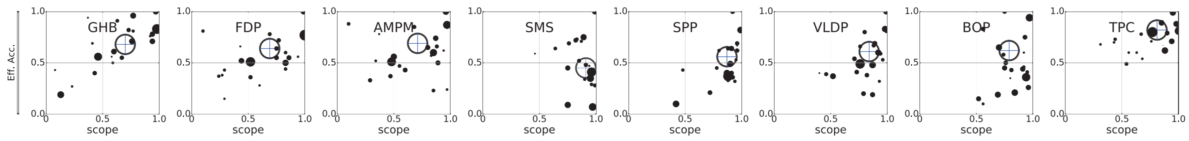


Figure 10. Effective accuracy vs scope for different prefetchers. Each small circle in the plots represents one application. The size (area) of the dot is proportional to the number of prefetches issued. The hollow dot with a cross in each plot summarizes the suite-wide average, weighted by the size of each small dot. The per-prefetcher summary is shown as a big circle with a cross.

4% [10], [19].

Different workloads: Next we broaden the testing benchmarks, which include running application mixes in a multicore environment, where the speedup is measured as weighted speedup for each application. With these new tests, the general conclusion remains the same. As summarized in Figure 11, TPC consistently outperforms its competitors, albeit with varying effectiveness. Taking the geometric mean result of all 68 workloads, TPC achieves a speedup of 1.39 compared to 1.22-1.31 for the other seven prefetchers.

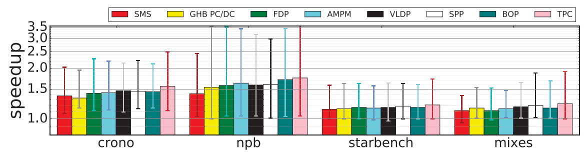


Figure 11. Comparison of speedup of individual prefetchers with different benchmark suites.

C. In-depth Analysis

To sum, our composite prefetcher TPC obtains noticeably better performance compared to the state-of-the-art monolithic prefetchers while generating less traffic. While we certainly believe this particular design is worth considering by processor designers, the central argument of this paper is rather on the design *methodology*. Specifically, as conventional prefetcher design matures, it is increasingly difficult to simultaneously improve prefetching scope and accuracy with a single idea. We believe it is a better practice to decouple the goals of improving each and design composite prefetchers with multiple specialized components. With such a methodology, we lower the entry barrier of new ideas. In the following, we will perform a number of experiments to show some supporting evidence for this view.

1) Coverage, accuracy, and scope: As discussed in Sec. III, we use scope to quantify a prefetcher's attempt without considering its actual achieved result. We use effective accuracy to fully account for the pollution of prefetches. In Figure 10, we plot the effective accuracy of L1 cache [5] and scope for every benchmark under every prefetcher.

[5]Note that many prefetchers are not designed to prefetch into L1. But in almost all designs, the version prefetching into L1 is (slightly) better in overall speed than prefetching into L2. We will discuss this point more later.

In the figure, each box shows one prefetcher. Within the box, each dot represents the result of a benchmark. To get a suite-wide average, we calculate weighted-average from each benchmark, with the miss per kilo instruction (MPKI) as the weight. This way, applications with more misses are given more weight in evaluating prefetcher behavior. To show the weight visually, the area of each dot is proportional to the weight.

The first thing to note is the sheer range of accuracy in essentially all monolithic prefetchers. The worst-performing application for each monolithic prefetcher achieves an effective accuracy between 7% and 23%. On average, the effective accuracy of the monolithic prefetchers ranges from 45% to 69%. In contrast, TPC has a much more limited range, with the worst application having an effective accuracy of 49% and the global average is 82%. To see this a bit better, we summarize the global average of effective accuracy of each prefetcher in Figure 12. For TPC, we show the effect as we add one component after another. Similar to effective accuracy, we define effective coverage as the percentage reduction of misses as a result of using a prefetcher. We show effective accuracy and coverage for both L1 and L2 caches. We see that the effective coverage of TPC is significantly better than monolithic prefetchers in the L1 cache. The difference is much smaller in the L2 cache. Despite issuing fewer prefetches, TPC has higher effective coverage. This is because of better accuracies.

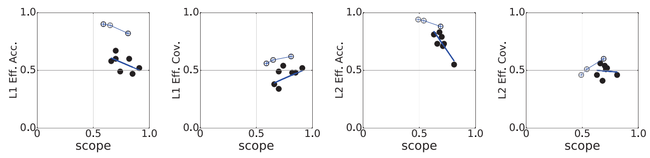


Figure 12. Effective accuracy and coverage vs scope for different prefetchers at L1 and L2 caches. Monolithic prefetchers are shown in solid dots. Composite prefetcher TPC's result is shown in hollow circles from left to right as we incrementally turn on T2, P1, and C1 components.

Intuitively, as we increase the scope, we are facing more difficult targets, which lowers the effective accuracy. This intuition largely agrees with empirical results. For TPC, as we go from strided stream, to pointer chasing, to high locality streams, the accuracy of prefetches issued understandably drops. In monolithic prefetchers, those with a high scope tend to have a lower effective accuracy. The line in the figure

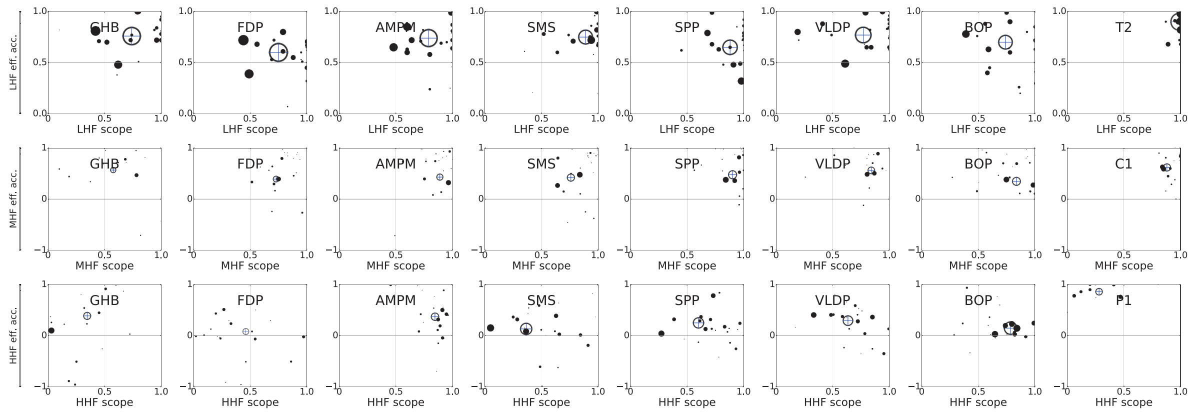


Figure 13. Effective accuracy vs scope by different prefetchers (left to right) in three categories: from top to bottom LHF, MHF, and HHF. For one application using one prefetcher, one dot shows the effective accuracy and scope in the corresponding category.

shows the result of linear regression. As we increase the scope, some of the drop in accuracy may be inevitable. But for the current target, there is an accuracy gap between TPC and the monolithic prefetchers.

To understand this accuracy gap better, we divide all accesses subjectively into three categories with increasing difficulty of prefetch and see the accuracies in each category. We call these categories low-, mid-, and high-hanging fruit (LHF, MHF, and HHF for short). They correspond to strided accesses, non-strided accesses with high spatial locality, and everything else. Note that the division is done offline to have a better approximation to "ground truth". Nevertheless, it is still a somewhat subjective division. Every prefetch issued is labeled as one of the categories.

To properly account for overall impact of each prefetcher's effort in these categories, we follow every prefetch and try to account for its pollution impact. Specifically, any prefetched line is marked. If it serves an on-demand access later, the line earns a positive credit. If it causes an additional miss, then it earns a negative credit. We maintain an additional set of cache tags, which track the alternative reality where no prefetch is issued. When an access misses in the cache but finds its tag in the alternative-reality cache tags, we have a prefetching-induced miss. In this case, one negative credit is equally divided among the prefetched lines currently in the set.

Figure 13 shows the results of this analysis. Several observations can be made about these results:

1) We can see that for most prefetchers, the vast majority of prefetches belong to LHF (canonical strided streams). Having a higher accuracy can be an important factor in the overall performance of the prefetcher. In this seemingly simple category, there is still substantial variation across prefetcher as well as applications, suggesting potential optimization opportunities. For this category of targets, T2 offers noticeably better accuracies and is a compelling component for a composite prefetcher.
2) Many stride prefetchers are designed to capture variations of canonical strided streams. Whether the design actually does a good job capturing these variations remains unclear. We can see that the almost all monolithic prefetchers register a very high scope in the MHF category. While they do show reasonably good effective accuracies (between 32% and 56% on average, C1's effective accuracy is 61%, which is noticeably better. In other words, to explain the access behavior, C1's access pattern is arguably a better hypothesis than other patterns targeted in those prefetchers.
3) Finally, HHF is indeed a more challenging category. While for LHF all application-prefetcher pairs show positive effective accuracy, and for MHF, only a few points show up in the negative range for effective accuracy, for HHF, there are many more points. In other words, for all monolithic prefetchers, there are often many applications, where on the balance, the prefetches issued for this category are counter-productive. Indeed, some points hover around -1 effective accuracy, which means for those cases, almost every single prefetch in this category is harmful – not just useless! The overall average is only 38% for the *best* monolithic prefetcher. P1, in contrast, has a comparatively higher effective accuracy 86%. On the other hand, its scope is relatively limited.

Note that not only do incorrect prefetches hurt performance, they also present a number of other challenges in a monolithic prefetcher. First, there is the subtle impact of increasing design efforts. When designing a monolithic prefetcher, results from these categories are lumped together, making it hard to isolate the reason for lower accuracies.

Second, consider a multicore environment where both cache space and memory bandwidth could be precious resources. Issuing highly speculative prefetches simply should not be a local decision. Take their HHF performance for example, if we take the monolithic prefetcher with the lowest average effective accuracy of 8%, it suggests that for every 12 prefetches issued in this category, only one miss is avoided. Yet, these highly inefficient prefetches are distributed among all other prefetches, making it hard to make judicious global tradeoffs.

By separating the prefetch of different patterns into different components, it is much easier to attribute problems to a particular component. It makes improving the design easier as we can add components to increase scope and upgrade individual components to improve accuracy. This also makes it easier to perform high-level control dynamically in shared-resource environments. Though this is largely outside the scope of this paper, we note the following result. We change the memory controller such that when it is forced to drop a request, (when the queue fills up) it chooses low-probability prefetches (in our case from the C1 component). Compared to the default option where the memory controller randomly drops prefetches, this change alone accounts for an average of 6% performance gain in a multicore enviroment.

2) Existing prefetchers as components: As discussed in Sec. IV-E, future designs of components will improve the overall capability of a composite prefetcher. Existing prefetchers serve as possible examples, though they are far from ideal. If an existing prefetcher design has better accuracies than one of our components in its scope of prefetch, we can replace the component. If the design covers additional scope that we do not cover, then they can serve as an additional component. Given the set of prefetchers we have experimented, we have no example of the former case, but almost all designs offer some additional scope. Here we take a few examples: VLDP, SPP, FDP, and SMS.

Adding these components all contribute to a small performance improvement, which we will show in more detail later in Figure 15. But first, we highlight one underlying benefit of the overall division-of-labor approach. And that is efficiency. Each of these existing prefetchers use some storage to track access patterns in order to generate prefetch. Through division of labor, when we use them as a component, they are only focusing on the prefetching scope beyond what we already covered with TPC. This frees up some resources and in theory can make the design work better. Figure 14 illustrates this effect.

In this figure, we zoom into the region TPC does not cover. We observe the effective accuracy and scope of these prefetchers in this region in two states: working alone or as an additional component to TPC. In all cases, there is an improvement in effective accuracy when the prefetcher is used as a component as indicated by the arrows. In fact, when used as standalone prefetchers, in each case, many applications have an overall negative accuracy in this region. When used as components, all accuracies are positive. Overall, the improvement is clearly noticeable. For instance,

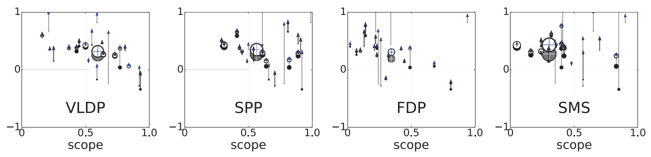


Figure 14. Effective accuracy (y-axis) and scope for different prefetchers working alone (solid circle) vs as a component in addition to TPC (hollow circle). Each pair of circles that represent the same application are linked with a line. The per-prefetcher summary is shown as a big circle with a cross inside. This summary is a weighted average of per-application result where the weight is the number of prefetches issued. All circles are drawn with their area proportional to the weight.

in SMS, the effective accuracy is 43% as a component compared to 27% of a standalone prefetcher. This example shows that division of labor can already help a somewhat general-purpose design do its job better. We conjecture that if the design is further specialized to purposefully target the region of interest, the benefit will be even more pronounced. Finally, there is an improvement in scope as well, but the magnitude is too small to be noticed.

3) Division of labor and stratification: An important point of forming a composite prefetcher is that through *division* of labor, each component can focus on its strengths. This is different from having multiple prefetchers working in parallel (or shunting). Though they both increase prefetching scope, the latter has overlapping efforts instead of a division of labor. To see the difference, we compare compositing with shunting, where components are unaware of each other.

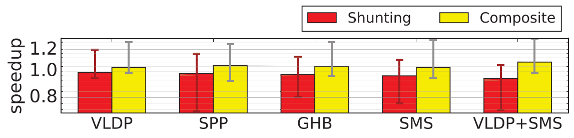


Figure 15. The effect of shunting vs compositing an existing prefetcher with TPC.

In Figure 15, we compare the performance impact of compositing and shunting, both using the same components TPC and the existing prefetchers VLDP, SPP, FDP, and SMS. We normalize all performance to that of using TPC alone. We show the suite-wide average as bars and the range of all applications as "I-beams" on top of the bar. The difference is clear and similar across all experiments. When compositing an existing prefetcher with TPC, the performance is never worse than TPC alone and on average somewhat better than TPC by 3-8%. On the other hand, if we shunt the two prefetchers, the result is almost always worse than having TPC alone. On average, it is clearly worse (by 1-6%). The division of labor is performed by the coordinator, which implicitly depends on the knowledge of each component's expected accuracy. In a shared-resource environment, this knowledge can help improve resource utilization. Even within a single thread context, this knowledge can help too, for instance, in deciding prefetch destination.

As discussed earlier, in monolithic prefetchers we used in this paper, we prefetch to L1 cache. This is after verifying that for most prefetchers, on average, it is better than prefetching only into L2. However, if we stratify the accesses into LHF, MHF, and HHF, we can make individual decisions about prefetch destination. We found that LHF usually has sufficiently high prefetch accuracy for all prefetchers and on average benefits from prefetching into L1. For the rest, prefetching into L2 is better on average. Figure 16 shows the effect in more detail. Unfortunately for the monolithic prefetchers, the stratification of accesses into LHF, MHF, and HHF is only an analysis mechanism similar to having an oracle. In a real implementation, there is no such information to help decide prefetch destination. In TPC, the components naturally performs a stratification (which is reasonably accurate) and we can thus decide prefetch destination based on which component issues the prefetch.

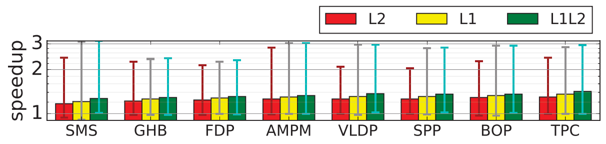


Figure 16. Effect of different prefetch destination. From left to right, the bars show the average speedup when the prefetch destination is L2, L1, and when it depends on the category of the access. The I-beam on each bar shows the range of speedups in the suite.

Recap: Based on the various analyses presented above, we can gain some insights as to why a composite prefetcher such as TPC outperforms state-of-the-art monolithic prefetchers and what future work can further improve such a design.

1) Monolithic prefetchers all have high prefetching scope. But in the effort to target more, there is a sacrifice in accuracy.
2) In contrast, TPC maintains a high accuracy that more than compensate for a more limited prefetching scope. Thus, it achieves better performance impact with less traffic demand.
3) TPC currently lacks in HHF scope, suggesting more components targeting this area will be helpful. Existing prefetchers as a group perform rather poorly at the moment. However, when they are composited with TPC, they already improve its effectiveness in targeting HHF, thanks to efficiency gained through proper division of labor. Further specialization is likely to deliver additional benefits.

VI. Conclusion

Prefetchers are important components in modern microarchitectures. Most proposals of prefetch designs are what we call monolithic ones, where a single heuristic is used to observe access patterns and predict future accesses accordingly. Intuitively, a monolithic design faces inherent tradeoffs in scope and accuracy and may not be the best approach. Instead, we believe building composite prefetchers through division of labor among multiple, specialized prefetcher components may be a more effective approach.

In this paper, we have shown a composite prefetcher (called TPC) with three components targeting strided accesses, a subset of pointer-chasing accesses, and high spatial locality accesses. Our design is effective, achieving 1.41 speedup compared to that of 1.21-1.33 from an array of state-of-the-art prefetchers. It is also efficient, increasing traffic by 6% compared to 8-12% in other prefetchers. While this design is a high-performance prefetcher, it is only a proof-of-concept example of a different approach to prefetcher design. In this approach, we decouple the goals of high accuracy and scope, achieving the former with focused, specialized components and the latter with a coordinated composite of components.

Through in-depth analyses of TPC and conventional monolithic prefetchers, we gain a number of insights about the behavior of the prefetchers which can help guide future work improving designs. First, we can see that TPC achieves superior result mainly through higher accuracies, thanks to its components dedicated each to simpler patterns than conventional designs. The higher accuracies in turn allow easier division of labor, which makes the overall composite prefetcher effective. Second, TPC's effectiveness can be further improved by increasing its scope and further improving the components' accuracy. Adding existing prefetchers as a component already shows tangible benefits.

References

[1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *IEEE International Symposium on Workload Characterization*, pages 44–55, 2015.

[2] M. Andersch, B. Juurlink, and C. Chi. A benchmark suite for evaluating parallel programming models. In *Proceedings of Workshop on Parallel Systems and Algorithms (PARS)*, volume 28, pages 1–6, 2013.

[3] N. Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[4] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth Prefetching. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, October 2006.

[5] C. F. Chen, Se-Hyun Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2004.

[6] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[7] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, Content-Directed Data Prefetching Mechanism. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, March 2010.

[8] P. Diaz and M. Cintra. Stream chaining: Exploiting multiple leveles of correlation in data prefetching. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[9] I. Ganusov and M. Burtscher. Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 350–360, September 2005.

[10] A. Garg and M. Huang. A Performance-Correctness Explicitly Decoupled Architecture. In *Proceedings of the International Symposium on Microarchitecture*, pages 306–317, November 2008.

[11] I. Hur and C. Lin. Memory prefetching Using Adaptive Stream Detection. In *Proceedings of the International Symposium on Microarchitecture*, December 2006.

[12] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 2011.

[13] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the International Symposium on Microarchitecture*, December 2013.

[14] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the International Symposium on Computer Architecture*, June 1997.

[15] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the International Symposium on Computer Architecture*, pages 364–373, May 1990.

[16] J. Kim, Teran E, P. Gratz, D. Jimnez, S. Pugsley, and C. Wilkerson. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, April 2017.

[17] J. Kim, S. Pugsley, P. Gratz, A. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *Proceedings of the International Symposium on Microarchitecture*, October 2016.

[18] S. Kondguli and M. Huang. T2: A Highly Accurate and Energy Efficient Stride Prefetcher. In *IEEE International Conference on Computer Design*, 2017.

[19] S. Kondguli and M. Huang. A Case for a More Effective, Power-Efficient Turbo Boosting. *ACM Transactions on Architecture and Code Optimization*, 2018.

[20] P. Michaud. Best-offset hardware prefetching. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, March 2016.

[21] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. Ac/dc: An adaptive data cache prefetcher. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, September 2004.

[22] K. J. Nesbit and J. E. Smith. Data Cache Prefetching using a Global History Buffer. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2004.

[23] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–33, April 1994.

[24] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the International Symposium on Computer Architecture*, June 2015.

[25] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 115–126, October 1998.

[26] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the International Symposium on Computer Architecture*, May 1999.

[27] A. Seznec. A 256 kbits l-tage branch predictor. In *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, pages 1–6, 2007.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 45–57, October 2002.

[29] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the International Symposium on Microarchitecture*, December 2015.

[30] S. Somogyi, T. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-Temporal Memory Streaming. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[31] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. In *Proceedings of the International Symposium on Computer Architecture*, June 2006.

[32] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2003.

[33] Z. Wang, D. Burger, S McKinely, S. Reinhardt, and C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.

[34] T. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for address-correlated prefetching. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2009.

[35] T. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. *ACM SIGARCH Computer Architecture News*, 33(2):222–233, 2005.

[36] V. Young and A. Krishna. Towards Bandwidth-Efficient Prefetching with Slim AMPM. In *The 2nd Data Prefetching Championship*, 2015.