Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems

Eiman Ebrahimi† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

Abstract

Linked data structure (LDS) accesses are critical to the performance of many large scale applications. Techniques have been proposed to prefetch such accesses. Unfortunately, many LDS prefetching techniques 1) generate a large number of useless prefetches, thereby degrading performance and bandwidth efficiency, 2) require significant hardware or storage cost, or 3) when employed together with stream-based prefetchers, cause significant resource contention in the memory system. As a result, existing processors do not employ LDS prefetchers even though they commonly employ stream-based prefetchers.

This paper proposes a low-cost hardware/software cooperative technique that enables bandwidth-efficient prefetching of linked data structures. Our solution has two new components: 1) a compiler-guided prefetch filtering mechanism that informs the hardware about which pointer addresses to prefetch, 2) a coordinated prefetcher throttling mechanism that uses run-time feedback to manage the interference between multiple prefetchers (LDS and stream-based) in a hybrid prefetching system. Evaluations show that the proposed solution improves average performance by 22.5% while decreasing memory bandwidth consumption by 25% over a baseline system that employs an effective stream prefetcher on a set of memory- and pointer-intensive applications. We compare our proposal to three different LDS/correlation prefetching techniques and find that it provides significantly better performance on both single-core and multi-core systems, while requiring less hardware cost.

1. Introduction

As DRAM speed improvement continues to lag processor speed improvement, memory access latency remains a significant system performance bottleneck. As such, mechanisms to reduce and tolerate memory latency continue to be critical to improving system performance. Prefetching is one such mechanism: it attempts to predict the memory addresses a program will access, and issue memory requests to them before the program flow needs the data. In this way, prefetching can hide the latency of a memory access since the processor either does not incur a cache miss for that access or it incurs a cache miss that is satisfied earlier (because prefetching already started the memory access). Prefetchers that deal with streaming (or striding) access patterns have been researched for decades [12, 18, 27] and are implemented in many existing processor designs [13, 38, 10]. Aggressive stream prefetchers can significantly reduce the effective memory access latency of many workloads. However, costly last-level cache misses do not always adhere to streaming access patterns. Access patterns that follow pointers in a linked data structure (i.e., chase pointers in memory) are an example. Since pointer-chasing access patterns are common in real applications (e.g., databases [9] and garbage collection [21]), prefetchers that are able to efficiently predict such patterns are needed. Our goal in this paper is to develop techniques that 1) enable the efficient prefetching of linked data structures and 2) efficiently combine such prefetchers with commonly-employed stream-based prefetchers.

To motivate the need for prefetchers for linked data structures (LDS), Figure 1 (top) shows the performance improvement of an aggressive stream prefetcher and the fraction of last-level cache misses it prefetches (i.e. coverage) on a set of workloads from the SPEC 2006, SPEC 2000, and Olden benchmark suites. The stream prefetcher significantly improves the performance of five benchmarks. However, in eight of the remaining benchmarks (`mcf`, `astar`, `xalancbmk`, `omnetpp`, `ammp`, `bisort`, `health`, `pfast`), the stream prefetcher eliminates less than 20% of the last-level cache misses. As a result, it either degrades or does not affect the performance of these benchmarks. In these eight benchmarks, a large fraction of the cache misses are caused by non-streaming accesses to LDS that cannot be prefetched by a stream prefetcher. Figure 1 (bottom) shows the potential performance improvement possible over the aggressive stream prefetcher if all last-level cache misses due to LDS accesses were *ideally* converted to cache hits using oracle information. This ideal experiment improves average performance by 53.7% (37.7% w/o health), showing that significant performance potential exists for techniques that enable the prefetching of linked data structures.

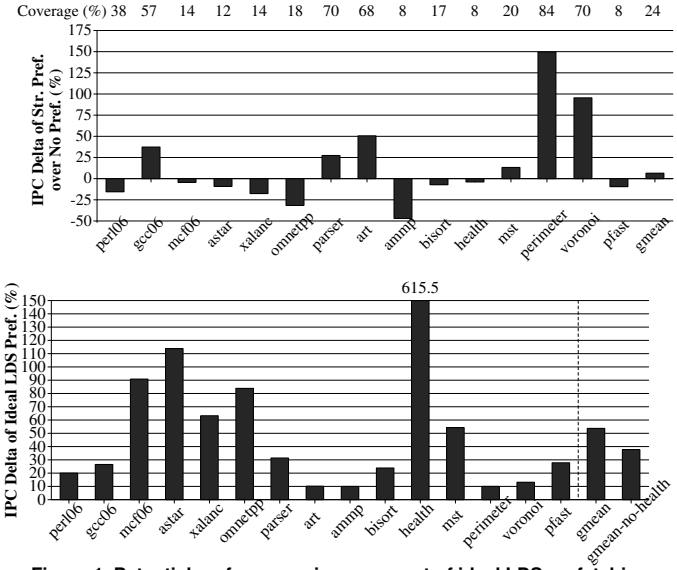


Figure 1. Potential performance improvement of ideal LDS prefetching

Previous work [5, 17, 30, 7, 31, 9, 43, 23] proposed techniques that prefetch non-streaming accesses to LDS. Unfortunately, many of these prefetchers have not found widespread acceptance in current designs because they have one or both of the following two major drawbacks that make their implementation difficult or costly:

1- **Large storage/hardware cost:** Some LDS prefetchers need very large storage to be effective because they usually need to store the pointers that will be prefetched.[1] Examples include jump pointer prefetchers [31], the pointer cache [7], and hardware correlation prefetchers [5, 17, 20]. Since the pointer working set of applications is usually very large, keeping track of it in a hardware structure requires a large amount of storage. Other, pre-execution based, LDS prefetchers (e.g., [43, 23, 6, 8]) are also costly because they require an extra thread context or pre-computation hardware to execute helper threads. As energy and power consumption becomes more pressing

[1] By "prefetching a pointer", we mean issuing a prefetch request to the address the pointer points to.

with each processor generation, simple prefetchers that require small storage cost and no additional thread context become desirable and necessary.

2- **Large number of useless prefetch requests:** Many LDS prefetchers (e.g., [5, 17, 9]) generate a large number of requests to effectively prefetch pointer addresses. An example is content-directed prefetching (CDP) [9]. CDP is attractive because it requires neither state to store pointers nor a thread context for pre-execution. Instead, it greedily scans values in accessed cache blocks to discover pointer addresses and generates prefetch requests for *all pointer addresses*. Unfortunately, such a greedy prefetch mechanism wastes valuable memory bandwidth and degrades performance due to many useless prefetches and cache pollution. The large number of generated useless prefetch requests makes such LDS prefetchers undesirable, especially in the bandwidth-limited environment of multi-core processors.

Designing a Hybrid Prefetching System Incorporating LDS Prefetching: This paper first proposes a technique that overcomes the problems mentioned above to make LDS prefetching low-cost and bandwidth-efficient in a hybrid prefetching system. To this end, we start with content-directed prefetching, which is stateless and requires no extra thread context, and develop a technique that reduces its useless prefetches. Our technique is hardware/software cooperative. The compiler, using profile and LDS data layout information, determines which pointers in memory could be beneficial to prefetch and conveys this information as hints to the content-directed prefetcher. The content-directed prefetcher, at run-time, uses the hints to prefetch beneficial pointers instead of indiscriminately prefetching all pointers. The resulting LDS prefetcher is *low hardware-cost and bandwidth-efficient*: it neither requires state to store pointer addresses nor consumes a large amount of memory bandwidth.

Second, since an efficient LDS prefetcher is not intended for prefetching streaming accesses, any real processor implementation requires such a prefetcher to be used in conjunction with an aggressive stream prefetcher, which is already employed in modern processors. Unfortunately, building a hybrid prefetcher by naively putting together two prefetchers places significant pressure on memory system resources. Prefetch requests from the two prefetchers compete with each other for valuable resources, such as memory bandwidth, and useless prefetches can deny service to useful ones by causing resource contention. If competition between the two prefetchers is not intelligently managed, both performance and bandwidth-efficiency can degrade and full potential of the prefetchers cannot be exploited. To address this problem, we propose a technique to efficiently manage the resource contention between the two prefetchers: our mechanism throttles the aggressiveness of the prefetchers intelligently based on how well they are doing in order to give more memory system resources to the prefetcher that is more effective at improving performance. The resulting technique is a bandwidth-efficient hybrid (streaming and LDS) prefetching mechanism.

Our evaluation in Section 6 shows that the combination of the techniques we propose in this paper (*efficient content-directed LDS prefetching* and *coordinated prefetcher throttling*) improves average performance by 22.5% (16% w/o health) while also reducing average bandwidth consumption by 25% (27.1% w/o health) on a state-of-the-art system employing an aggressive stream prefetcher.

Contributions: We make the following major contributions:

1. We propose a very low-hardware-cost mechanism to bandwidth-efficiently prefetch pointer accesses without requiring any storage for pointers or separate thread contexts for pre-execution. Our solution is based on a new compiler-guided technique that determines which pointer addresses to prefetch in content-directed LDS prefetching. To our knowledge, this is the first solution that enables us to build not only very low-cost but also bandwidth-efficient, yet effective, LDS prefetchers by overcoming the fundamental limitations of content-directed prefetching.

2. We propose a hybrid prefetching mechanism that throttles multiple different prefetchers in a coordinated fashion based on run-time feedback information. To our knowledge, this is the first proposal to intelligently manage scarce off-chip bandwidth and inter-prefetcher interference cooperatively between different types of prefetchers (e.g., LDS and stream prefetchers). This mechanism can be used in conjunction with any form of hybrid prefetching.

3. We show that our proposal is effective for both single-core as well as multi-core processors. We extensively compare our proposal to previous techniques and show that it significantly outperforms hardware prefetch filtering and three other forms of LDS/correlation prefetching, while requiring less hardware storage cost.

2. Background and Motivation

We briefly describe our baseline stream-based prefetcher and content-directed prefetching since our proposal builds upon them. We also describe the shortcomings of content-directed prefetching that motivate our mechanisms.

2.1. Baseline Stream Prefetcher Design

We assume that any modern system will implement stream (or stride) prefetching, which is already commonly used in existing systems [13, 10, 38]. Our baseline stream prefetcher is based on that of the IBM POWER4/POWER5 prefetcher, which is described in more detail in [38, 36]. The prefetcher brings cache blocks into the L2 (last-level) cache, since we use an out-of-order execution machine that can tolerate short L1-miss latencies. How far ahead of the demand miss stream the prefetcher can send requests is determined by the *Prefetch Distance* parameter. *Prefetch Degree* determines how many requests the prefetcher issues at once. A detailed description of our prefetcher can be found in [36].

2.2. Content-Directed Prefetching (CDP)

Content directed prefetching (CDP) [9] is an attractive technique for prefetching LDS because it does not require additional state to store the pointers that form the linkages in an LDS. This mechanism monitors incoming cache blocks at a certain level of the memory hierarchy, and identifies candidate addresses to prefetch within those cache blocks. To do so, it uses a virtual address matching predictor, which relies on the observation that most virtual addresses share common high-order bits. If a value in the incoming cache block has the same high-order bits as the address of the cache block (the number of which is a static parameter of the prefetcher design; Cooksey et al. [9] refer to these bits as *compare bits*), the value is predicted to be a virtual address (pointer) and a prefetch request is generated for that address. This prefetch request first accesses the last-level cache; if it misses, a memory request is issued to main memory.

CDP generates prefetches recursively, i.e. it scans prefetched cache blocks and generates prefetch requests based on the pointers found in those cache blocks. The depth of the recursion determines how aggressive CDP is. For example, a maximum recursion depth of 1 means that prefetched cache blocks will *not* be scanned to generate any more prefetches.

2.3. Shortcomings of Content-Directed Prefetching

Although content-directed prefetching is attractive because it is stateless, there is a major deficiency in its identification of addresses to prefetch, which reduces its usefulness. The intuition behind its choice of candidate addresses is simple: if a pointer is loaded from memory, there is a good likelihood that the pointer will be used as the data address of a future load. Unfortunately, this intuition results in a significant deficiency: CDP generates prefetch requests for *all identified pointers* in a scanned cache block. Greedily prefetching *all pointers* results in low prefetch accuracy and significantly increases bandwidth consumption because *not all* loaded pointers are later used as load addresses by the program.

Figure 2 and Table 1 demonstrate the effect of this deficiency on the performance, bandwidth consumption, and accuracy of CDP. Figure 2 shows the performance and bandwidth consumption (in terms of BPKI - bus accesses per thousand retired instructions) of 1) using the baseline stream prefetcher alone, and 2) using both the baseline stream prefetcher and CDP together.[2] Adding CDP to a system with a stream prefetcher significantly reduces performance (by

[2]For this experiment we use the same configuration as that of the original CDP proposal [9], which is described in Section 5.

Benchmark	perlbench	gcc	mcf	astar	xalancbmk	omnetpp	parser	art	ammp	bisort	health	mst	perimeter	voronoi	pfast
CDP Accuracy (%)	28.0	6.0	1.4	29.1	0.9	8.4	13.3	1.9	22.3	3.4	58.9	1.4	83.3	47.0	37.4

Table 1. Prefetch accuracy of the original content-directed prefetcher

14%) and increases bandwidth consumption (by 83.3%). Even though CDP improves performance in several applications (`gcc`, `astar`, `health`, `perimeter`, and `voronoi`), it causes significant performance loss and extra bandwidth consumption in several others (`mcf`, `xalancbmk`, `bisort`, and `mst`). These effects are due to CDP's very low accuracy for these benchmarks (shown in Table 1), caused by indiscriminate prefetching of all pointer addresses found in cache lines. Cache pollution resulting from useless prefetches is the major reason why CDP degrades performance. In fact, we found that if cache pollution were eliminated *ideally* using oracle information, CDP would improve performance by 29.4% and 30.4% on `bisort` and `mst` respectively.

To provide insight into the behavior of CDP, we briefly describe why it drastically degrades performance in `bisort`. Section 3 provides a detailed explanation of the performance degradation in `mst`. `bisort` performs a bitonic sort of two disjoint sets of numbers stored in binary trees. As a major part of the sorting process, it swaps subtrees very frequently while traversing the tree. Upon a cache miss to a tree node, CDP prefetches pointers under the subtree belonging to the node. When this subtree is swapped with another subtree of a separate node, the program starts traversing the newly swapped-in subtree. Hence, *almost all of the previously prefetched pointers are useless* because the swapped-out subtree is not traversed. Being unaware of the high-level program behavior, CDP indiscriminately prefetches pointers in scanned cache blocks, significantly degrading performance and wasting bandwidth in such cases.

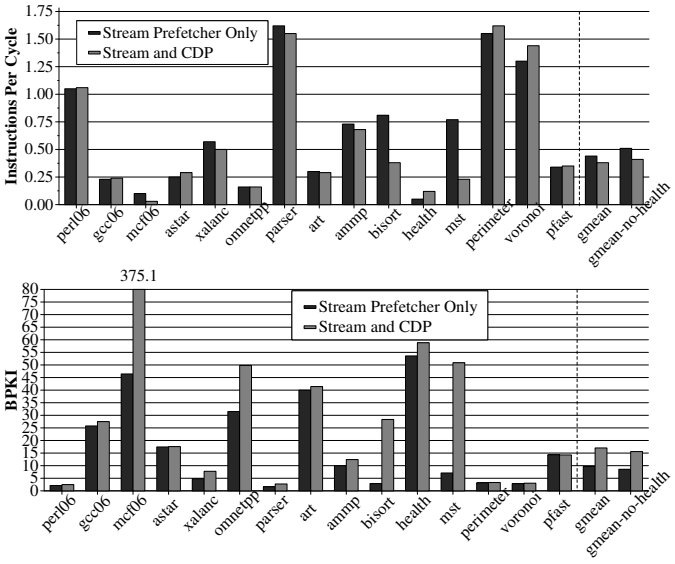


Figure 2. Effect of the original CDP on performance and memory bandwidth

Our goal: In this paper, we aim to provide an effective, bandwidth-efficient, and low-cost solution to prefetching linked data structures by 1) overcoming the described deficiencies of the content-directed prefetcher and 2) incorporating it efficiently in a hybrid prefetching system. To this end, we propose techniques for efficient content-directed LDS prefetching (Section 3) and hybrid prefetcher management via coordinated throttling of prefetchers (Section 4).

3. Efficient Content-Directed LDS Prefetching

The first component of our solution to efficient LDS prefetching is a compiler-guided technique that *selectively* identifies which pointer addresses should be prefetched at run-time. In our technique, efficient CDP (ECDP), the compiler uses its knowledge of the location of pointers in LDS along with its ability to gather profile information about the usefulness of prefetches to determine which pointers would be beneficial to prefetch. The content-directed LDS prefetcher, at run-time, uses this information to prefetch beneficial pointers instead of indiscriminately prefetching all pointers.

Terminology: We first provide the terminology we will use to describe ECDP. Consider the code example in Figure 3(a). The load labeled LD1 accesses the data cache to obtain the `data` field of the `node` structure. When this instruction generates a last-level cache miss, the cache block fetched for it is scanned for pointers by the content-directed prefetcher. Note that the pointers that exist in the accessed node (i.e., the `left` and `right` pointers) are always at the same offset from the byte LD1 accesses. For example, say LD1 accesses bytes 0, 16, and 32 respectively in cache blocks 1, 2, and 3, as shown in Figure 3(b). The `left` pointer of the node LD1 accesses is always at an offset of 8 from the byte LD1 accesses (i.e., the `left` pointer is at bytes 8, 24, and 40 in cache blocks 1, 2, 3 respectively). If different nodes are allocated consecutively in memory (as shown in the figure), then each pointer field of any other node in the same cache block is also at a constant offset from the byte LD1 accesses.

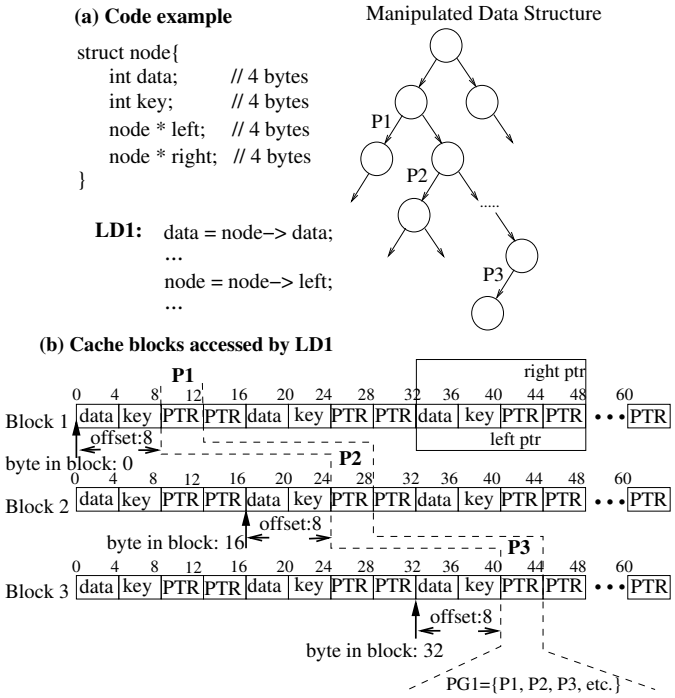


Figure 3. Example illustrating the concept of Pointer Groups (PGs)

Hence, the pointers in a cache block are almost always at a constant offset from the address accessed by the load that fetches the block.[3] For our analysis, we define a **Pointer Group**, *PG(L, X)*, as follows: PG(L, X) is the set of pointers in all cache blocks fetched by a load instruction L that are at a constant offset X from the data address L accesses. The example in Figure 3(b) shows PG(LD1, 8), which consists of the pointers P1, P2, P3. At a program-level abstraction, each PG corresponds to a pointer in the code. For example, PG1 in Figure 3(b) corresponds to `node->left`.

Usefulness of Pointer Groups: We define a *PG's prefetches* to be the set of all prefetches CDP generates (including recursive prefetches) to prefetch *any* pointer belonging to that PG. For example, in Figure 3, the set of all prefetches generated to prefetch P1, P2, P3 (and any other pointer belonging to PG1) form PG1's prefetches. Figure 4 shows the breakdown of all the PGs in the shown workloads into those whose majority (more than 50%) of prefetches are useful,[4] and those whose

[3] We say "almost always" because dynamic memory allocations and deallocations can change the layout of pointers in the cache block.

[4] We found PG's with less than 50% useful prefetches usually result in performance loss. Figure 10 provides more detailed analysis of PGs.

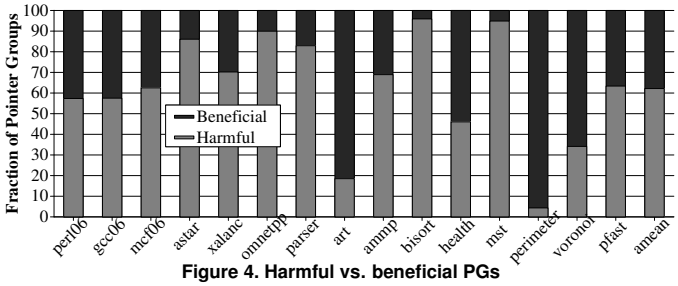


Figure 4. Harmful vs. beneficial PGs

majority of prefetches are useless. We name the former *beneficial PGs* and the latter *harmful PGs*.

Figure 4 shows that, in many benchmarks (e.g. `astar`, `omnetpp`, `bisort`, `mst`), a large fraction of the PGs are harmful. Generating prefetch requests for such PGs would likely waste bandwidth and reduce performance. To motivate ECDP, Figure 5 provides insight into where harmful PGs come from. This figure shows a code portion and cache block layout from the `mst` benchmark. The example shows a hash table, consisting of an array of pointers to linked lists of nodes. Each node contains a key, multiple data elements, and a pointer to the next node. The program repetitively attempts to find a particular node based on the key value, using the HashLookup function shown in Figure 5(a). Figure 5(c) shows a sample layout of the nodes in a cache block fetched into the last-level cache when a miss happens on the execution of `ent->Key!=Key`. Conventional CDP would generate prefetch requests for all the pointers in each incoming cache block. This is inefficient because, among the PGs shown in Figure 5, prefetches generated by PG1 and PG2 (i.e., `D1` and `D2`) will almost always be *useless*, but those generated by PG3 (i.e., `Next`) could be *useful*. This is because only one of the linked list nodes contains the key that is being searched. Therefore, when traversing the linked list, it is more likely that each iteration of the traversal accesses the `Next` node (because a matching key is *not* found) rather than accessing a data element of the node (as a result of a key match in the node). In our mechanism, we would like to enable the prefetches due to PG3, while disabling those due to PG1 and PG2.

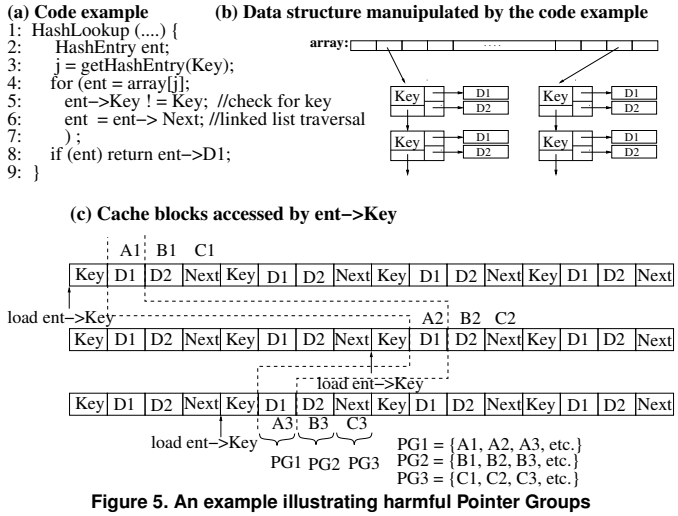


Figure 5. An example illustrating harmful Pointer Groups

ECDP Mechanism: We use a profiling compiler to distinguish beneficial and harmful PGs. The compiler profiles the code and classifies each PG as harmful/beneficial based on the accuracy of the prefetches the PG generates in the profiling run. Using this classification, the compiler provides hints to the content-directed prefetcher. At runtime, the content-directed prefetcher uses these hints such that it generates prefetch requests only for pointers in beneficial PGs.

To accomplish this, the compiler attributes a number of PGs to each static load instruction. For example, the static load instruction missing in the cache block shown in Figure 5(c) will have PGs PG1, PG2 and PG3 associated with it. During the profiling step, the compiler gathers usefulness information about the PGs associated with each load instruction in the program. The compiler informs the hardware of *beneficial* PGs of each load using a hint bit vector. This bit vector must be long enough to hold a bit for each possible pointer in a cache block. For example, with a 64-byte cache block and 4-byte addresses, the bit vector is 16 bits long. Figure 6 illustrates the information contained in the bit vector. If the n*th* bit of the bit vector is set, it means that the PG at offset 4×n from the address accessed by the load is beneficial. This bit vector is conveyed to the microarchitecture as part of the load instruction, using a new instruction added to the target ISA which has enough hint bits in its format to support the bit vector.[5]

At runtime, when a demand miss happens, the content-directed prefetcher scans the fetched cache block and consults the missing load's hint bit vector. For a pointer found in the cache block, CDP issues a prefetch request only if the bit vector indicates that prefetching that pointer is beneficial. For example, the bit vector shown in Figure 6 has bit positions 2, 6 and 11 set. When a load instruction misses in the last-level cache and accesses the shown cache block at byte 12 in the block, CDP will only make prefetch requests for pointers it finds at offsets 8 (4×2), 24 (4×6), and 44 (4×11) from byte 12 (corresponding to bytes 20, 36 and 56 in the block).[6] Note that this compiler-guided mechanism is used only on cache blocks that are fetched by a load demand miss. If the cache block is fetched as a result of a miss caused by a content-directed prefetch, our mechanism prefetches *all* of the pointers it finds in that cache block.

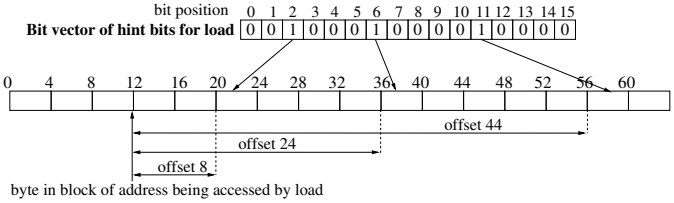


Figure 6. Correspondence of hint bits to pointers in a fetched cache block

Profiling Implementation: The profiling step needed for our mechanism can be implemented in multiple ways. We briefly sketch two alternative implementations. In one approach, the compiler profiles the program by simulating the behavior of the cache hierarchy and prefetcher of the target machine. The simulation is used to gather usefulness information of the PGs. Note that this profiling approach does not require a detailed timing simulation of the processor: it requires only enough simulation of the cache hierarchy and the prefetcher to determine the usefulness of PGs.

In another approach, the target machine can provide support for profiling, e.g. using informing load operations [14]. With this support, the compiler detects whether a load results in a hit or miss and whether the hit is due to a prefetch request. During the profiling run, the compiler constructs the usefulness of each PG. Due to space limitations we do not describe this implementation in more detail.

4. Managing Multiple Prefetchers: Incorporating Efficient CDP in a Hybrid Prefetching Scheme

Since stream-based prefetchers are very effective and already employed in existing processors, ECDP should be used in combination with stream prefetching. Unfortunately, naively combining these prefetchers (or any two prefetchers) together can be problematic. The two prefetchers contend for the same memory subsystem resources and as a result can deny service to each other. In particular, prefetches from one prefetcher can deny service to prefetches from another due to resource contention, i.e., by 1) occupying memory request buffer entries, 2) consuming DRAM bus bandwidth, 3) keeping DRAM banks busy for a long time, and 4) evicting cache blocks fetched by another

[5] According to our evaluations, adding such a new instruction has a negligible effect on both code size and instruction cache miss rate.

[6] Without loss of generality, the shown bit vector encodes only positive offset values. Negative offset values could also exist. E.g., a pointer at byte 0 would be at an offset of -12 with respect to the byte the load accesses. In our implementation, we use a negative bit vector as well.

prefetcher from the last-level cache before they are used. In our evaluation, we found that resource contention increases the average latency of useful prefetch requests by 52% when the two prefetchers are used together compared to when each is used alone.

Resource contention between prefetchers can result in either performance degradation or the inability to exploit the full performance potential of using multiple prefetchers. In addition, it can significantly increase bandwidth consumption due to increased cache misses and conflicts in DRAM banks/buses between different prefetcher requests. Therefore, we would like to decrease the negative impact of resource contention by managing the sharing of the memory system resources between multiple prefetchers.

We propose throttling the aggressiveness of each prefetcher in a coordinated fashion using dynamic feedback information. We use the accuracy and coverage of each prefetcher as feedback information that is input to the logic that decides the aggressiveness of both prefetchers. We first explain how this feedback information is collected (Section 4.1). Then, we describe how this information is used to guide the heuristics that throttle the prefetchers (Section 4.2). Note that, even though we mainly evaluate it for the combination of ECDP and stream prefetchers, the proposed coordinated throttling mechanism is a general technique that can be used to coordinate prefetch requests from any two prefetchers.

4.1. Collecting Feedback Information

Our mechanism uses the coverage and accuracy of each prefetcher as feedback information. To collect this information, two counters per prefetcher are maintained: 1) *total-prefetched* keeps track of the total number of issued prefetch requests, 2) *total-used* keeps track of the number of prefetch requests that are used by demand requests. To determine whether a prefetch request is useful, the tag entry of each cache block is extended by one *prefetched* bit per prefetcher, *prefetched-CDP* and *prefetched-stream*. When a prefetcher fetches a cache block into the cache, it sets the corresponding *prefetched* bit. When a demand request accesses a prefetched cache block, the *total-used* counter is incremented and both *prefetched* bits are reset. In addition, we maintain one counter, *total-misses* that keeps track of the total number of last-level cache misses due to demand requests. Using these counters, accuracy and coverage are calculated as follows:

$$(1) \quad Accuracy = \frac{total\text{-}used}{total\text{-}prefetched}$$

$$(2) \quad Coverage = \frac{total\text{-}used}{total\text{-}used \; + \; total\text{-}misses}$$

We use an interval-based sampling mechanism similar to that proposed in [36] to update the counters. To take into account program phase behavior, we divide data collection into intervals. We define an interval based on the number of cache lines evicted from the L2 cache. A hardware counter keeps track of this number, and an interval ends when the counter exceeds some statically defined threshold (8192 in our experiments). At the end of an interval, each counter is updated as shown in Equation 3. Then, *CounterValueDuringInt* is reset. Equation 3 gives more weight to the program behavior in the most recent interval while taking into account the behavior in all previous intervals. Accuracy and coverage values calculated using these counters are used to make throttling decisions in the following interval.

$$(3) CounterValue = \frac{1}{2} \, CounterValueAtTheBeginningOfInt + \frac{1}{2} \, CounterValueDuringInt$$

4.2. Coordinated Throttling of Multiple Prefetchers

Table 2 shows the different aggressiveness levels for each of the prefetchers employed in this study. Each prefetcher has 4 levels of aggressiveness, varying from very conservative to aggressive. The aggressiveness of the stream prefetcher is controlled using the *Prefetch Distance* and *Prefetch Degree* parameters (described in Section 2.1). We use the *maximum recursion depth* parameter of the CDP to control its aggressiveness as defined in Section 2.2.

Aggressiveness Level	Stream Prefetcher *Distance*	Stream Prefetcher *Degree*	Content-Directed Prefetcher *Maximum Recursion Depth*
Very Conservative	4	1	1
Conservative	8	1	2
Moderate	16	2	3
Aggressive	32	4	4

Table 2. Prefetcher Aggressiveness Configurations

The computed prefetcher coverage is compared to a single threshold $T_{coverage}$ to indicate high or low coverage. The computed accuracy is compared to two thresholds A_{high} and A_{low} and the corresponding accuracy is classified as high, medium, or low. Our rules for throttling the prefetchers' aggressiveness are based on a set of heuristics shown in Table 3. The same set of heuristics are applied to throttling both prefetchers. The throttling decision for each prefetcher is made based on its own coverage and accuracy *and* the other prefetcher's coverage.[7] In the following explanations and in Table 3, the prefetcher that is being throttled is referred to as the *deciding prefetcher*, and the other prefetcher is referred to as the *rival prefetcher*. For example, when the stream prefetcher throttles itself based on its own accuracy/coverage and CDP's coverage, we refer to the stream prefetcher as the deciding prefetcher and the CDP as the rival prefetcher.

Heuristics for Coordinated Prefetcher Throttling: When the deciding prefetcher has high coverage (case 1), we found that decreasing or not changing its aggressiveness results in an overall decrease in system performance (regardless of its accuracy or the rival prefetcher's coverage).[8] In such cases, we throttle the deciding prefetcher up to keep it at its maximum aggressiveness to avoid losing performance. When the deciding prefetcher has low coverage and low accuracy we throttle it down to avoid unnecessary bandwidth consumption and cache pollution (case 2). If the deciding prefetcher has low coverage, and so does the rival prefetcher, and the deciding prefetcher's accuracy is medium or high, we increase the aggressiveness of the deciding prefetcher to give it a chance to get better coverage using a more aggressive configuration (case 3). When the deciding prefetcher has low coverage and medium or low accuracy, and the rival prefetcher has high coverage, we throttle down the deciding prefetcher (case 4). Doing so allows the rival prefetcher to make better use of the shared memory subsystem resources because the deciding prefetcher is not performing as well as the rival. On the other hand, if the deciding prefetcher has low coverage and *high* accuracy, and the rival prefetcher has high coverage, we do not change the aggressiveness of the deciding prefetcher (case 5). In this case, the deciding prefetcher is not throttled down because it is highly accurate. However, it is also not throttled up because the rival prefetcher has high coverage, and throttling up the deciding prefetcher could interfere with the rival's useful requests.

Case	Deciding Prefetcher Coverage	Deciding Prefetcher Accuracy	Rival Prefetcher Coverage	Deciding Prefetcher Throttling Decision
1	High	-	-	Throttle Up
2	Low	Low	-	Throttle Down
3	Low	Medium or High	Low	Throttle Up
4	Low	Low or Medium	High	Throttle Down
5	Low	High	High	Do Nothing

Table 3. Heuristics for Coordinated Prefetcher Throttling

$T_{coverage}$	A_{low}	A_{high}
0.2	0.4	0.7

Table 4. Thresholds used for coordinated prefetcher throttling

Table 4 shows the thresholds we used in the implementation of coordinated prefetcher throttling. These values are determined em-

[7] We refer to increasing a prefetcher's aggressiveness (by a level) as throttling it *up* and decreasing its aggressiveness as throttling it *down*.

[8] If a prefetcher has high coverage in a program phase, it is unlikely that its accuracy is low. This is because coverage will decrease if the accuracy is low, since more last-level cache misses will be generated due to polluting prefetches.

pirically but not fine tuned. The small number of parameters used in our mechanism makes it feasible to adjust the values to fit a particular system. For example, in systems where off-chip bandwidth is limited (e.g., systems with a large number of cores on the chip), or where there is more contention for last-level cache space (e.g., the last-level cache is relatively small or many cores share the last-level cache), $T_{coverage}$ and A_{low} can be increased to trigger Case 2 of Table 3 sooner in order to keep bandwidth consumption and cache contention of prefetchers in check. In addition, due to the prefetcher-symmetric and prefetcher-agnostic setup of our throttling heuristics in Table 3, the proposed scheme can potentially be used with more than two prefetchers. Each prefetcher makes a decision on how aggressive it should be based on its own coverage/accuracy and the coverage of other prefetchers in the system. The use of throttling for more than two prefetchers is part of ongoing work and is out of the scope of this paper.

5. Experimental Methodology

We evaluate the performance impact of the proposed techniques using an execution-driven x86 simulator. We model both single core and multi-core (2 and 4 core) systems. We model the processor and the memory system in detail, faithfully modeling port contention, queuing effects, bank conflicts at all levels of the memory hierarchy, including the DRAM system. Table 5 shows the parameters of each core. Each baseline core employs the aggressive stream prefetcher described in Section 2.1. Unless otherwise specified, all single-core performance results presented in this paper are normalized to the IPC of the baseline core. Note that our baseline stream prefetcher is very effective: it improves average performance by 25% across all SPEC CPU2006/2000 and Olden benchmarks compared to no prefetching at all.

Execution Core	Out of order, 15 (fetch, decode, rename stages) stages, decode/retire up to 4 instructions, issue/execute up to 8 μ-instructions 256-entry reorder buffer; 32-entry ld-st queue; 256 physical registers
Front End	fetch up to 2 branches; 4K-entry BTB; 64-entry return address stack; hybrid BP: 64K-entry gshare, 64K-entry PAs, 64K-entry selector
On-chip Caches	L1 I-cache: 32KB, 4-way, 2-cycle, 1 rd port, 1 wr port; L1 D-cache: 32KB, 4-way, 4-bank, 2-cycle, 2 rd ports, 1 wr port; L2 cache: 1MB, 8-way, 8 banks, 15-cycle, 1 read/write port; LRU replacement and 128B line size, 32 L2 MSHRs
Memory	450-cycle minimum memory latency; 8 memory banks; 8B-wide core-to-memory bus at 5:1 frequency ratio;
Prefetcher	Stream prefetcher [38, 36] with 32 streams, prefetch degree 4, distance 32; 128-entry prefetch request queue per core
Multi-core	each core has a private L2 cache, on-chip DRAM controller, memory request buffer size = 32 * (core-count)

Table 5. Baseline processor configuration

Benchmarks: We classify a benchmark as pointer-intensive if it gains at least 10% performance when all LDS accesses are ideally converted to hit in the L2 cache on our baseline processor. For most of our evaluations we use the pointer-intensive workloads from SPEC CPU2006, CPU2000 and Olden [29] benchmark suites, which consists of 14 applications. We also evaluate one application from the bioinformatics domain, `pfast` (parallel fast alignment search tool) [3]. `pfast` is a pointer-intensive workload used to identify single nucleotide and structural variation of human genomes associated with disease. Section 6.7 presents results for the remaining applications in the suites. Since `health` from the Olden suite skews average results, we state average performance gains with and without this benchmark throughout the paper.[9]

All benchmarks were compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. SPEC INT2000 benchmarks are run to completion with a reduced input set [19]. For SPEC2006/SPEC FP2000 benchmarks, we use a representative sample of 200M instructions obtained with a tool we developed using the SimPoint [32] methodology. Olden benchmarks are all run to completion using the input sets described in [24]. For profiling, we use the train input set of SPEC benchmarks and a smaller training input set for Olden benchmarks.

Workloads for Multi-Core Experiments: We use 12 multiprogrammed 2-benchmark SPEC2006 workloads for the 2-core experiments and 4 4-benchmark SPEC2006 workloads for the 4-core experiments. The 2-core workloads were randomly selected to combine both pointer-intensive and non-pointer-intensive benchmarks. The 4-core workloads are used as case studies: one workload has 4 pointer-intensive benchmarks, 2 workloads are mixed (2 intensive, 2 non-intensive), and one workload is non-pointer-intensive (1 pointer-intensive combined with 3 non-intensive).

Prefetcher Configurations: In the x86 ISA, pointers are 4 bytes. Thus, CDP compares the address of a cache block with 4-byte values read out of the cache block to determine pointers to prefetch, as Section 2.2 describes. Our CDP implementation uses 8 bits (out of the 32 bits of an address) for the *number of compare bits* parameter and 4 levels as the *maximum recursion depth* parameter (described in Section 2.2). We found this CDP configuration to provide the best performance. Section 2.1 describes the stream prefetcher configuration. Both prefetchers fetch data into the L2 cache.

6. Experimental Evaluation

6.1. Single-Core Results and Analyses

6.1.1. Performance Figure 7 (top) shows the performance improvement of our proposed techniques. The performance of each mechanism is normalized to the performance of the baseline processor employing stream prefetching. On average, the combination of our mechanisms, ECDP with coordinated prefetcher throttling (rightmost bars), improves performance over the baseline by 22.5% (16% w/o health), thereby making content-directed LDS prefetching effective.

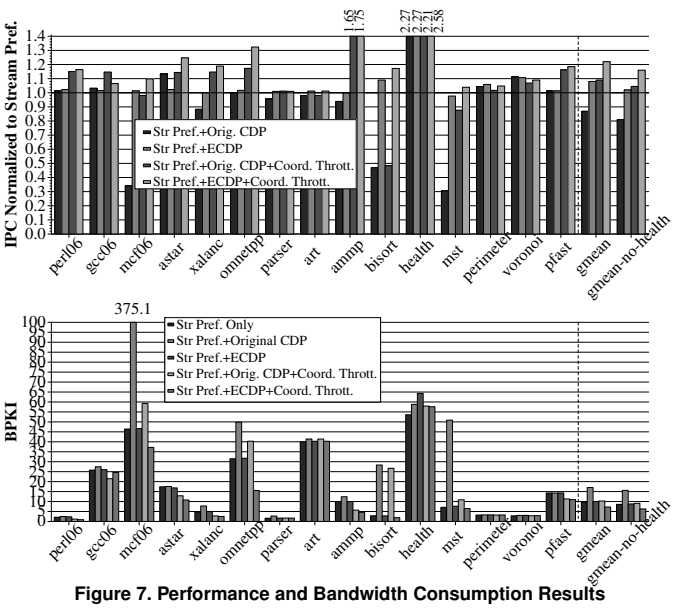


Figure 7. Performance and Bandwidth Consumption Results

Several observations are in order from Figure 7. First, the original CDP (leftmost bars) improves performance on benchmarks such as `astar`, `gcc`, `health`, `perimeter`, and `voronoi`, but significantly degrades performance on `mcf`, `xalancbmk`, `bisort`, and `mst`. In the latter, the original CDP generates a very large number of prefetch requests and has very low accuracy (see Figure 8). As a result, the original CDP causes cache pollution and significantly degrades performance. In fact, it degrades average performance by 14% due to its useless prefetches.[10]

[9] Zilles [42] shows that the performance of `health` benchmark from the Olden suite can be improved by orders of magnitude by rewriting the program. We do not remove this benchmark from our evaluations since previous work commonly used this benchmark and some of our evaluations compares previous LDS prefetching proposals to ours. However, we do give health less weight by presenting average results without it.

[10] The original CDP proposal [9] showed that CDP improved average performance on a set of selected traces. Our results show that CDP actually degrades performance on pointer-intensive SPEC 2000/2006 and Olden applications. We believe the difference is due to the different evaluated applications.

	perlb.	gcc	mcf	astar	xalan.	omnet.	parser	art	ammp	bisort	health	mst	perim.	voron.	pfast	**gmean**	**gmean-no-health**
IPC Δ (%)	**16.3**	**6.5**	**9.8**	**24.7**	**18.9**	**32.4**	1.0	1.3	**74.9**	**17.2**	**158.4**	3.9	4.8	**9.0**	**18.5**	**22.5**	**16**
BPKI Δ	**-56.3**	-4.5	**-20.1**	**-38.1**	**-47.8**	**-50.6**	4.3	0.7	**-53.6**	**-33.3**	7.5	-8.7	1.7	2.3	**-23.3**	**-25.0**	**-27.1**

Table 6. Change in IPC performance and BPKI with our proposal (ECDP and coordinated prefetcher throttling combined)

Second, our compiler-guided selective LDS prefetching technique, ECDP (second bars from the left), improves performance by reducing useless prefetches (and cache pollution) due to CDP in many benchmarks, thereby providing an 8.6% (2.7% w/o health) performance improvement over the baseline. The large performance degradations in `mcf`, `xalancbmk`, `bisort`, and `mst` are eliminated by using the hints provided by the compiler to detect and disable prefetching of harmful pointer groups. Benchmarks such as `bisort`, `health`, and `perimeter` significantly gain performance due to the increased effectiveness of useful prefetches enabled by eliminating interference from useless prefetches. Even though ECDP is effective at identifying useful prefetches (as described in more detail in Section 6.1.5), we found that in most of the remaining benchmarks ECDP *alone* does not improve performance because aggressive stream prefetcher's requests interfere with ECDP's useful prefetch requests. Our coordinated prefetcher throttling technique is used to manage this interference and increase the effectiveness of both prefetchers.

Third, using coordinated prefetcher throttling by itself with the original CDP and the stream prefetcher (third bars from left) improves performance by reducing useless prefetches, and increasing the benefits of useful prefetches from both prefetchers. This results in a net performance gain of 9.4% (4.5% w/o heath).

Finally, ECDP and coordinated prefetcher throttling interact positively: when employed together, they improve performance by 22.5% (16% w/o health), significantly more than when each of them is employed alone. Eleven of the fifteen benchmarks gain more than 5% from adding coordinated prefetcher throttling over ECDP. In `perlbench`, `bisort` and `health`, throttling improves the effectiveness of ECDP because the stream prefetcher throttles itself down as it has lower coverage than CDP (due to case 4 in Table 3). This amplifies the benefits of useful ECDP prefetches by getting useless stream prefetches out of the way in the memory system. In `gcc`, ECDP throttles itself down because the stream prefetcher has very high coverage (57% as shown in Figure 1(left)). This decreases contention caused by ECDP prefetches and allows the stream prefetcher to maintain its coverage of cache misses. In `astar`, `mcf`, `omnetpp`, and `mst`, the stream prefetcher has both low coverage and low accuracy. As a result, the stream prefetcher throttles itself down, eliminating its detrimental effects on the effectiveness of ECDP.

We conclude that the synergistic combination of ECDP and coordinated prefetcher throttling makes content-directed LDS prefetching very effective and allows it to interact positively with stream prefetching. Hence, our proposal enables an effective hybrid prefetcher that can cover both streaming and LDS access patterns.

6.1.2. Off-Chip Bandwidth Figure 7 (bottom) shows the effect of our techniques on off-chip bandwidth consumption. ECDP with coordinated prefetcher throttling reduces bandwidth consumption by 25% over the baseline. Hence, our proposal not only significantly improves performance (as shown previously) but also significantly reduces off-chip bandwidth consumption, thereby improving bandwidth-efficiency.

Contrary to the very bandwidth-inefficient original CDP (which increases bandwidth consumption by 83%), ECDP increases bandwidth consumption by only 3.7% over the baseline. ECDP and coordinated throttling act synergistically: together, they increase bandwidth efficiency more than either of them alone. Using coordinated prefetcher throttling with ECDP results in the lowest bandwidth consumption. The largest bandwidth savings can be seen in `mcf`, `astar`, `xalancbmk`, `omnetpp`, `ammp`, `bisort`, and `pfast`. In these benchmarks, the throttling mechanism reduces the useless prefetches generated by the stream prefetcher because it has low accuracy and coverage. Throttling the inaccurate prefetcher reduces the pollution-induced misses, and hence unnecessary bandwidth consumption.

Summary: Table 6 summarizes the performance improvement and bandwidth reduction of our proposal, ECDP with coordinated prefetcher throttling. Our efficient LDS prefetching techniques improve performance by more than 5% on eleven benchmarks, while also reducing bandwidth consumption by more than 20% on eight benchmarks. Our mechanism eliminates all performance losses due to CDP.

6.1.3. Accuracy of Prefetchers Figure 8 shows that ECDP with prefetcher throttling (rightmost bars) improves CDP accuracy by 129% and stream prefetcher accuracy by 28% compared to when the stream prefetcher and original CDP are employed together. Our techniques increase the accuracy of CDP significantly on all benchmarks. Using both our techniques also increases the accuracy of the stream prefetcher on almost all benchmarks because it 1) reduces the interference caused by useless CDP prefetches, 2) reduces useless stream prefetches via throttling. `health` is an exception, where some misses that the stream prefetcher was covering (when running alone) are prefetched by ECDP in a more timely fashion, resulting in a decrease in stream prefetcher's accuracy. Increases in both prefetchers' accuracies results in the performance and bandwidth benefits shown in Sections 6.1.1 and 6.1.2.

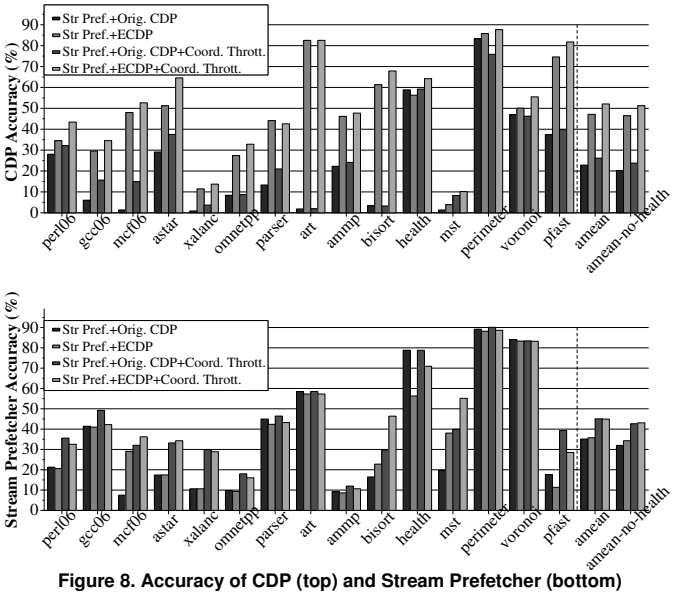


Figure 8. Accuracy of CDP (top) and Stream Prefetcher (bottom)

6.1.4. Coverage Of Prefetchers Figure 9 shows that ECDP with coordinated throttling slightly reduces the average coverage of both CDP and stream prefetchers. ECDP improves CDP coverage in several benchmarks (`art`, `health`, `perimeter`, and `pfast`) because it eliminates useless and polluting prefetches. In some others, it decreases coverage because it also eliminates some useful prefetches. Using coordinated prefetcher throttling also slightly reduces the average coverage of each prefetcher. This happens because each prefetcher can be throttled down due to low coverage/accuracy or because the other prefetcher performs better in some program phases. The loss in coverage is the price paid for the increase in accuracy. We conclude that our proposed mechanisms trade off a small reduction in CDP and stream prefetcher coverage to significant increases in CDP and stream prefetcher accuracy, resulting in large gains in overall system performance and bandwidth efficiency.

6.1.5. Effect of ECDP on Pointer Group Usefulness Figure 10 provides insight into the performance improvement of ECDP by showing the distribution of the *usefulness of pointer groups* with the original CDP and with ECDP. Recall that the usefulness of a pointer group is the fraction of useful prefetches generated by that pointer group (as described in Section 3). Using ECDP significantly increases the fraction of pointer groups that are useful. In the original CDP mechanism, only 27% of the pointer groups are very useful (75-100%

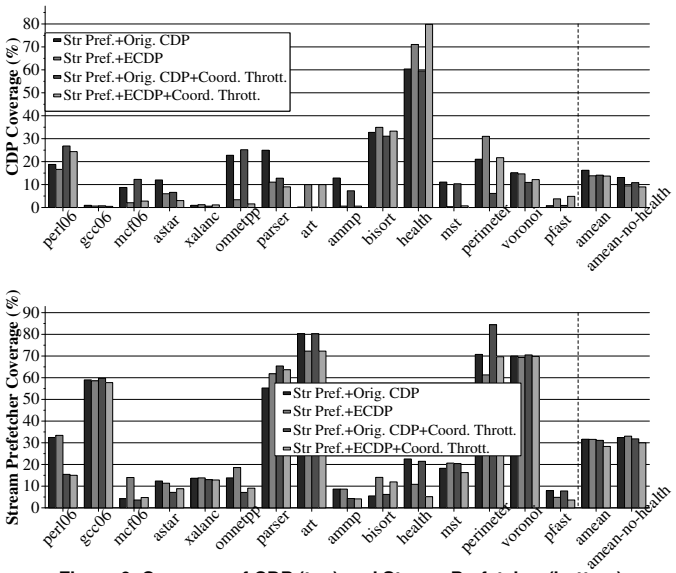


Figure 9. Coverage of CDP (top) and Stream Prefetcher (bottom)

useful) and 46% of the pointer groups are very useless (0-25% useful), With ECDP, 68.5% of all pointer groups become very useful (75-100% useful) whereas the fraction of very useless pointer groups drops to only 5.2%. Hence, ECDP significantly increases the usefulness of pointer groups, thereby increasing the performance and efficiency of content-directed LDS prefetching. Note that this is a direct result of the compiler discovering (via profiling) the beneficial pointer groups for each load to guide CDP.

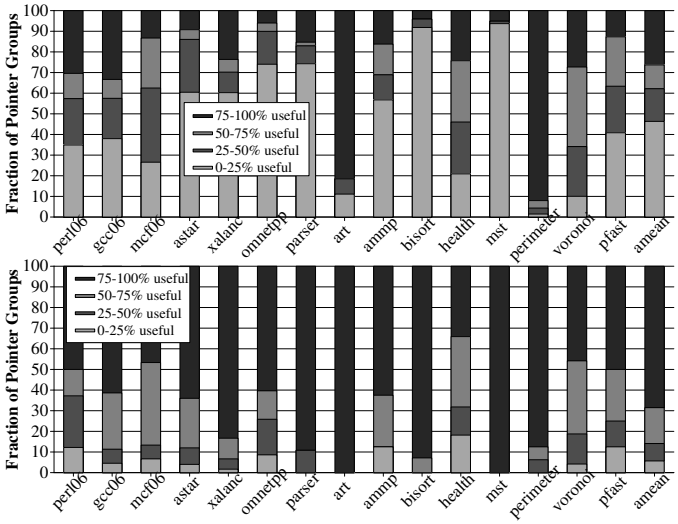


Figure 10. PG Usefulness: Original CDP Mechanism (top) ECDP (bottom)

6.1.6. Effect of Profiling Input Set The results we presented so far were obtained by profiling a different input set from the actual one used in experimental runs (as discussed in Section 5). To determine the sensitivity of ECDP to the profiling input set, we also profiled the applications with the same input set used for actual runs. We found that using the same input set for profiling as the actual input set improved our mechanism's performance by more than 1% only for one benchmark, `mst` (by 4%). Hence, our mechanism's benefits are insensitive to the input set used in the profiling phase.

6.2. Hardware Cost

Table 7 summarizes the storage cost required by our proposal. The storage overhead of our mechanism is very modest, 2.11 KB. Neither ECDP nor coordinated throttling requires any structures or logic that are on the critical path of execution. They require a small amount of combinational logic to 1) decide whether or not to prefetch a pointer based on the prefetch hints provided by a load instruction (in ECDP), 2) update the counters used to collect prefetcher accuracy and coverage for coordinated throttling, 3) update the *prefetched* bits in the cache. The major part of the storage cost of our mechanism is due to the *prefetched* bits in the cache. If these bits are already present in the baseline processor (e.g., for profiling or feedback-directed prefetching purposes), the storage cost of our proposal would be only 912 bits.

prefetched bits for each block in the L2 cache	8192 blocks × 2 bits/block
Counters used to estimate prefetcher coverage and accuracy (coordinated prefetcher throttling)	11 counters × 16 bits/counter
Storage for recording block offset and hint bit-vector for each MSHR entry	32 entries × (7 + 16 bits)/entry
Total hardware cost	17296 bits = 2.11 KB
Percentage area overhead (as fraction of the baseline 1MB L2 cache)	2.11KB/1024KB = **0.206%**

Table 7. Hardware cost of our mechanism (ECDP with coordinated throttling)

6.3. Comparison to LDS and Correlation Prefetchers

Figure 11 compares the performance and bandwidth consumption of our mechanism to those of a dependence based LDS prefetcher (DBP) [30], Markov prefetcher [17], and a global-history-buffer (GHB) based global delta correlation (G/DC) prefetcher [16]. Only the GHB prefetcher is not used in conjunction with the stream prefetcher because we found that GHB provides better performance when used alone as it can capture stream-based memory access patterns as well as correlation patterns. Previous research showed that the GHB prefetcher outperforms a large number of other prefetching mechanisms [28]. The DBP we model has a correlation table of 256 entries and a potential producer window of 128 entries, resulting in a ≈3 KB total hardware storage. The Markov prefetcher uses a 1MB correlation table where each entry contains 4 addresses. GHB uses a 1k-entry buffer and has 12KB hardware cost.[11] Our mechanism's cost is 2.11KB.

Results in Figure 11 show that our LDS prefetching proposal provides respectively 19%, 7.2%, 8.9% (12.7%, 7.1%, 5% w/o health) higher performance than DBP, Markov, and GHB prefetchers, while having significantly smaller hardware cost than Markov and GHB. Our technique consumes 22.7% and 29% (24% and 32% w/o health) less bandwidth than DBP and Markov prefetchers and 22% (19% w/o health) more bandwidth than GHB. We found that there are several major reasons our proposal performs better than these previous LDS/correlation prefetching approaches: 1) our approach is more likely to issue useful prefetches because the compiler provides information as to which addresses are pointers that are likely to be used, 2) our approach can prefetch pointer addresses that are not "correlated" with any previously seen address since it can prefetch any pointer value that resides in a fetched cache block, whereas Markov and GHB need to find correlation between addresses, 3) the Markov prefetcher cannot prefetch addresses that have not been observed and recorded previously, 4) the effectiveness of DBP is limited by the distance between pointer producing and consuming instructions, as shown by [30] and therefore DBP cannot prefetch far ahead enough to cover modern memory latencies [31], 5) our mechanism uses coordinated prefetcher throttling to control the interference between different prefetching techniques whereas none of the three mechanisms provide such a control mechanism.

Even though we provide a direct comparison to these LDS/correlation prefetchers, our mechanism is partly orthogonal to them. Both ECDP and coordinated prefetcher throttling can be used together with any of the three prefetchers when they are used in a hybrid prefetching system. For example, when ECDP is added to a baseline with GHB, the combination provides 4.6% performance improvement compared to GHB alone. Also, using coordinated throttling on top of a hybrid of GHB and ECDP provides a further 2% performance improvement and 6.5% bandwidth savings.

6.4. Comparison to Hardware Prefetch Filtering

Purely hardware-based mechanisms were proposed to reduce useless prefetches due to next sequential prefetching [41]. We compare our techniques to Zhuang and Lee's hardware filter [41], which disables prefetches to a memory address if the prefetch of that address

[11] The structures were sized such that each previous prefetcher provides the best performance.

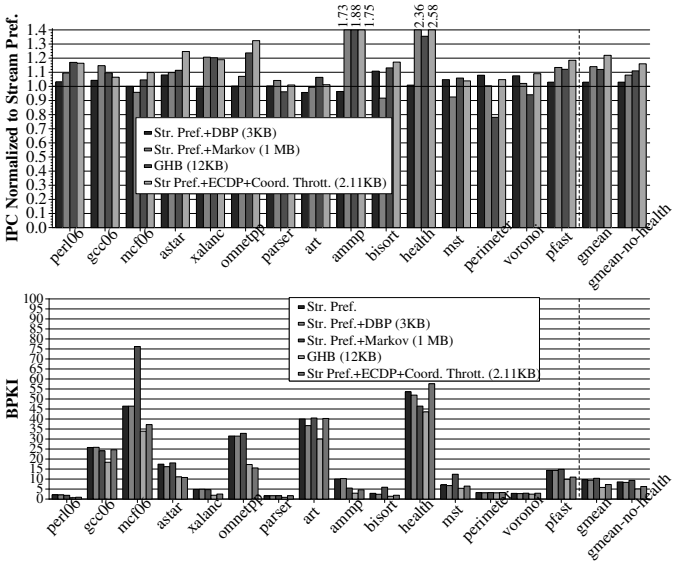


Figure 11. Comparison to other LDS/correlation prefetching techniques

was useless in the past. Figure 12 shows the effect of using a hardware filter with the original CDP (second bars from left) and in combination with coordinated throttling (third bars from left). We use an 8KB hardware filter, which provides the best performance in our benchmarks. The hardware filter by itself improves performance by only 4.4% (1.5% w/o health) and increases bandwidth consumption by 1.2% (2.6% w/o health). We found that the hardware filter is very aggressive and thus eliminates too many useful CDP prefetches. Using ECDP by itself is more effective than the hardware filter because ECDP is more selective in eliminating prefetches. Adding coordinated throttling on top of the hardware filter improves performance significantly, showing that the benefits of coordinated throttling are applicable to hardware filtering. However, using ECDP together with coordinated throttling provides better performance than using hardware filter and coordinated throttling. On average, our proposal (ECDP and coordinated throttling) provides 17% (14.2% w/o health) performance improvement and 25.8% (28.7% w/o health) bandwidth savings compared to simply using a hardware filter, which is more costly in terms of hardware, alone.

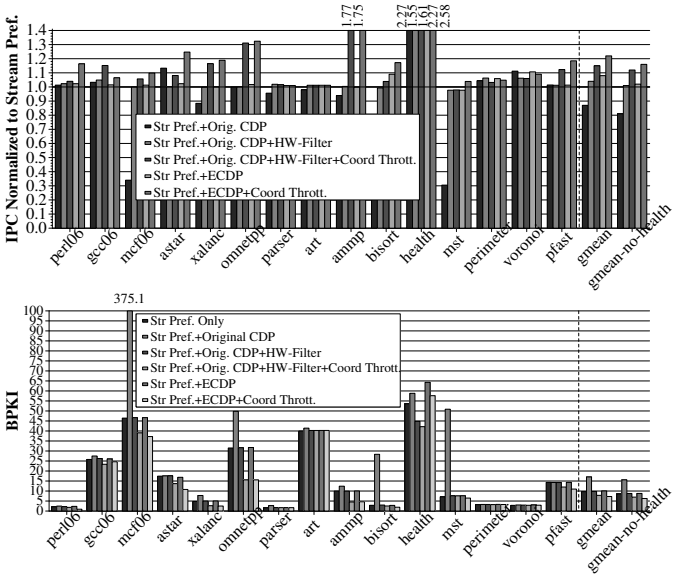


Figure 12. Performance and bandwidth comparison to HW prefetch filtering

6.5. Comparison to Feedback Directed Prefetching

Feedback directed prefetching (FDP) [36] incorporates dynamic feedback into the design of a single prefetcher to reduce the negative effects of prefetching. It was originally proposed for stream prefetchers. We compare the performance of coordinated prefetcher throttling in a hybrid prefetching system comprising a stream prefetcher and CDP. We implement and simulate FDP as explained in [36] and use it to change the aggressiveness of both the stream prefetcher and the bandwidth-efficient content-directed prefetcher individually. For these experiments, we set the cache block size to 64 bytes (and use the threshold values tuned in [36]), which we found to provide the best performance for FDP. Figure 13 compares coordinated throttling and FDP. Coordinated throttling outperforms FDP by 5% while consuming 11% more bandwidth on average. Coordinated throttling outperforms FDP due to two major reasons. First, throttling decisions made by our mechanism take into account the state of the other prefetcher(s), hence, the interaction between multiple prefetchers. In contrast, FDP does not coordinate the multiple prefetchers together; rather it throttles each of them individually. As a result, FDP cannot distinguish whether a prefetcher is performing well (or poorly) due to its own behavior or due to its interaction with other prefetchers. Second, our mechanism uses a smaller number of threshold values (three) than FDP, which requires six threshold values. Finding an effective combination of a smaller number of thresholds is easier. Therefore, our prefetcher throttling proposal is not only easier to tune but also easier to implement.

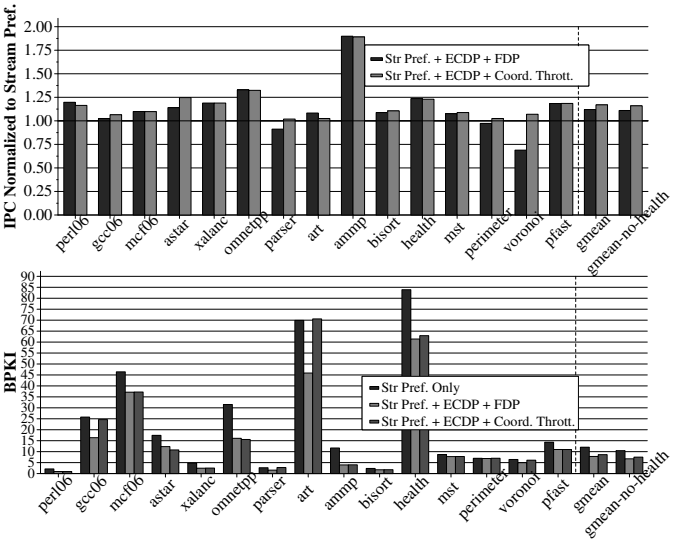
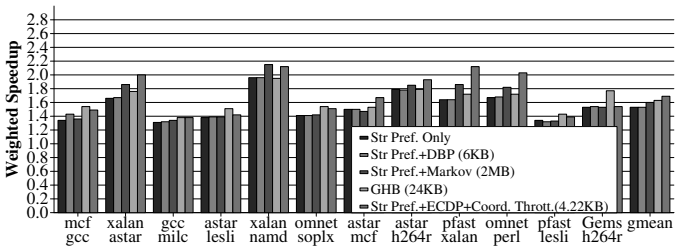


Figure 13. Prefetcher Throttling vs. Feedback Directed Prefetching

6.6. Effect on Multi-Core Systems

Dual-core System: Figure 14 shows the effect of combined ECDP and coordinated throttling on performance (weighted-speedup [33]) and bus traffic on a dual-core system. Our techniques improve weighted-speedup by 10.4%, hmean-speedup [25] by 9.9% (not shown), while reducing bus traffic by 14.9%. The highest performance gains are seen when two pointer-intensive benchmarks are run together. For example, when `xalancbmk` and `astar` run together, our mechanisms improve performance by 20% and reduces bus traffic by 28.3%. On the other hand, when both applications are pointer-non-intensive, the benefit of our mechanisms, as expected, is small (e.g., 1% performance improvement for `GemsFDTD` and `h264ref` combination). The results also show that our mechanism significantly outperforms DBP, Markov, and GHB prefetchers on the dual-core system. DBP is ineffective due to increased L2-miss latencies caused by each core's interfering requests. The Markov prefetcher (with a 1MB table per core) improves weighted/hmean-speedup by 4.1%/4.9% but increases bus traffic by 19.5%. GHB improves weighted/hmean-speedup by 6.2%/1% while reducing bus traffic by 5%.

4-Core System: Figure 15 shows that ECDP with coordinated throttling improves weighted/hmean-speedup by 9.5%/9.7% while reducing bus traffic by 15.3%. These benefits are significantly larger than those provided by Markov and GHB-based delta-correlation prefetchers that have higher hardware cost. We conclude that our low-cost and bandwidth-efficient LDS prefetching technique is effective in multi-core as well as single-core systems.



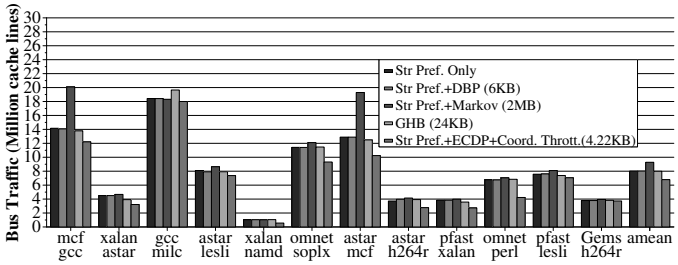
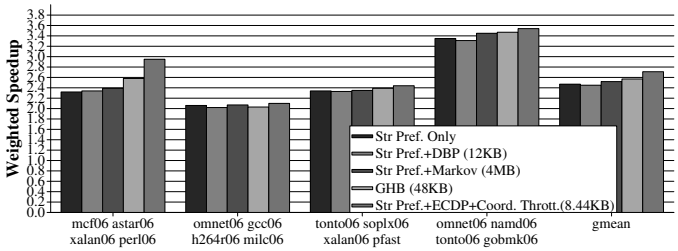


Figure 14. Effect of proposed mechanisms in a dual-core system



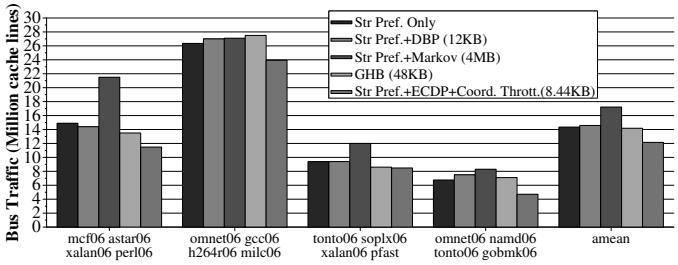


Figure 15. Effect of proposed mechanisms in a four-core system

6.7. Remaining SPEC and Olden Benchmarks

We evaluated our proposal on the remaining SPEC CPU2006/2000 and Olden benchmarks that have little LDS prefetching potential. We find that our combined proposal ECDP and coordinated throttling does not significantly affect the performance or bandwidth consumption of any remaining benchmark because these benchmarks do not have a significant number of cache misses caused by LDS traversals. On average, our mechanism improves performance by 0.3% and reduces bandwidth consumption by 0.1% on the remaining benchmarks.We conclude that our bandwidth-efficient CDP proposal does not degrade the performance of applications that are not memory- or pointer-intensive.

7. Related Work

To our knowledge, this paper provides the first comprehensive solution that enables both very-low-cost ($\approx$ 2KB extra storage) and bandwidth-efficient prefetching of linked data structures in a hybrid prefetching system. Our proposal has two new components: 1) a compiler-guided technique that determines which pointer addresses to prefetch in content-directed LDS prefetching, 2) a mechanism that throttles multiple different prefetchers (stream and LDS) in a coordinated fashion based on feedback information. The second component, coordinated prefetcher throttling, is orthogonal to LDS or any prefetching method employed in the system and can be used in conjunction with any hybrid prefetcher.

In previous sections, we already provided extensive quantitative comparisons to hardware prefetch filtering [41], three methods of LDS/correlation prefetching (dependence-based [30], Markov [17], global-history-buffer [16]), and feedback-directed prefetching [36]. Our evaluations showed that our proposal significantly outperforms these techniques, while requiring less hardware cost. Here, we briefly review and provide comparisons to other related work in content-directed prefetching, prefetch filtering, LDS prefetching, and multiple-prefetcher systems.

7.1. Related Work in Content Directed Prefetching

Guided Region Prefetching (GRP) [39] uses static compiler analysis to produce a set of load hints for its hardware prefetching engine, which includes the original CDP scheme [9]. GRP is a coarse-grained mechanism: it enables or disables prefetching for *all* pointers in cache blocks fetched by a load instruction. In contrast, our mechanism is fine-grained: it selectively enables/disables the prefetching of *useful/useless* pointers rather than *all* pointers related to a load instruction. We implemented GRP's coarse-grained control mechanism and found that, similarly to the results presented in [39], controlling CDP in a coarse-grained fashion provides negligible (0.4%) performance improvement.

Al-Sukhni et al. [2] propose a technique to statically identify values that are pointer addresses. Our work uses compile-time information to guide CDP in deciding *which* pointers to prefetch. Our proposal is orthogonal to theirs: static identification of pointers at compile time can be used in conjunction with our technique of deciding *which pointers to prefetch* to construct an even more accurate LDS prefetcher.

7.2. Related Work in Prefetch Filtering

Srinivasan et al. [37] use profiling to select which load instructions should initiate prefetches with a next sequential prefetcher and a shadow directory prefetcher. For CDP, we found that disabling prefetches on the basis of the triggering load results in the elimination of a very large number of useful prefetch requests and results in only 1% performance improvement because it is too coarse-grained an approach to eliminating content-directed prefetches.

7.3. Related Work in LDS Prefetching

Hardware-based approaches: Some hardware-based LDS prefetching approaches, such as correlation prefetching [5, 17, 20], pointer cache [7], spatial memory streaming [35], and hardware jump pointer prefetching [31] require large storage overhead to maintain pointer or correlation values in hardware. Specifically, correlation prefetching requires at least 1-2MB tables [5, 17, 20], the pointer cache requires 1.1MB of storage [7], spatial memory streaming [35] and hardware jump pointer prefetching [31] each require at least 64KB of storage. In contrast, our mechanism requires only 2.11KB storage since it does not require storing any pointer or correlation values. In addition, most correlation-based prefetchers are only capable of prefetching addresses that have been observed and recorded previously. Our technique can prefetch addresses that have not previously been used by the program.

Hu et al. [15] propose a correlation prefetcher with smaller storage requirements. This prefetcher can record only those correlations that are in the same cache set. Unlike our mechanism, it cannot capture across-set address correlations in LDS accesses.

Mutlu et al. [26] propose address-value delta prediction to predict pointer addresses loaded by *pointer load* instructions. AVD prediction is less effective when employed for prefetching instead of value prediction [26].

Pre-execution-based approaches: Pre-execution-based LDS prefetching techniques [6, 4, 43, 23, 8, 34, 40] use idle thread contexts or separate pre-execution hardware to run "threads" that help the primary program thread. Such helper threads, constructed either by the compiler [6, 43, 23] or the hardware [40, 8, 4], execute code that prefetches for the primary thread. These techniques require either separate, idle thread contexts and spare resources (e.g., fetch and execution bandwidth), which are scarce when the processor is well used, or specialized engines/hardware.

Software-based approaches: Software-based LDS prefetching techniques (e.g. [22, 24, 31, 1]) require the programmer or the compiler to analyze program objects, determine objects that lead to a ma-

jority of the cache misses via profiling, and insert prefetch instructions sufficiently ahead of a pointer access to hide memory latency. Most of these approaches [22, 24, 31], while shown to be beneficial in small benchmarks using hand-optimized code, usually require significant programmer support to generate timely LDS prefetch requests, as described in [24, 31]. Software techniques that do not require programmer support, e.g. [1], are limited to managed runtime systems with dynamic profile feedback and are not generally applicable to C/C++ and other non-managed languages.

7.4. Related Work in Multiple-Prefetcher Systems

Gendler et al. [11] propose turning off (not throttling) *all prefetchers* but the most accurate one based on only per-prefetcher accuracy data obtained from the last N prefetched addresses. Unlike our coordinated prefetcher throttling technique, this simplistic mechanism 1) does not take into account prefetch coverage, 2) can disable a very accurate, high-coverage, non-interfering prefetcher that is improving performance while enabling a very low-coverage yet more accurate prefetcher that does not help performance, 3) cannot capture the interaction between prefetchers for different access patterns because it does not *throttle* them in a coordinated fashion. We implemented this scheme and found that it reduces average performance by 11% while decreasing bandwidth consumption by 6.7% on our benchmarks.

8. Conclusion

We proposed a very-low-cost and bandwidth-efficient hardware/software cooperative prefetching solution for linked data structures. Our solution comprises two new techniques. First, a compiler-guided prefetch hint mechanism that enables efficient content-directed LDS prefetching. Second, a technique to manage the interference between multiple prefetchers (streaming and LDS) in a hybrid prefetching system. We showed that our proposal significantly improves performance and reduces memory bandwidth consumption on both single-core and multi-core systems compared to three other LDS/correlation prefetchers on a set of pointer-intensive applications. We conclude that our techniques enable low-cost and efficient prefetching of linked data structures in hybrid prefetching systems.

Acknowledgments

Many thanks to Chang Joo Lee, Veynu Narasiman, other HPS members and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Microsoft Research, and Intel Corporation. Part of this work was done while Onur Mutlu was a researcher and Eiman Ebrahimi was a research intern at Microsoft Research.

References

[1] A.-R. Adl-Tabatabai et al. Prefetch injection based on hardware monitoring and object metadata. In *PLDI*, 2004.

[2] H. Al-Sukhni, I. Bratt, and D. A. Connors. Compiler directed content-aware prefetching for dynamic data structures. In *PACT-12*, 2003.

[3] C. Alkan et al. Structural variation detection using high-throughput sequencing. In *Pacific Symposium on Biocomputing*, 2008.

[4] M. Annavaram et al. Data prefetching by dependence graph precomputation. In *ISCA-29*, 2001.

[5] M. J. Charney and A. P. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell Univ., 1995.

[6] J. D. Collins et al. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA-28*, 2001.

[7] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *MICRO-35*, 2002.

[8] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *MICRO-34*, 2001.

[9] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X*, 2002.

[10] J. Doweck. *Inside Intel Core Microarchitecture and Smart Memory Access – White Paper*. Intel, Jul 2006.

[11] A. Gendler et al. A pab-based multi-prefetcher mechanism. *International Journal of Parallel Programming*, 34(2):171–478, Apr. 2006.

[12] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

[13] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001. Q1 2001 Issue.

[14] M. Horowitz et al. Informing memory operations: providing memory performance feedback in modern processors. In *ISCA-23*, 1996.

[15] Z. Hu, M. Martonosi, and S. Kaxiras. TCP: Tag Correlating Prefetchers. In *HPCA-8*, 2002.

[16] K. J.Nesbit and J. E.Smith. Data cache prefetching using a global history buffer. In *HPCA-10*, 2004.

[17] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA-24*, 1997.

[18] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.

[19] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Comp Arch Letters*, 2002.

[20] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *ISCA-28*, 2001.

[21] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *ACM Communications*, 26, June 1983.

[22] M. H. Lipasti et al. SPAID: Software prefetching in pointer- and call-intensive environments. In *MICRO-28*, 1995.

[23] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA*, 2001.

[24] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-7*, 1996.

[25] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.

[26] O. Mutlu et al. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploting regular memory allocation patterns. In *MICRO-38*, 2005.

[27] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA-21*, 1994.

[28] D. G. Perez et al. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO-37*, 2004.

[29] A. Rogers et al. Supporting dynamic data structures on distributed memory machines. *ACM TOPLAS*, 17(2), Mar. 1995.

[30] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-8*, 1998.

[31] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA-26*, 1999.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.

[33] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.

[34] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA-29*, 2002.

[35] S. Somogyi et al. Spatial memory streaming. In *ISCA-33*, 2006.

[36] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[37] V. Srinivasan et al. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan, 1999.

[38] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[39] Z. Wang et al. Guided region prefetching: a cooperative hardware/software approach. In *ISCA-30*, 2003.

[40] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *ICS-2000*, 2000.

[41] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP-32*, 2003.

[42] C. Zilles. Benchmark health considered harmful. *Computer Architecture News*, 29(3), 2001.

[43] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA-28*, 2001.