# Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors

Wim Heirman
Intel Corporation
wim.heirman@intel.com

Kristof Du Bois
Intel Corporation
kristof.du.bois@intel.com

Yves Vandriessche
Intel Corporation
yves.vandriessche@intel.com

Stijn Eyerman
Intel Corporation
stijn.eyerman@intel.com

Ibrahim Hur
Intel Corporation
ibrahim.hur@intel.com

## ABSTRACT

In modern processors, prefetching is an essential component for hiding long-latency memory accesses. However, prefetching too aggressively can easily degrade performance by evicting useful data from cache, or by saturating precious memory bandwidth. Tuning the prefetcher's activity is thus an important problem. Existing techniques tend to focus on detecting negative symptoms of aggressive prefetching, such as unused prefetches being evicted or memory bandwidth saturation, and throttle the prefetcher in response.

We argue that these *far-side* throttling techniques are inefficient because they require significant tracking state, and are reactive to negative effects rather than being proactive. We propose an alternative technique which we coin *near-side* throttling, which works by detecting *late* prefetches and tuning the prefetch distance to closely track the point at which most prefetches are not late. Because late prefetches are by definition useful, detecting late prefetches exclusively suffices to detect and prevent useless prefetches as well. Our solution is cheap to implement in hardware, includes throttling on off-chip bandwidth saturation, applies to both hardware and software prefetching, and can control multiple concurrent prefetchers where it will naturally allow the most useful prefetch algorithm to generate most of the requests. Through detailed simulation of a many-core architecture running a wide range of sequential and parallel applications, we show that our near-side throttling (NST) proposal performs similar to the state-of-the-art feedback-directed prefetching (FDP), even though it has a significantly lower implementation cost, can react more quickly to changes in application behavior and is applicable to a more varied set of use cases.

**ACM Reference Format:**
Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-Side Prefetch Throttling: Adaptive Prefetching, for High-Performance Many-Core Processors. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3243176.3243181


*PACT '18, November 1–4, 2018, Limassol, Cyprus*

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243181

## 1 INTRODUCTION

In modern computer systems, both memory latency and off-chip bandwidth have struggled to keep up with compute performance. While caches can help exploit spatial and temporal locality, many workloads have large data sets that do not fit in cache, and instead have to rely on hardware and/or software prefetching to hide main memory latency. This problem becomes especially important on many-core high-performance processors, where cache sizes are usually limited (in favor of core count) and memory latency is higher because of a more complex network fabric and competition for off-chip bandwidth. Significant research has gone into determining *what* to prefetch, resulting in several algorithms that detect memory access patterns and predict future memory references, so they can be prefetched ahead of time to hide (most of) the main memory access latency from the application. Popular examples include stream prefetchers [6, 12, 15], the global history buffer [23], indirect prefetching [32], and many others [13, 16, 17, 22].

In addition to generating the addresses to prefetch, an important question is *when* to prefetch them. Typically, when looking at a memory access stream, the application has reached one point on the stream and the prefetcher is generating requests at some future point, ahead of the application by a given distance. This prefetch distance must be high enough such that prefetch requests are *timely*, i.e., they are made sufficiently ahead of time such that the request has completed by the time the application makes its demand request for that cache block. If not, the prefetch is considered *late*. However, prefetching too far ahead can be problematic, for two main reasons. First, the prefetcher's prediction may be wrong. Typically the error increases for predictions made further out, the prefetch distance should therefore be limited to the range where prediction accuracy is good. Wrong prefetches waste both off-chip memory bandwidth, and displace useful data from the caches. Second, if a cache block (even when correct) is prefetched too far ahead, it will be displaced from the cache, by demand misses or by more prefetches, before the application has a chance to use it (*useless* prefetch).

Different applications can have contrasting behavior when varying the prefetch distance. Figure 1 plots application performance, relative to a baseline that has no L2 prefetching, for a selection of applications and prefetcher distances. (See Section 3 for more methodological details.) Some applications, such as `HPCG`, benefit from prefetching and perform best with an aggressive prefetcher. For others, such as `cactus`, the prefetcher is not able to bring in useful data and instead saturates memory bandwidth with useless prefetches. Finally, some applications such as `mcf` benefit from a

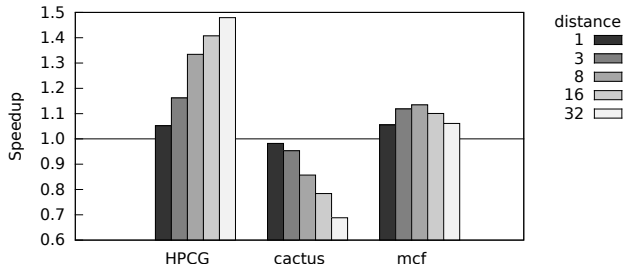


**Figure 1: Impact of varying prefetch distance on application performance.**

limited prefetch distance but see reduced performance beyond a certain distance. These differences between applications, and inside applications that exhibit phase behavior, make it difficult to tune a prefetcher with a single set of parameters. In addition, the optimal distance can depend on data locality (whether the data resides in a last-level cache, or in fast on-package versus slow off-package memory) and on the bandwidth available to each core in a many-core environment (e.g., when not all cores are active, memory bandwidth pressure is lower so there is a higher tolerance for useless prefetches). A dynamic approach is therefore needed, which should be able to detect the prefetcher's usefulness at runtime, and reign in the prefetcher or let it run ahead as appropriate for the current application or phase.

Existing solutions tend to focus on detecting the effect of wrong prefetching directly. A common approach, followed by feedback-directed prefetching (FDP) [29] and others, instruments the cache and adds a bit to each cache block to indicate it was brought in by prefetching, and clears the bit whenever the application uses that cache block. If any block is evicted with the bit still set, a useless prefetch is detected, which serves as input to algorithms to throttle the prefetcher. The same can be done when memory bandwidth is saturated. We call these techniques *far-side* throttling, as they keep the prefetch distance generally at a high level, and reduce it only in reaction to negative prefetching effects.

We propose an alternative way of looking at prefetch throttling, which we coin *near-side* throttling, that keeps the prefetch distance as low as possible while still retaining all of the prefetcher's benefits. This tends to operate the prefetcher in its most accurate range, and minimizes cache pollution and memory bandwidth wastage. We argue that near-side throttling makes more efficient use of bandwidth, which on today's many-core processors is becoming more and more scarse. Near-side throttling is implemented by matching demand accesses with outstanding prefetch requests, and hence detecting late prefetches. A control loop then adapts the prefetch distance to a level where the amount of late prefetches is still detectable, but harmlessly low. This approach has several advantages. First, hardware implementation is cheap: outstanding prefetches can be found in the miss status holding register (MSHR) which is an existing structure, and demand accesses are already matched to it to avoid duplicate memory accesses. We add just one bit per MSHR entry to tag prefetches; no additional tracking state is needed in the caches. Second, this technique can respond to application phase behavior much more quickly: late prefetches are detected after at most a time equal to the memory access latency (less than one microsecond), whereas useless prefetches need to be evicted before they can be detected as such (which can take up to several milliseconds for large caches). Finally, near-side throttling detects bandwidth saturation locally (through memory latency), so no global coordination is needed between the per-core prefetchers in a many-core architecture, or even between multiple prefetch algorithms on a single core.

In this paper we make the following contributions:

- We introduce the concept of near-side prefetch throttling (NST) and discuss its implementation. We estimate hardware storage overhead and show that it is considerably cheaper to implement than traditional far-side throttling, exemplified by feedback-directed prefetching (FDP) as a the state-of-the-art technique.
- We evaluate near-side throttling for a wide range of applications and show that it can find the optimal prefetch distance, both per workload and in each application phase. NST even slightly outperforms FDP (9.6% vs. 9.4% speedup over no prefetching, respectively).
- We study the sensitivity of our implementation and show stable behavior over a wide range of parameters, negating the need for machine-specific tuning.
- We show that near-side throttling can be extended to multiple prefetchers per core, where it will naturally throttle those prefetchers that yield no useful requests, allowing for a diverse set of prefetch algorithms to co-exist.
- We apply near-side throttling to find the optimal distance when doing software prefetching, eliminating the need for tuning and achieving performance portability.

## 2 NEAR-SIDE PREFETCH THROTTLING

The aim of prefetch throttling is to allow the prefetcher to run ahead far enough such that prefetches are timely, but prevent it from running too far ahead into a region where the prefetch algorithm can no longer accurately predict the application's access pattern which leads to useless prefetches. An additional concern is to prevent prefetches from hogging all available off-chip memory bandwidth which slows down demand misses.

The basic concept of near-side prefetch throttling (NST) is to detect late prefetches, and tune the prefetcher aggressiveness such that the amount of late prefetches is balanced around a small but non-zero fraction of all prefetches. Because late prefetches — which occur when an application (demand) access hits on an outstanding prefetch request — are by definition useful, reigning in the prefetcher to a point where the prefetches are (almost) just-in-time automatically prevents overly aggressive prefetching which is more likely to generate useless prefetches. Maintaining a small fraction (e.g., 10%) of late prefetches does not harm performance as long as the late prefetches are only late by a small amount, i.e., the demand access is made only just before the prefetch request completes. In practice, this means that even late prefetches hide most (all but a few clock cycles) of a long-latency off-chip memory access (totaling hundreds of cycles).

| Symbol | Description | |
|---|---|---|
| | *MSHR bits (per entry and per prefetcher)* | |
| $p_x$ | Request was launched by prefetcher $x$ | |
| | *Counters (per prefetcher)* | |
| $a_x$ | All prefetches | |
| $l_x$ | Late prefetches | |
| | *Configuration parameters* | *Value* |
| $W_{pos}$ | Window size while increasing distance | [10 µs] |
| $W_{neg}$ | Window size while decreasing distance | [5 µs] |
| $F_{max}$ | Maximum fraction of late prefetches | [10%] |
| $H_{max}$ | Hold time (in windows) until decrease | [10] |
| $R_{min}$ | Minimum prefetch rate | [1] |
| $R_{max}$ | Maximum prefetch rate | [8] |
| | *Algorithm variables (per prefetcher)* | |
| $R_x$ | Current prefetch rate | |
| $W_x$ | Remainder of current window | |
| $H_x$ | Elapsed hold time | |
| $F_x$ | Late prefetch fraction | |
| | *Machine parameters* | |
| $L$ | Uncontended main memory latency (cycles) | |

**Table 1: Notations used in Section 2.**

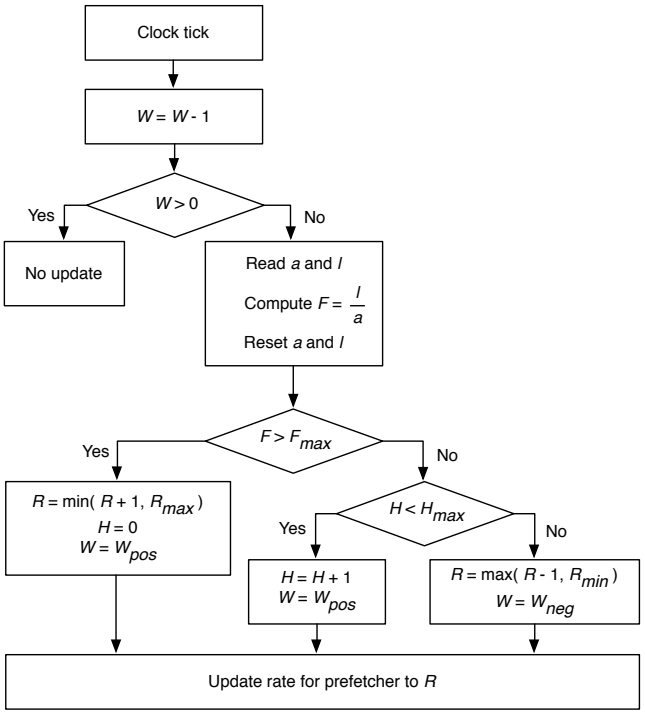


**Figure 2: Flowchart of the prefetcher control algorithm.**

Tuning the late prefetch fraction around a fixed point also works when the application changes its behavior. When the application's memory requests come closer together, the late prefetch fraction will increase and NST will increase the prefetch distance. When the distance is larger than needed no late prefetches will be detected and NST reigns in the prefetcher to ensure the late prefetch fraction remains at its operating point. In more complex scenarios where there are both a high fraction of late and useless prefetches, NST will extend the prefetch distance to reduce late prefetches as long as off-chip bandwidth is not saturated.

Our proposed implementation of near-side prefetch throttling consists of some extra state in the processor's miss status holding register (MSHR) and a state machine. The MSHR tracks outstanding cache misses triggered by both demand requests (application loads and stores), and prefetches; the extra state allows for detection of late prefetches. The state machine periodically computes the fraction of late prefetches, and updates the optimal prefetch distance.

## 2.1 Detecting late prefetches

Late prefetches occur when the core accesses a cache block on which a prefetch request is still outstanding. These prefetch requests are by definition useful (the application does access the same address), but should have been issued earlier to allow the prefetch to hide the full memory latency from the application.

We extend the MSHR with one extra bit per entry (the $p$ bits, see Table 1 for a summary of notations used throughout this section). Two counters are kept: one for counting all prefetches issued ($a$), and another one for counting late prefetches ($l$). Each time a cache miss occurs and an MSHR entry is allocated, the entry's $p$ bit is initialized depending on whether the request was a prefetch ($p$ bit is set) or another type of request (bit is cleared). When $p$ is set, the $a$ counter is incremented.

The $l$ counter is incremented as part of the existing cache miss handling logic. An application load or store that misses the cache triggers a check of the MSHR to filter out duplicate requests to the same cache block. When there is an MSHR match, we now check whether the matching entry's $p$ bit is set. If so, the $l$ counter is incremented and the $p$ bit is cleared.

## 2.2 Determining prefetch rate

The prefetch rate is updated every $W$ clock cycles, based on the values of the $a$ and $l$ counters using the algorithm in Figure 2. First, the fraction of late prefetches $F = l/a$ is computed and both counters are reset. As long as the fraction is larger than the threshold $F_{max}$, the current prefetch rate $R$ is increased (up to $R_{max}$). If there is no significant amount of late prefetches, $R$ is held constant for $H_{max}$ update windows. After that, $R$ starts to decrement down to $R_{min}$, or until the late prefetch fraction again exceeds $F_{max}$. It is possible to decrement $R$ faster than it is incremented by changing the ratio of $W_{pos}$ to $W_{neg}$. The algorithm keeps $R$ just above its minimum required value, this ensures there are only few late prefetches, but also avoids excessive or early prefetching.

Figure 3 illustrates the behavior of the algorithm through time. The instantaneous optimal prefetch distance is shown as a dotted line, and varies through time depending on memory latency, access pattern, etc. Once the current prefetch rate ($R$, solid line) is smaller than the optimal prefetch distance, late prefetches start to occur

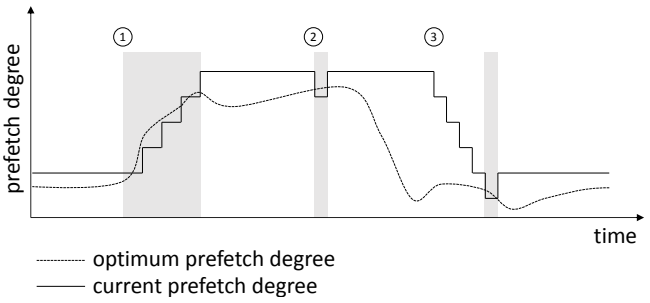


**Figure 3: Illustration of the behavior through time.**

(gray areas). At time ①, the memory access pattern shifts and a larger prefetch distance is needed. At this point, late prefetches occur and the algorithm will increase $R$ as long as the late prefetches persist. After $R$ was raised high enough, it then periodically (every $H_{max}$ windows) tries to lower $R$ to see if conditions have changed. The first time this is attempted is at time ②, where conditions did not allow for a lower prefetch distance. Here, $R$ reverts to its original value in the next window. By time ③ the current optimal prefetch distance did reduce so $R$ is decremented, and will eventually stabilize at just above the optimal distance.

## 2.3 Detecting bandwidth saturation

When the application is bandwidth bound, the processor's off-chip memory channels will saturate and memory latency (through queueing delays) increases. In this situation, issueing more prefetches or launching them earlier will not help as this only increases queueing delays; moreover it increases the probability of generating wrong prefetches which will aggravate the bandwidth pressure. From the point of view of the MSHR, memory accesses that have been in the MSHR for significantly longer than the normal access latency are indicative of queueing delay, and hence should not trigger NST to increase the prefetch rate even if they result in late prefetch hits.

To throttle the prefetcher when off-chip bandwidth is saturated, we simply clear all $p$ bits periodically, using an interval that is slightly longer than the uncontended memory access latency ($L$). We do not explicitly measure off-chip bandwidth consumption, nor is there any need to coordinate bandwidth measurements or prefetch throttling between cores or other agents on the same processor chip — the fact that memory latency increased beyond a threshold value (we used $2 \cdot L$ in our experiments) is a simple, localized indication that bandwidth saturation has occurred.

By clearing all $p$ bits periodically (every $2L$ clock cycles), requests with a latency longer than $2L$, which are affected by queueing delays, can no longer count as late prefetch hits. This will reduce the late prefetch fraction, and therefore automatically reduce the prefetch distance, in the face of bandwidth saturation. In uncontended scenarios, it is possible for $p$ bits to be cleared unnecessarily, depending on when the prefetch request is launched in relation to the edge of the clearing interval. An uncontended request with latency $L$ has a 50% chance of having its $p$ bit cleared. However this is not a problem as we can simply take this into account when selecting the $F_{max}$ parameter. Moreover, our technique does not require a direct measurement of the latency of individual requests, saving the associated counters and timing circuits this measurement would entail.

| Prefetcher | $R_x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Stream [5] | distance | 1 | 2 | 4 | 8 | 12 | 16 | 24 | 32 |
| GHB [23] | width:depth | 1:1 | 2:1 | 2:2 | 3:2 | 4:3 | 4:4 | 6:4 | 6:6 |
| ISB [13] | distance | 1 | 2 | 3 | 4 | 5 | 6 | | |

**Table 2: Potential mapping between prefetch rate and prefetcher-specific settings.**

## 2.4 Multiple prefetchers

In a many-core processor, the prefetchers in each core can be controlled independently based on their own specific late prefetch fraction. This allows for heterogenous applications or multi-programming workloads, and will tune each prefetcher's distance to the specific access pattern it is experiencing. In addition, we can extend near-prefetch throttling to support multiple hardware prefetchers per core, or to simultaneously control both a software prefetch distance and one or more hardware prefetchers.

To support multiple prefetchers, each MSHR entry contains multiple $p$ bits, one per prefetcher (denoted as $p_x$ bits, where $x$ denotes either software prefetches or one of the hardware prefetchers). The *all* and *late* prefetch counters are also duplicated per prefetcher ($a_x$ and $l_x$). When there is an outstanding prefetch request for a given address (MSHR entry has $p_x$ set), and another prefetcher ($y$) predicts the same address, this request is matched in the MSHR and we set the entry's $p_y$ as well and increment $a_y$. If the address proves useful (when an application request hits this MSHR entry), multiple $p_x$ bits can be set and each corresponding prefetcher's $l_x$ counter is incremented, after which all of the entry's $p$ bits are cleared. This ensures that, when multiple prefetchers predict the same block, both will get credit if the block proves useful.

Finally, the rate for each prefetcher is determined independently, based on the values of its $a_x$ and $l_x$ counters using the algorithm in Figure 2. This way the most useful prefetch algorithm is allowed to make most of the prefetch requests, while algorithms that do not generate useful prefetches are throttled. This makes it possible to combine different prefetch algorithms that each perform well on some applications, but minimize the activity of those prefetchers that are not applicable to the current application or phase.

## 2.5 Computing prefetch settings

While the variable $R_x$ provides a per-prefetcher indication of its desired prefetch rate, different prefetch algorithms have different ways of controlling them. Therefore the $R_x$ value is converted using a mapping function that depends on the type of prefetcher. For a stream prefetcher, the rate can be controlled by setting the prefetch distance, this is the number of consecutive blocks to fetch for each stream. The mapping can be made non-linear, so the prefetch distance goes up slowly for small values of $R_x$ while moving more quickly through larger distances. More complex prefetch algorithms can have multiple parameters that are all controlled by the $R_x$

| Component | FDP [29] | Near-side throttling |
|---|---|---|
| *Detecting late prefetches* | | |
| 1 bit per L2 MSHR (32 MSHRs) | 4 B | 4 B |
| *Detecting useless prefetches* | | |
| 1 bit per L2 cache block (L2 capacity = 1024 KB) | 2048 B | — |
| *Additional counters* | | |
| All prefetches | 4 B | 4 B |
| Late prefetches | 4 B | 4 B |
| Useless prefetches | 4 B | — |
| Demand misses | 4 B | — |
| Demand misses caused by prefetcher | 4 B | — |
| *Internal state algorithm* | 12 B | 16 B |
| **Total** | 2084 B | 28 B |

**Table 3: Storage overhead per prefetcher.**

| Benchmark | Input size | Cores (OpenMP×MPI) |
|---|---|---|
| *SPEC CPU2017* | | |
| * | ref | 1 |
| *APEX* | | |
| GTCP | A.txt | 8 (4×2) |
| HPCG | 64×64×64 | 8 (4×2) |
| Meraculous | bwa-25pct.list.ufx.bin | 2 |
| MILC | 6×6×18×6 | 8 (4×2) |
| MiniDFT | small.in | 1 |
| MiniPIC | 10×10×10 | 8 (4×2) |
| PENNANT | leblancbig | 8 (4×2) |
| SNAP | in_s | 8 (8×1) |
| UMT | grid2MPI_3x6x8.cmg | 8 (4×2) |
| *Caffe* | | |
| Intel Caffe | AlexNet | 72 |

**Table 4: Workload details.**

variable. We used the mapping from Table 2 for the experiments described later in this paper.

Note that a given prefetcher should never be completely disabled: near-side throttling needs to see an amount of late prefetches before it can decide to increase the activity for that prefetcher, so if a prefetcher is turned off entirely the algorithm will get stuck in $R = 0$. We therefore set $R_{min}$ to one rather than zero. It is possible to mostly disable a prefetcher, by implementing the lowest rate to generate only a small fraction of the usual prefetch rate. This reduces the amount of useless prefetches to a harmlessly low value while still allowing for the late prefetch fraction to be measured. This approach can be used on for instance an indirect prefetcher [32], which can be very useful on some applications but will not work at all if it is not operating on pointer data.

## 2.6 Hardware overhead

Table 3 provides an overview of the total storage overhead of near-side prefetch throttling per prefetcher (see Section 3 and Table 5 for more details about our experimental setup). We compare our proposal to Feedback Directed Prefetching (FDP) [29], which is the work mostly related to ours.

Near-side prefetch throttling requires one bit per MSHR entry. For a 32-entry MSHR, this adds up to a total of just 4 bytes. Also, we need two additional registers to count the number of total and late prefetches. As these counters are reset at the end of a time window (which has a length in the order of 10 µs, see Figure 2), a 32-bit wide register is enough to prevent overflow. The internal state of our algorithm consists of four variables, which are also 32 bit wide, resulting in a storage overhead of 16 B. The total storage is limited to 28 B per prefetcher, and scales linearly with the number of prefetchers.

FDP requires one bit per L2 cache block to keep track of blocks brought in by the prefetcher. For an L2 cache with a capacity of 1024 KB, this leads to a non-negligible storage overhead of 2048 B. Moreover, FDP detects late prefetches with a similar approach to ours and needs five additional counters. The state of the algorithm is limited to three 32-bit variables. In total, the storage overhead of FDP is 2084 B. This overhead scales linearly with the amount of prefetchers, and with cache capacity.

# 3 EXPERIMENTAL SETUP

We evaluate near-side prefetch throttling on the SPEC CPU2017 [3] suite as well the APEX benchmarks [1] (a selection of HPC workloads) and Intel Caffe [2] (a machine learning workload). Table 4 provides more details on these workloads, the input sets, and the number of OpenMP threads and MPI ranks used for each workload.

The SPEC CPU workloads execute up to several trillion instructions each, so we apply a sampling strategy where we select ten one-billion instruction slices, spaced evenly throughout the run for each workload and input combination. We use the *ref* inputs, and simulate all input sets. When reporting speedup, we aggregate these multiple inputs per application by summing their runtimes. The APEX benchmarks consist of a set of hybrid MPI+OpenMP real-world HPC applications used by the NERSC supercomputing center. We use a shared-memory MPI implementation, running on a single node. We use the small problem sizes, which is intended for node level performance analysis, and simulate only the parallel part of the application. For some applications, even the small problem size is too large to simulate in a reasonable amount of time, therefore we further reduced problem sizes or created a kernel that is representative for the execution of the application (this is the case for `MiniDFT` and `Meraculous`). Intel Caffe is a deep learning framework [26], we configure AlexNet [18] as neural networks, and use the ImageNet [8] dataset. We simulate two iterations of inference, the first iteration to warm up caches and branch prediction structures, and report on the second iteration.

We use an internal many-core simulator derived from Sniper [7] to model a high-performance many-core architecture, inspired by

| Component | Parameters |
|---|---|
| Core | 72× 1.5 GHz, 2-way OOO, 72-entry ROB |
| Branch predictor | hybrid local/global predictor |
| L1-I | 32 KB, 4 way, 1 cycle access time |
| L1-D | 32 KB, 8 way, 1 cycle access time |
| L1-D prefetcher | IP-indexed, 256 entries |
| L2 cache | 1024 KB shared per tile, 16 way, 18 cycle |
| L2 prefetcher | • stride-based, 32 streams, degree 3<br>• global history buffer, 1024 entries<br>• irregular stream buffer |
| Coherence protocol | directory-based MESIF, distributed tags |
| On-chip network | 6×6 mesh, 2 cores per tile, 1 cycle per hop<br>32 GB/s per link per direction |
| Memory controller | FR-FCFS scheduling, line-interleaved mapping,<br>closed-page policy |
| On-package memory | 16 GB MCDRAM, 8 channels, 400 GB/s total |

**Table 5: Simulated architecture details.**

Intel's Xeon Phi 7290 (Knights Landing) processor [28], see Table 5 for details. To keep simulation times feasible, we simulate either one, eight, or all 72 processor cores, depending on the workload as indicated in Table 4. For simulations that do not use the full chip, we rescale memory bandwidth, mesh topology, and (for single-core runs) L2 cache capacity. This way, we ensure the per-core bandwidth and cache capacity correspond to their full-chip equivalent. This makes our results representative for running SPEC_rate across the full chip, or filling the chip with more MPI ranks (assuming a weak scaling problem) for parallel workloads.

Our baseline results have only the L1-I and L1-D prefetchers enabled. We then enable one or more L2 prefetchers, and for each workload plot performance relative to this baseline. Unless mentioned otherwise, we use the L2 stride prefetcher and the default values from Table 1 to configure the near-side throttling mechanism. The stride prefetcher has a degree of three, meaning that it will launch three new prefetch requests on every hit — until the prefetcher has run out to its maximum configured distance away from the application's last access at which point it will revert to making a single new request per hit. The distance is set statically or through a throttling mechanism and is expressed as a multiplier for the stride of a stream. Our simulator tags prefetched blocks in the cache so we can report on useless prefetching, although this information is not used by the near-side throttling algorithm. For comparison, we also implemented feedback-directed prefetching (FDP) using the algorithm and default configuration values from [29].

# 4 RESULTS

## 4.1 Performance

We begin by evaluating the speedup obtained by prefetching, computed as the baseline runtime (no L2 prefetcher) divided by runtime with the L2 stride prefetcher enabled, for a range of static prefetch distances and two dynamic throttling mechanisms: near-side prefetch throttling (NST, our proposal) and feedback-directed prefetching (FDP, the state of the art). Figure 4 plots the results for all workloads. As we saw in Figure 1, workloads tend to respond to an increasing prefetcher distance in three ways: either prefetching should be as aggressive as possible (e.g., `parest`, `xalancbmk`, and most of the APEX workloads), prefetching consistently causes degradation (`cactus`, `fotonik3d`), or there is some optimal distance that causes a speedup but after which additional prefetches cause performance degradation (`bwaves`, `mcf`, `GTCP`). In most cases, both NST and FDP are able to match the best static optimum for each specific workload and in some cases outperform it (e.g., `bwaves`) when the optimum changes depending on the input set or because of application phase behavior.

To explain why this happens, Figure 5 plots the amount of late (top) and useless (bottom) prefetches, both normalized to the total number of prefetches issued. Late prefetches load useful data, but are not generated early enough to overlap the full latency. In these cases, increasing the distance to prefetch further ahead of the application can help. In `HPCG` for instance, a distance of one results in a late prefetch fraction of 93% and very limited benefit over no prefetching. Increasing the distance to 32 reduces the late prefetch fraction to 7.3% and results in a performance benefit of over 50% compared to the baseline. Here, near-side throttling is able to detect the late prefetches and responds by increasing the distance to the maximum configured value of $R_{max}$ with distance 32.

In contrast, `GTCP` sees only a small amount late prefetches, but suffers from large amounts of useless prefetching due to its irregular access patterns — a distance of one already incurs 35% of useless prefetches which increases to 80% with a distance of 32. Because these useless prefetches consume memory bandwidth that could have been used by the application, performance degradation occurs (beyond a distance of eight, GTCP saturates off-chip memory bandwidth). In this case, near-side throttling detects a late prefetch fraction that never exceeds the threshold of $F_{max} = 10\%$ and reduces the prefetch distance to its minimum setting of one.

Other workloads such as `mcf` perform best with a middle-of-the-road prefetch distance that balances some late prefetches with a low amount of useless prefetches. Figure 5 shows that for distance 8 and beyond, the amount of late prefetches is minimized, but that useless prefetches go up at the same time. Hence, the static results indicate `mcf` performs best with a distance of around 8. In fact, our dynamic algorithm outperforms the static options, at an average distance of 10.7. Note that this prefetch distance retains a certain amount of late prefetches (remember that the dynamic algorithm tunes the distance to generate about 10% late prefetches), but these are not necessarily detrimental to performance: even a prefetch that is late by one cycle is still counted as late, but has a negligible performance penalty given that most of the latency (which can be hundreds of cycles for an off-chip memory access) is still hidden from the application.

On average, a static prefetch distance of eight provides the best results, with a speedup over no prefetching of 6.6% albeit with large variations (and often significant degradation) between individual workloads and input sets. Near-side prefetching outperforms the static optimum, providing a speedup of 9.6%; which is similar to feedback directed prefetching's 9.4% yet NST has a much smaller implementation cost.

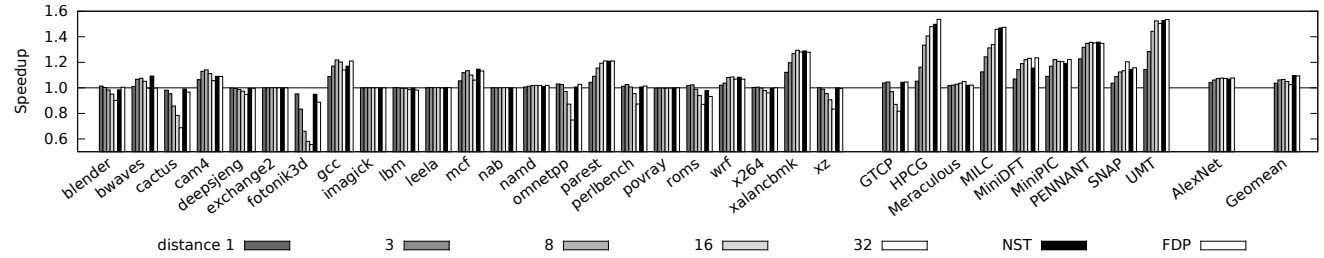


**Figure 4: Speedup for static prefetch distances, near-side throttling (NST) and feedback-directed prefetching (FDP). Dynamic throttling performs close to each workload's static optimum, and outperforms a global static optimum.**

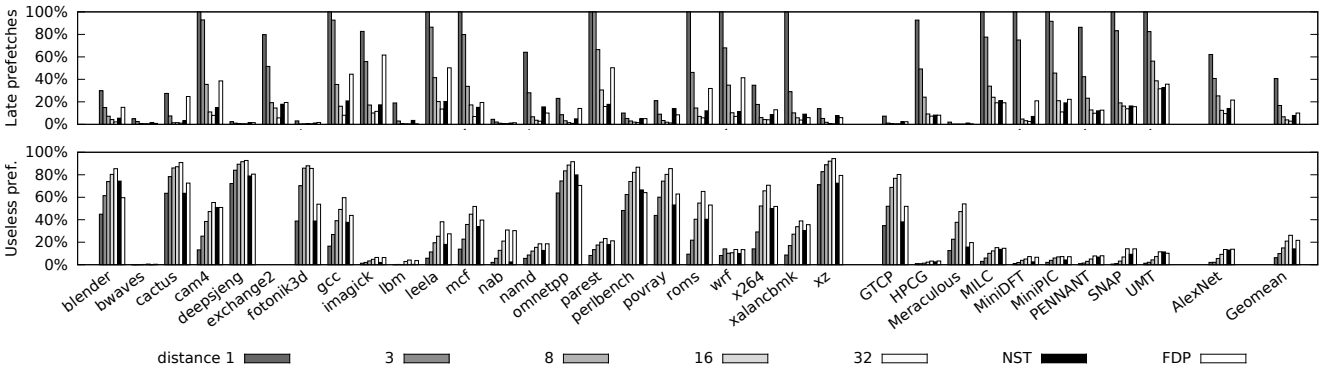


**Figure 5: Late prefetch fraction (top), and useless prefetch fraction (bottom), relative to the total number of prefetches.**

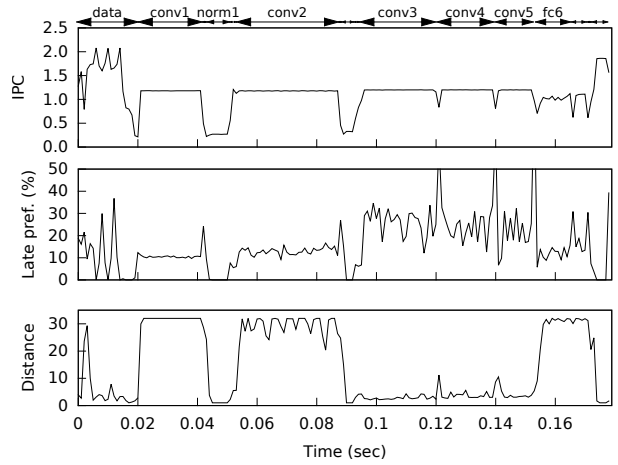


**Figure 6: Behavior of near-side throttling through time while running `AlexNet`.**

## 4.2 Dynamic behavior

When an application contains phase behavior, dynamic prefetching can select the optimal prefetch distance not just for the application as a whole but for each individual phase. Figure 6 plots the behavior of `AlexNet` through time over the course of one iteration, using near-side throttling to control the prefetcher distance. The markers at the top of the figure indicate the different layers of the AlexNet deep learning network, the timings of which were obtained by instrumenting the Caffe binary. From top to bottom, the graphs plot per-core IPC, the fraction of late prefetches, and the selected prefetch distance. The convolution layers `conv1` and `conv2` are data-intensive compute phases with linear accesses, here the stride prefetcher is able to generate useful prefetches and is set to its maximum distance. The later convolution layers `conv3`…`conv5` operate on much smaller data sets that are mostly cache-resident, after an initial burst of cache misses (during which the prefetch distance is increased temporarily) the miss rate reduces significantly. A relatively high fraction of late prefetches remains (>10%), but this misleading because the prefetch rate is so low (often less than 10 prefetches per update interval). Here, NST does not increase the prefetch distance because the bursts of late prefetches are alternated with long periods without prefetcher activity during which the distance reduces to its minimum value.

In contrast, FDP evaluates the prefetcher settings only once every 8,192 cache misses, which is 100× slower than the update interval of NST. FDP therefore does not see the intra-iteration variations of this workload and keeps the prefetcher distance at its most aggressive setting of 32, resulting in a 20% higher prefetch rate without an increase in useful prefetches — wasting off-chip bandwidth during some of the workload phases.

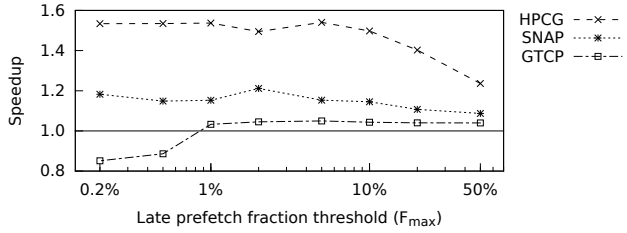


**Figure 7: Speedup of NST for a range of $F_{max}$ values.**

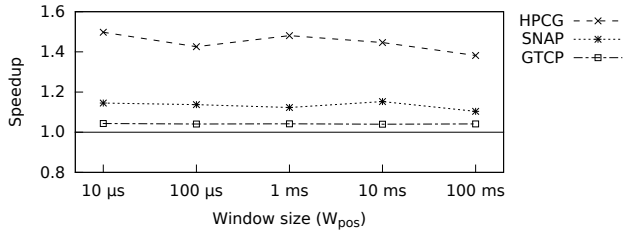


**Figure 8: Speedup of NST for a range of $W_{pos}$ values.**

## 4.3 Sensitivity analysis

When introducing an algorithm that should automatically tune a system parameter, it is not desirable to require a set of new tuning parameters that need to be set optimally to make the algorithm work well as this just moves the tuning effort, rather than being able to avoid it. While our algorithm does contain a number of new parameters, most notably $F_{max}$ and the window sizes $W_{pos}$ and $W_{neg}$, we show the algorithm retains its optimal behavior over a wide range of settings. This means there is no need to tune these parameters to individual systems or applications.

Figure 7 plots the application speedup, for a representative selection of workloads, while changing the threshold value for late prefetches. We used a value of $F_{max} = 10\%$ for all results shown earlier, but now sweep over a range of 0.2%–50%. A very large threshold (>10%) retains too many late prefetches even in stable conditions, and causes performance degradation. Lowering the threshold below 2% makes the algorithm too aggressive even in the presence of unavoidable late prefetches. A value of $F_{max}$ between 2% and 10% is able to provide optimal working point for most applications.

In Figure 8 we explore the size of the update window. The length of this window should balance the need to make a reliable measurement of the current application behavior, while being short enough to be able to react to changes in application phase behavior. We maintain a constant ratio between the positive and negative windows (used while increasing or decreasing the distance, respectively) of $W_{pos} = 2 \cdot W_{neg}$, and sweep $W_{pos}$ over a range between 10 µs and 100 ms. Very short intervals (<10 µs) do not contain enough completed prefetches (off-chip memory latency is typically a few hundred nanoseconds) to make a useful measurement. Beyond a window of 1 ms the algorithm becomes too slow to react to application phase changes. Between 10 µs and 1 ms is a wide range that provides the best trade-off. We used $W_{pos} = 10$ µs and $W_{neg} = 5$ µs for the results in this paper. Note that techniques that rely on measuring useless prefetches such as FDP, require update windows that are in the order of several milliseconds, as that is the average lifetime of a cache block in our experiments. This makes far-side techniques respond much slower to aplication phases than near-side throttling.

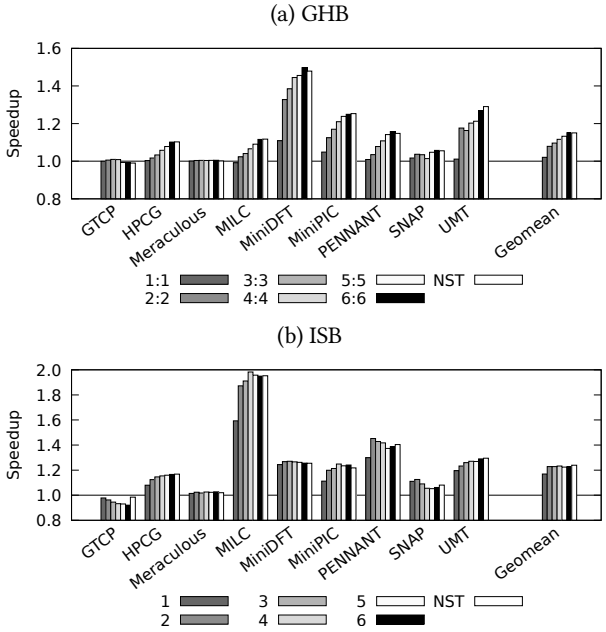


**Figure 9: Speedup of NST applied to the GHB and ISB prefetchers.**

## 4.4 Modern prefetchers

Thus far we evaluated NST with a simple stride prefetcher, where there is a natural relation between timeliness and accuracy through the distance parameter: prefetching with a higher distance increases timeliness of strided access patterns, but at the same time reduces accuracy on those access patterns that consist of short strides or random accesses. Many modern prefetcher algorithms have been proposed that provide higher accuracy. Near-side throttling can be applied advantageously to these newer prefetchers as well, as seen in Figure 9. We apply NST to the Global History Buffer prefetcher [23] (GHB, top) and Irregular Stream Buffer [13] (ISB , bottom) prefetchers and show results for the APEX benchmark suite (other workloads behave in a similar manner). Whereas the stride prefetcher caused degradation of GTCP at higher distances, these newer prefetch algorithms do not have this disadvantage. Still, applying NST to them allows us to approach, and in some cases improve on the statically optimum prefetch distance for each workload.

## 4.5 Multiple prefetchers

Thus far the L2 prefetcher consisted of a single stride-based prefetch algorithm. Near-side throttling is able to control multiple prefetchers, and will naturally allow the most accurate algorithm to issue the largest number of prefetch requests, during each application phase.

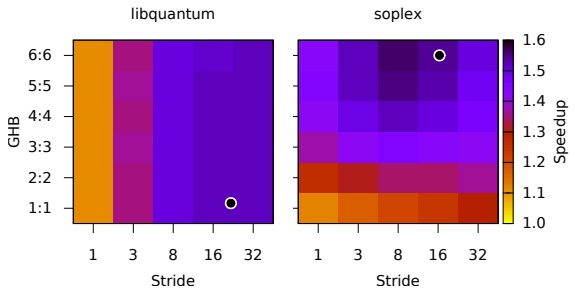


**Figure 10: Combination of stride and GHB prefetcher.**

For the results shown in Figure 10, we combine a stride prefetcher with a global history buffer based prefetcher [23] at each L2 cache, and plot the speedup (over the baseline with only L1 prefetchers) for a few example applications. We sweep the stride prefetcher distance between 1 and 32, while simultaneously sweeping over the GHB's parameters (width : height) from 1 : 1 to 6 : 6. We also plot the operating point that is selected by the dynamic distance (black marker, averaged over the application).

`libquantum` benefits from the stride prefetcher and reaches an optimal performance at a distance between 16 and 32, but is neutral with respect to the GHB. Our algorithm selects a stride prefetch distance of 23 while minimizing the activity of the GHB. While this application does not hit the off-chip memory bandwidth and the GHB's extra, useless prefetches do not affect performance in this case, minimizing its activity does reduce the number of off-chip memory accesses. This results in energy savings that can (for a power-constrained environment) be used to increase clock frequencies and lead to an additional performance advantage.

In contrast, `soplex` has a more complex memory access pattern which does benefit from the GHB prefetcher. Some amount of stride prefetching is useful yet a stride prefetcher distance over 16 generates too many useless prefetches. For this application, near-side throttling tunes the activity of the GHB up to its maximum 6 : 6 setting, while selecting a distance of 16 for the stride prefetcher, which is near the static optimal configuration.

## 5 SOFTWARE PREFETCHING

Finally, we look at how near-side throttling can be applied to software prefetching. Today, the programmer has to select a prefetch distance, typically expressed statically in a number of loop iterations to prefetch ahead. The optimal distance is based on the length of a loop iteration and the off-chip memory access latency. Both parameters are architecture dependent, which necessitates either machine-specific tuning of the application, or large tables of precomputed prefetcher distances for various architectures. Neither approach can be considered programmer friendly, or provides much performance portability. Moreover, both the loop length and average memory access latency can change at runtime, through effects such as dynamic frequency scaling, competition for off-chip bandwidth from other cores, or whether the loop is operating on data that is in main memory or in cache.

```
void spmv_csr(int nrows, double* y, int* row_ptr,
              int* cols, double* values, double* x)
{
   for(int row = 0; row < nrows; ++row)
   {
      double tmp = 0;
      int dist = __read_msr(PF_DIST);
      for(int i = row_ptr[row]; i < row_ptr[row+1]; i++)
      {
         __builtin_prefetch( &x[cols[i+dist]] );
         tmp += values[i] * x[cols[i]];
      }
      y[row] = tmp;
   }
}
```

**Figure 11: Source code of an SpMV kernel with software prefetching.**

We propose to assist software prefetching by having the hardware compute the optimal software prefetch distance. As with hardware prefetches, we add a tracking bit $p_{swp}$ to each MSHR entry to track memory requests initiated by software prefetches, and keep $a_{swp}$ and $l_{swp}$ counters to compute the fraction of late software prefetches. These counters can be exposed to software as performance counters, or the algorithm from Figure 2 can be implemented fully in hardware to expose the prefetch distance as an architectural register.

We test our implementation with a sparse matrix-vector product kernel listed in Figure 11. This kernel computes $y = Ax$, with $x$ and $y$ being dense vectors while $A$ is a sparse matrix defined in compressed row storage format given by the `row_ptr`, `cols` and `values` variables. The arrays defining the matrix $A$ are accessed linearly, and hence will benefit from hardware prefetching. The input vector `x` has a non-linear access pattern, based on the non-zero elements of the sparse matrix, and will rely on software prefetching. We prefetch `x` at a distance of `dist` iterations ahead, where `dist` is either hard-coded to a static value or set by near-side throttling.

Figure 12 plots the performance of the SpMV kernel when varying the prefetch distance. The kernel was configured to use 1M double-precision elements in both X and Y dimensions, and a randomly generated matrix A with 4M non-zero elements. This means that all of the main structures reside in main memory. Compared to no software prefetching, a performance gain of up to 4.8× is possible. The best performance, given the memory access latency for this particular architecture, occurs at a distance between 16 and 64. For a distance shorter than 16, many prefetches are late so the memory access latency is not fully overlapped. When prefetching more than 4096 iterations in advance, the `x` values are evicted from the L2 cache before they are needed. For distances between 128 and 2048, software prefetching is still effective in the sense that there is no cache pollution, but here there is TLB pollution. When prefetching 64 or fewer iterations ahead, software prefetching is able to warm up both the L2 cache and the D-TLB.

Near-side prefetch throttling, when run with this particular architecture and matrix properties, converges on a distance of 17 iterations, which is at the low end of the optimal range. While the

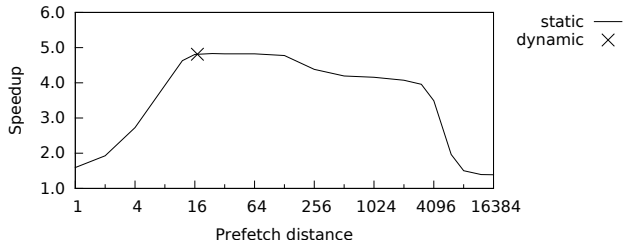


**Figure 12: Performance of SpMV kernel with varying distances of software prefetching.**

algorithm does not explicitly consider cache pollution nor TLB effects, it is still able to benefit from both types of warmup, exactly because we keep the prefetch and its use as close to each other as possible. In contrast, an approach that only detects useless prefetches would not throttle until a distance of about 4096, and hence not be able to benefit from the TLB warmup effect.

## 6 RELATED WORK

Prefetching and how to control prefetcher aggressiveness has been extensively studied in prior work. In this section, we provide an overview of prior work related to our study. We first describe methods that require modifications to the hardware to reduce the negative impact of too aggressively prefetching, later we discuss software solutions to steer prefetching.

### 6.1 Hardware solutions

Multiple techniques [4, 10, 29], which we call far-side throttling mechanisms in this paper, require additional instrumentation in the cache to estimate the negative impact of prefetching. These methods keep track of which cache blocks were brought in by the prefetcher, clear the prefetch flag when the core does an access to that block, and mark the prefetch as useless if the block gets evicted from the cache without being accessed by the core. This way, metrics such as prefetch accuracy can be calculated dynamically. However, we see a number of drawbacks with these mechanisms. First, they have a non-negligible hardware cost because extra tracking bits are needed for each cache block. Second, they react slowly to phases in the application because, especially for large cache sizes, it can take a significant amount of time before blocks get evicted. This means the measured useless prefetch metric trails behind application behavior by a significant amount of time. Last, these methods typically keep the prefetch distance high until they detect a negative impact on performance, which can miss causes of degradation that are not explicitly measured, such as TLB warmup as we saw in the case of software prefetching, or the displacement of useful data by useful prefetches. This is the opposite from what we are proposing in this paper, which is to keep the prefetch and its use closely together, thus ensuring effective warmup of as many processor structures as possible while using the least amount of cache capacity to hold prefetched data.

Ebrahimi et al. [10] present a mechanism to estimate interference between cores caused by prefetching. The algorithm consists of a local component which is similar to [29], and a global component which takes interactions between cores into account (e.g., saturating shared memory bandwidth), and can override decisions made by the local component. They apply their scheme to multiple prefetchers on multiple caches which is different from multiple prefetchers on the same cache as we do in this paper. Later in their follow-up work [9], they improve this mechanism by taking fairness between cores into account. Similar to this is the work done by Panda [24], who introduces a synergistic prefetcher aggressiveness controller (SPAC) that aims to achieve a fair speedup when making prefetch throttling decisions. In this paper, we target high-performance processors, where speedup is more important than fairness between cores, however our mechanism could easily be extended with a similar approach as [24] to incorporate fairness.

Pugsley et al. [25] propose Sandbox Prefetching, a technique where they evaluate the accuracy of the prefetcher using a Bloom filter, and if prefetcher accuracy exceeds a threshold, they allow the prefetcher to start fetching data into the cache. Other work [21, 27, 30] makes a distinction between demand and prefetch requests in the cache replacement policy to avoid cache pollution caused by prefetching. However, all of these proposals do not dynamically change the configuration of a prefetcher.

Hur and Lin [11] propose to dynamically adjust the priority of prefetch requests over demand requests in the memory controller. This allows to postpone useless prefetch requests, but can still lead to cache pollution when prefetch accuracy is low. Later work by Lee et al. improves on this with prefetch-aware DRAM controllers [19]. By making the memory controller aware of prefetch accuracy, useless prefetches can be dropped, and useful prefetches can be prioritized. However, the method to calculate prefetch accuray is still a type of far-side throttling similar to [29], with the disadvantages discussed earlier.

### 6.2 Software solutions

Most studies to control prefetcher aggressiveness offer hardware solutions, but in literature software solutions exist too. Wu and Martonosi [31] characterize LLC interference using hardware performance counters, and propose a prefetch managemement scheme to dynamically turn on or off the prefetcher. Similar to this is the work of Liao et al. [20], they propose a prefetcher tuning framework, using machine learning models, and based on the input from hardware performance counters. Unlike we do, the hardware setup used in both works does not allow to dynamically change the prefetching distance. Therefore, these proposals are more coarse-grained than our solution. Jiménez et al. [14] propose a more fine-grained solution, their hardware platform is more flexible, and they built a software scheme to dynamically reconfigure the prefetcher. However, most of the commercially available processors today have no support for this fine-grained prefetcher reconfiguration.

## 7 CONCLUSION

We propose near-side prefetch throttling, a technique that detects the fraction of late prefetches and uses it to dynamically select an appropriate prefetch aggressiveness. We show this technique is applicable in several situations, including controlling single and multiple hardware prefetchers in a high-performance many-core processor, and can be used to control software prefetching. Our

technique naturally integrates detection of off-chip bandwidth saturation and throttles the prefetch rate accordingly.

Using detailed simulations we measure application performance over a range of workloads, and show that our method can quickly adapt to application behavior, to match, or in some cases exceed, the best static optimal prefetch distance with only minimal hardware cost. This makes near-side throttling superior over traditional far-side throttling as it is able to provide even slightly better performance (9.6% vs. 9.4%), at a far cheaper implementation cost, and is more widely applicable to other use cases such as software prefetching and control of multiple hardware prefetchers.

## REFERENCES


[1] 2016. APEX Application Benchmarks. http://www.lanl.gov/projects/apex/

[2] 2016. Intel Caffe. https://github.com/intel/caffe

[3] 2017. SPEC CPU2017 benchmark suite. https://www.spec.org/cpu2017/

[4] Alaa R. Alameldeen and David A. Wood. 2007. Interactions Between Compression and Prefetching in Chip Multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 228–239.

[5] Jean-Loup Baer and Tien-Fu Chen. 1991. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 176–186.

[6] Jean-Loup Baer and Tien-Fu Chen. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.* 44 (1995), 609–623.

[7] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 248–255.

[9] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 141–152.

[10] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 316–326.

[11] Ibrahim Hur and Calvin Lin. 2006. Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 397–408.

[12] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access Map Pattern Matching for High Performance Data Cache Prefetch. *Journal of Instruction-Level Parallelism* 13 (2011), 1–24.

[13] Akanksha Jain and Calvin Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 247–259.

[14] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 137–146.

[15] Norman P. Jouppi. 1990. 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 (ISCA)*. 364–373.

[16] Prathmesh Kallurkar and Smruti R. Sarangi. 2016. pTask: A Smart Prefetching Scheme for OS Intensive Applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1–12.

[17] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path Confidence Based Lookahead Prefetching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1–12.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*. 1097–1105.

[19] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-Aware DRAM Controllers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 200–209.

[20] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine Learning-Based Prefetch Optimization for Data Center Applications. In *Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 56:1–56:10.

[21] Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. 2001. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 301–312.

[22] Pierre Michaud. 2016. Best-Offset Hardware Prefetching. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 469–480.

[23] Kyle J. Nesbit and James E. Smith. 2005. Data Cache Prefetching Using a Global History Buffer. *IEEE Micro* 25, 1 (2005), 90–97.

[24] Biswabandan Panda. 2016. SPAC: A Synergistic Prefetcher Aggressiveness Controller for Multi-Core Systems. *IEEE Trans. Comput.* 65, 12 (Dec 2016), 3740–3753.

[25] Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 626–637.

[26] Andres Rodriguez. 2016. Training and Deploying Deep Learning Networks with Caffe* Optimized for Intel® Architecture. Intel Developer Zone.

[27] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (Jan. 2015), 51:1–51:22.

[28] Avinash Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor. In *Hot Chips 27 Symposium*.

[29] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 63–74.

[30] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. PACMan: Prefetch-Aware Cache Management for High Performance Caching. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 442–453.

[31] Carole-Jean Wu and Margaret Martonosi. 2011. Characterization and Dynamic Mitigation of Intra-Application Cache Interference. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–11.

[32] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 178–190.