Appears in the 2001 International Conference on Computer Design

Filtering Superfluous Prefetches using Density Vectors

Wei-Fen Lin, Steven K. Reinhardt
EECS Department
University of Michigan
{wflin,stever}@eecs.umich.edu

Doug Burger
Dept. of Computer Sciences
University of Texas at Austin
dburger@cs.utexas.edu

Thomas R. Puzak
IBM Corporation
T. J. Watson Research Center
trpuzak@us.ibm.com

Abstract

A previous evaluation of scheduled region prefetching *showed that this technique eliminates the bulk of main-memory stall time for applications with spatial locality. The downside to that aggressive prefetching scheme is that, even when it successfully improves performance, it increases enormously the amount of superfluous memory traffic generated by a program. In this paper, we measure the predictability of spatial locality using* density vectors, *bit vectors that track the block-level access pattern within a region of memory. We evaluate a number of policies that use density vector information to filter out prefetches that are unlikely to be useful. We show that, across our benchmarks, an average of 70% of useless prefetches can be eliminated with virtually no overall performance loss from reduced coverage. Thanks to the increase in prefetch accuracy, a few benchmarks show performance improvements as high as 35% over the base region prefetching scheme.*

1. Introduction

Long memory latencies continue to degrade application performance. Previous work showed that main-memory access stalls accounted for 60% of SPEC CPU2000 runtime on a next-generation processor with an aggressive memory system [5]. That work also described *scheduled region prefetching*, a scheme for aggressively prefetching from large regions around each demand miss, which nearly eliminates those stalls for applications with high spatial locality. By combining request prioritization and DRAM-aware scheduling on the memory channels with low-priority loading of prefetches into the L2 cache, the proposed scheme avoids the penalties associated with increased memory channel contention, DRAM row buffer misses, and cache pollution. These techniques serve both to improve performance for applications with exploitable locality and to prevent performance degradation for applications that have little locality.

However, the scheduled region prefetching scheme indiscriminately prefetches everything in a large region, issuing many useless prefetches for applications without perfect spatial locality. Although the performance impact of useless prefetches in this scheme is minimal, superfluous traffic consumes power, reduces the number of useful prefetches that may be issued, and aggravates bandwidth limitations in a multiprocessor environment. In this paper, we extend scheduled region prefetching to select prefetch candidates more judiciously, attempting to eliminate the issue of useless prefetches. Specifically, we record locality patterns using *density vectors*—bit vectors that track access patterns within a memory region—and use these vectors to predict future locality. Our results show that we can eliminate 70% of useless prefetches, improving mean prefetch accuracy from 46% to 65%, without reducing mean prefetching performance gains. Although some of our benchmarks see slightly lower performance relative to the unfiltered prefetching scheme due to reduced coverage, the higher accuracy increases performance on others—by as much as 35% in one case.

In the next section, we discuss the most pertinent of the vast body of literature on prefetching. In Section 3, we measure several applications to determine the predictability of spatial locality patterns. In Section 4, we propose several prefetching policies motivated by these measurements, and measure their efficacy. We conclude in Section 5.

2. Related work

Previous studies have sought to exploit spatial locality while balancing memory traffic and cache pollution by dynamically fetching or prefetching a quantity of data on a cache miss, which was determined by predicted locality. These quantities were either variable-sized contiguous blocks or patterns of blocks within a larger region. The locality prediction may be based on profile data [2,7], compiler annotations [6], or dynamic hardware detection [3,4,7]. The scheduled region prefetching scheme, includ-

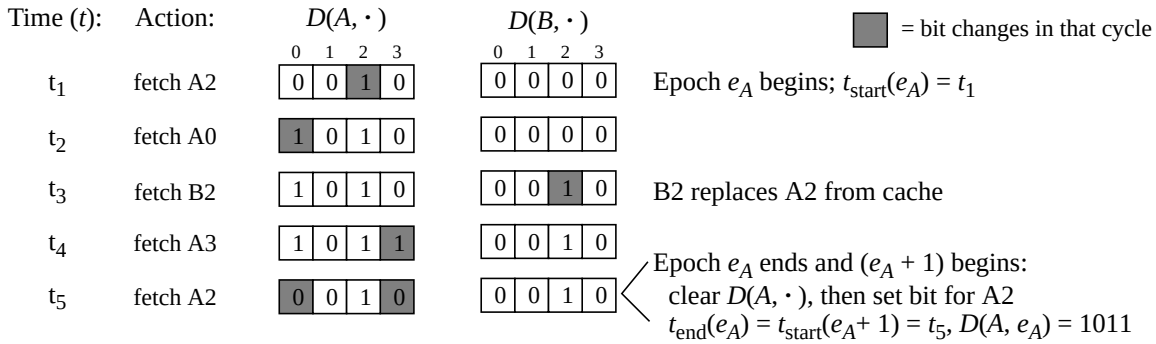

Figure 1. Density vector and epoch example

ing the filtering additions we study in this work, is purely hardware based. Unlike earlier works, which must use conservative locality estimates to avoid bandwidth contention and cache pollution, the prioritized scheduling and placement techniques on which we build already address these performance issues. Thus, in this work, we seek primarily to avoid fetching useless data rather than to identify precisely the most useful data.

Our density vector techniques are very similar to earlier proposals by Kumar and Wilkerson [4] and Burger [1]. Kumar and Wilkerson propose a Spatial Footprint Predictor (SFP) to predict the portions of a cache block that will get used before eviction. Our technique for measuring density vectors borrows from their spatial footprint work. However, they tracked doubleword-level locality within an L1 cache block to improve traffic efficiency relative to fetching conventional cache blocks. We track block-level locality within a large region to reduce superfluous traffic relative to an aggressive prefetching scheme.

3. Quantifying spatial locality and its predictability using density vectors

In this section, we define metrics to estimate both the available spatial locality in a program and its predictability. These metrics will indicate both whether enough locality exists for aggressive prefetching to be worthwhile, and whether this locality is sufficiently predictable to permit differentiation between likely good and likely bad prefetches. *Density vectors* form the cornerstone of our measurement methodology.

3.1. Density vectors

We define a density vector as a bit vector that records the set of cache blocks within a region of the address space, which are fetched from memory during a particular period of time (an *epoch*). Epoch boundaries are determined on a per-region basis. A region's current epoch ends, and its next epoch begins, when any block in the region is fetched for the second time (*i.e.*, a miss to a block occurs and the block is already in the density vector.) This policy is the same as was used by Kumar and Wilkerson [4].

We refer to a particular density vector within a program's execution as $D(x, e_x)$. That notation represents the density vector for epoch e_x of a region x. We denote the current epoch for region x at time t as $e_x(t)$. The starting and ending times of epoch e_x are $t_{\text{start}}(e_x)$ and $t_{\text{end}}(e_x)$, respectively.

In Figure 1, we depict an example of the construction of density vectors. We show the density vectors under construction for the current epoch for two four-block regions, *A* and *B*, denoted $D(A, \cdot)$ and $D(B, \cdot)$, respectively. The shaded squares represent the density vector bits that are changed in a given cycle. Block 2 of region *A* is fetched first, then block 0 of region *A*, and so on. When block 2 of region *A* is replaced by block B2, the corresponding bit in $D(A, e_A)$ is not cleared. However, the loading of this block for the second time—detected as a miss to a block already marked in the current density vector—triggers the end of epoch e_A. This reference then becomes the initial reference of epoch $e_A + 1$.

3.2. Density-vector metrics

We define several metrics to help measure density vector locality and consistency. The number of bits in a density vector is $b = \log_2(P/B)$, where P is the region size and B is the L2 block size. We define the *population count*, or PC, to be the number of bits in a density vector that are set. PC thus becomes a good measure of available spatial locality. Another definition is *longest run length*, or LRL, which represents the longest contiguous string of set bits in a vector. The LRL metric estimates whether the reference pattern is dense or sparse.

We define *correlation* as the fraction of bits that are identical between two density vectors. Formally, we define the correlation between two *b*-bit vectors *X* and *Y* to be:

$$CORR(X, Y) = \frac{PC(\overline{X \oplus Y})}{b} \qquad (1)$$

where $\oplus$ indicates the bit-wise exclusive-or operation.

3.3. Experimental Methodology

We simulate the same target system, using the same parameters, as in previously published region prefetching work [5]. The simulated CPU is a four-wide out-of-order superscalar with a 64-instruction issue window. The memory system includes 64KB, 2-way set-associative non-blocking L1 instruction and data caches, a 1MB 4-way set-associative non-blocking L2 cache, and a 4-channel Direct Rambus memory system. The region size for which we prefetch is 4KB, and all block sizes are 64 bytes. Density vectors thus contain $b = 64$ bits.

Previous work [5] defined memory-intensive benchmarks to be those that incurred more than 0.75 L2 misses per 1000 instructions. In this paper, we use a representative set of memory-intensive benchmarks from three different classes: (1) five benchmarks that showed good spatial locality (*equake, facerec, fma3d, mgrid, parser*), (2) three benchmarks that showed poor spatial locality (*ammp, bzip2, vpr*), and (3) a bandwidth-bound benchmark (*art*). Due to space considerations, we do not present the rest of the SPEC2000 benchmarks.

For each benchmark, we begin simulation from a checkpoint 20 to 60 billion instructions into the program, simulate for 2 billion instructions to warm up the caches and density-vector information, then record statistics for the following 500 million instructions of program execution.

3.4. Metrics for Correlation

In this section, we define metrics for quantifying the correlation that exists among density vectors in a program. We measure three types of correlation.

• *Local correlation:* This metric, abbreviated L-Cor, measures the correlation between the two most recent epochs in the same region. If the access patterns to a region remain constant over time, this local correlation will be high (close to 1).

$$L\text{-}Cor(x, e_x) = CORR(D(x, e_x), D(x, e_x - 1)) \quad (2)$$

• *Spatial correlation*: This metric, abbreviated S-Cor, measures the correlation between adjacent regions, indicating whether the recent density vectors of nearby regions will indicate the future access pattern for a given region.

$$S\text{-}Cor(x, e_x) = CORR(D((x, e_x), D(x - 1, e_x))) \quad (3)$$

• *Global correlation*: This metric, abbreviated G-Cor, measures the correlation based on the phase the application is currently in, by comparing the two most recent density vectors to be computed, regardless of their address.

$$G\text{-}Cor(x, e_x) = CORR(D(x, e_x), D(y, e_y)) \quad (4)$$
$$y, e_y \mid [\ t_{\text{end}}(e_x) > t_{\text{end}}(e_y) > t_{\text{end}}(e_z)\ \ \forall e_z, \quad z \neq x,y\ ,\ t_{\text{end}}(e_x) > t_{\text{end}}(e_z)\ ]$$

3.5. Measuring Correlation

In Figure 2, we show cumulative distribution functions of our locality and correlation metrics for our nine memory-intensive benchmarks. As with the correlation metrics, population count (PC) and longest run length (LRL) are also divided by *b*, so that all metrics fall between zero and one. A PC value of one indicates that all bits in the density vector are set; similarly, an LRL value of one indicates that the longest run of set bits encompasses the entire vector. PC is always greater than LRL, since every set bit in the longest consecutive run also counts toward the population count. A "perfect" correlation, in which every density vector matches exactly the vector against which it is compared, results in a value of one. A point on the horizontal (x) axis represents the fraction of density vectors examined that have a population count, longest run length, or correlation less than or equal to the y-value of the point in question.

For the nine benchmarks shown in Figure 2, the PC and LRL curves indicate a significant variation in spatial locality patterns. Some of the benchmarks have strong spatial locality, that is, the majority of blocks in every region are referenced (*equake, fma3d*). This situation is reflected by the population count curve quickly approaching 1 on the left-hand side of the graph. *Ammp* and *vpr* have little spatial locality: fewer than 20% of density vectors include more than 20% of the blocks in a region. *Bzip*'s population counts are only slightly higher; the large gap between the PC and LRL curves for *bzip* indicate that even when multiple blocks within a region are accessed, the pattern is sparse. For these latter benchmarks, prefetching every block in a region will generate significant wasted traffic.

Several benchmarks exhibit a bimodal spatial locality pattern. This pattern is illustrated best by *mgrid*, where the PC curve has a single nearly vertical section, reaching from near 0 to near 1 on the y axis, close to the 30% mark on the x axis. This shape indicates that nearly 30% of epochs have no spatial locality, while the remaining 70% have very strong spatial locality (the entire region is accessed). For *mgrid*, the LRL curve follows the PC curve, indicating that all of the spatial locality is comprised of dense access patterns. Other benchmarks exhibit a slightly different bimodality. For example, in *facerec*, the PC curve has two vertical sections (at 0% and near 75% on the x axis, with a crossover near 0.4 on the y axis), while the LRL curve has a single vertical section near 75%. This pattern indicates that about 25% of epochs show a dense access pattern covering the entire region, while the remaining epochs have only 40% referenced blocks in a sparse pattern.

Turning to the correlation curves in Figure 2, we see that most of the benchmarks show strong correlations for at least one of our correlation types. A strong correlation is

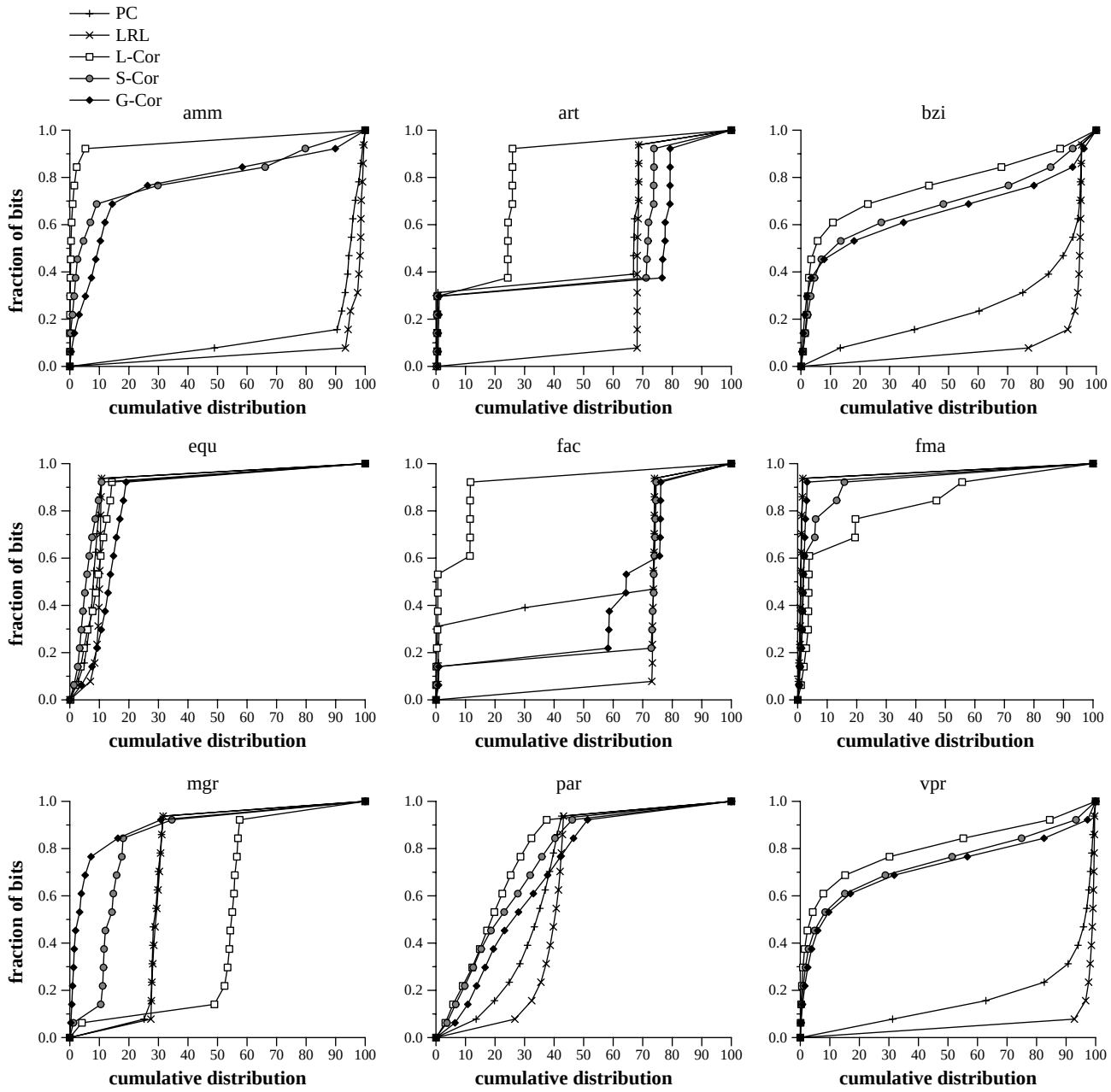

Figure 2. Density vector correlations and statistics

indicated by high y-axis values at low x-axis coordinates. In addition, if one curve is above another in the figure, the correlation represented by the first curve is stronger than that represented by the second.

Overall, the local correlation (L-Cor) is the most accurate of the three we measure; it shows the highest accuracy for six of the nine benchmarks. Spatial correlation (S-Cor) is slightly stronger in *equake*, but all three correlations are high for that benchmark. Global correlation (G-Cor) is generally slightly lower than L-Cor, but is much higher than either local or spatial correlation for *mgrid*, and somewhat higher for *fma3d*. However, G-Cor is much lower than L-Cor in *art* and *facerec*.

These results show both that there is significant variance in the number of blocks used in regions both across and within benchmarks, and that, for most benchmarks, the local correlation (L-Cor) of density vectors is strong. Spatial correlation (S-Cor) never shows a significant advantage over L-Cor, and will not be considered further in this paper. Global correlation (G-Cor) is generally slightly weaker

than L-Cor, but may be significantly better or worse than L-Cor for individual benchmarks. In the next section, we measure our ability to exploit these local and global correlations to reduce unnecessary prefetch traffic while maintaining high coverage for benchmarks with spatial locality.

4. Improving scheduled region prefetching using density vectors

Through careful scheduling and placement, scheduled region prefetching (**srp**) avoids the performance losses generally associated with prefetching schemes: pollution and contention. However, **srp** still generates a high quantity of useless prefetches. Although the bad prefetches do not degrade performance, they consume power, may interfere with multiprocessor performance, and may cause performance losses due to missed opportunities to issue good prefetches. In this section, we exploit the correlation explored in Section 3, using density vectors to filter out prefetches that are unlikely to be needed by the processor. Ideally, this filtering will reduce traffic with no losses in prefetch performance, and perhaps even slight gains. In Section 4.1, we review **srp**. In Section 4.2, we show how density-vector filters may be added to **srp**, and describe a number of possible filter policies. In Section 4.3, we evaluate the efficacy of those policies assuming unlimited storage for density vectors. Finally, in Section 4.4, we quantify the losses due to finite density vector buffering.

4.1. Scheduled region prefetching

Srp aggressively exploits spatial locality by prefetching every block in an aligned region surrounding a demand miss that is not already in the cache [6]. For example, a cache with 64-byte blocks and 4KB regions would fetch the required 64-byte block upon a miss, and then prefetch all of the 63 other blocks in the surrounding 4KB-aligned region not already resident in the cache.

Aggressive prefetching can reduce performance in three ways: by competing with demand misses for bandwidth, by interfering with DRAM row buffer locality, and by polluting the cache with useless data. **Srp** uses three mechanisms to avoid each of those sources of performance loss: reducing the priority of prefetch requests in memory-channel scheduling (to eliminate the bandwidth-related performance loss), prioritizing prefetches to maximize row buffer hits (to minimize DRAM row buffer interference), and placing prefetches in the lowest-priority replacement position in the L2 cache sets (to minimize cache pollution-related performance loss). The memory channel scheduler functions by issuing prefetch requests only when the memory channels are otherwise idle.

In Figure 3, we depict the proposed memory controller design that implements this scheduling policy. (The shaded

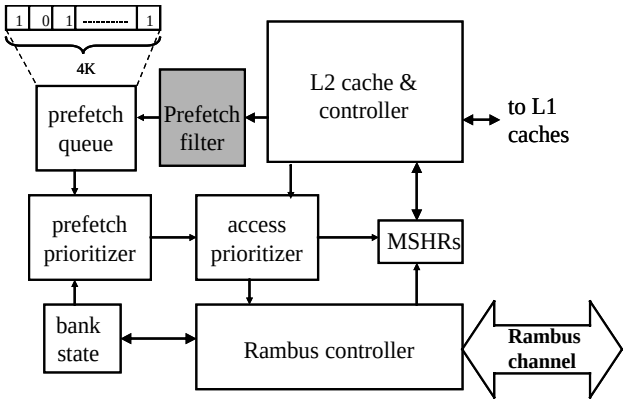

Figure 3. Throttling prefetch memory controller

prefetch filter component is not part of the original design, and will be discussed in the following section.) The prefetch queue maintains a list of region entries. Each entry is a bit vector, with one bit for each block in the region. A bit in the vector is set if the corresponding block is to be prefetched. When a demand miss occurs, an entry for the surrounding region is added to the prefetch queue, if one is not already present. The L2 cache tags are probed to initialize the vector, marking only those blocks not already in the cache. We assume that these probes can occur in the background, using idle tag ports, but do not model this process in detail. Within a region, prefetches are issued in linear order beginning with the block after the demand miss, and wrapping around at the end of the region. If a demand miss or writeback arrives at the access prioritizer, it is issued as soon as possible, interrupting the flow of prefetch requests from the prefetch prioritizer. Prefetches thus add little additional channel contention, delaying a demand miss only until the current prefetch finishes issuing.

The base design used in this paper is enhanced from that of previous work [5] in two ways. First, this Rambus controller schedules demand fetches more aggressively, interleaving the row and column packets of accesses to independent banks to better hide precharge latencies where possible. The previous study pipelined Rambus demand requests, but did not reorder them. Second, we tuned the prefetch queue size, finding that a two-entry queue provides the best overall performance; deeper queues do not benefit benchmarks with locality, but penalize benchmarks without locality by buffering large numbers of useless prefetch candidates.

This base design improves performance by an average of 65% across the ten SPEC CPU2000 benchmarks with significant spatial locality, without degrading performance on the remaining 16 benchmarks. However, as demonstrated in Section 4.3, memory channel utilization is increased significantly for the base **srp** scheme, particularly for the benchmarks with little spatial locality.

4.2. Prefetch filter design

The prefetch filter shown in Figure 3 is intended to eliminate the bulk of the useless prefetches. The filter takes the prefetch candidates nominated by the L2 cache (*i.e.*, all non-resident blocks within the region of a demand miss), and filters out the set of candidates that it predicts will not be useful based on its internally stored density-vector history. We model the filter as a function that provides a bit mask indicating the predicted useful blocks within a given region at a given point in time. We denote this function as $y = PF(x, t)$, where y is the bitmap that indicates the predicted useful blocks within region x at time t. The vector of prefetch candidates added to the prefetch queue is then the intersection (bit-wise AND) of the vector of non-resident blocks provided by the L2 cache and the vector of predicted useful blocks provided by the filter function.

The original **srp** scheme effectively used the following filter:

$$PF_{\text{orig}}(x, t) = 111....111_2 \quad (5)$$

Density vectors can be used as a prefetch filter in a straightforward fashion. A simple prefetch filter that exploits local correlation (L-Cor) among density vectors can be represented as:

$$PF_{\text{ldv}}(x, t) = D(x, e_x(t) - 1) \quad (6)$$

Note that time t is in the middle of epoch $e_x(t)$, so $D(x, e_x(t) - 1)$ is the latest complete density vector for the region containing x.

Because our scheduling and placement techniques allow more aggressive prefetching, we also examine a pair of less conservative prefetch filters using the union (bit-wise OR) of multiple previous density vectors. The **or2** filter combines the previous two density vectors for a region:

$$PF_{\text{or2}}(x, t) = D(x, e_x(t) - 1) + D(x, e_x(t) - 2) \quad (7)$$

The least conservative filter, **orN**, selects the blocks that have been referenced at any point in the past, by maintaining the union of all previous density vectors. We do not present results for **orN** in the paper, since it performed worse in all cases than other filters.

$$PF_{orN}(x, t) = \sum_{e_x=0}^{e_x(t)-1} D(x, e_x) \quad (8)$$

Finally, the prefetch filter that exploits global correlation (G-Cor) uses the most recently recorded density vector from *any* region:

$$PF_{\text{gdv}}(x, t) = D(y, e_y(t) - 1) \quad (9)$$
$$y, e_y \,|\, t > t_{\text{end}}(e_y) > t_{\text{end}}(e_z) \;\; \forall z \neq y$$

Implementations of these prefetch filters require two types of storage: a set of current density vectors under construction, as illustrated in Figure 1, likely maintained by the cache controller; and a set of previously recorded density vectors accessed by the filter mechanism. Throughout this paper, we assume adequate storage for all density vectors under construction. In the following section, we also assume infinite storage for recorded density vectors. We address the impact of finite storage for these vectors in Section 4.4.

4.3. Results

In Figure 4, we show the memory channel utilization and IPC for each benchmark across a variety of configurations. The stacked bars represent memory channel utilization, divided into three categories. From bottom to top, these categories are demand miss traffic, useful prefetch traffic, and useless prefetch traffic. The sum of the lower two bars is the total useful traffic, and represents a nearly constant number of bytes for each benchmark across all configurations; the combined height of these bars varies with performance because we plot utilization, rather than absolute traffic. The line indicates performance in instructions per cycle (IPC), calibrated against the right axis.

At the left of each graph, the first two bars represent the base system with no prefetching (**no-pf**) and a system that uses the last local (per-region) density vector to choose prefetches (**ldv**). The **ldv** policy, which includes none of the **srp** optimizations, is similar to the Kumar and Wilkerson scheme (although it is applied to larger L2 blocks here, as opposed to the tiny L1 blocks of previous work [4]). The third bar, **srp**, corresponds to the published scheduled region prefetching scheme, with the two enhancements described in Section 4.1. The results show large speedups for **srp** compared to **no-pf** and **ldv** for the applications that have spatial locality (*art*, *equake*, *fma3d*, *mgrid*, and *parser*), and virtually no performance losses for other applications. In contrast, unscheduled **ldv** prefetching incurs small but noticeable performance losses on two benchmarks (*art* and *vpr*), primarily due to the interference of the prefetches with demand misses.

Only *ammp* and *facerec* show higher performance with **ldv** than with **srp**, due to significantly higher prefetch accuracies. This behavior can be understood by referring to Figure 2. Although these applications exhibit some spatial locality, the population-count statistic shows that this locality is concentrated in a small fraction of the regions. In addition, referring to the "L-Cor" line in Figure 2, this locality is very strongly correlated with the region's previous density vector. Thus the simple **ldv** scheme has a prefetch accuracy of 83% for *ammp* and 89% for *facerec*, versus 4% and 52% respectively for **srp**. This large

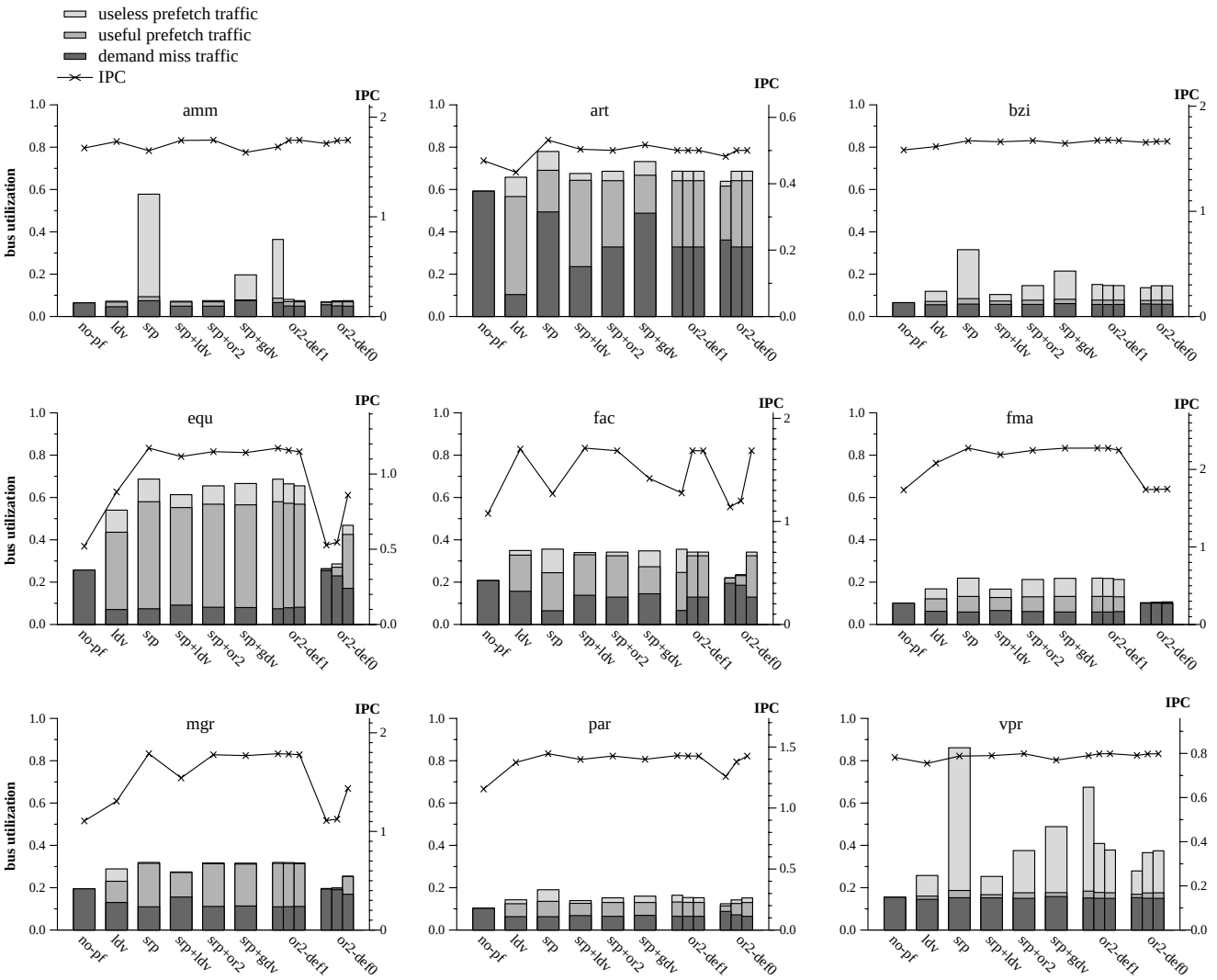

Figure 4. Performance results

improvement in accuracy, coupled with high absolute accuracy, overwhelms the benefit of **srp**'s scheduling and placement optimizations for these two benchmarks.

The bus utilization bars in Figure 4 illustrate the issue we seek to address: **srp**'s performance improvements come at the expense of dramatic increases in memory traffic, particularly on the low-locality benchmarks for which prefetching accuracy is poor. In the worst case, *ammp*, traffic increases by over 800%. Due to **srp**'s channel scheduling, even this dramatic traffic increase leads to a mere 2% performance loss. No other benchmark loses performance from **srp**. The goal of our filtering schemes is to minimize this unnecessary traffic increase without impacting the performance gain.

In the fourth bar from the left, labeled **srp**+**ldv**, we show the result of filtering **srp** prefetching using the last local density vector (see Equation 6 in Section 4.2). This **ldv** filter successfully reduces traffic, eliminating 70% of unnecessary prefetches. In addition, six of the nine benchmarks perform within 5% of the unfiltered **srp** scheme. *Mgrid* suffers the most from the **ldv** filter, slowing by 14%. Figure 2 shows why: *mgrid*'s spatial locality is moderately high, but is poorly predicted by a region's previous density vector. Thus the coverage of **srp** with the **ldv** filter is low, reducing performance. The **ldv** filter gives **srp**+**ldv** the same accuracy benefits on *ammp* and *facerec* as the unscheduled **ldv** scheme, improving performance over **srp** by 6% and 35%, respectively.

The **or2** filter, described by Equation 7 in Section 4.2, seeks to reduce further the performance gap between filtered and unfiltered **srp** by prefetching any block in either of the two previous local density vectors. The **srp+or2** bar

	amm	bzi	vpr	art	equ	fac	fma	mgr	par	HM
No prefetching	1.69	1.58	0.78	0.47	0.52	1.08	1.74	1.11	1.16	0.92
SRP	-1.7%	5.5%	0.8%	13.2%	125.4%	17.8%	31.1%	61.7%	25.1%	27.9%
LDV	3.8%	2.0%	-3.5%	-7.5%	69.4%	58.0%	19.9%	18.1%	18.8%	15.5%
SRP+OR2	4.6%	5.6%	2.2%	6.6%	121.0%	56.5%	29.5%	60.7%	23.4%	29.6%
SRP+GDV	-2.6%	3.9%	-1.5%	10.2%	119.8%	31.5%	31.1%	59.9%	21.0%	26.9%

Table 1: Performance effects of prefetching policies

in Figure 4 shows the result: it improves the performance of *mgrid* by 15%, *equake* by 3%, and *fma3d* by 3% over **srp**+**ldv**—bringing these three benchmarks within 2% of the performance of unfiltered **srp**—with negligible performance impact on the other benchmarks. While total traffic increases significantly relative to **no-pf** for a few low-accuracy benchmarks (40% for *bzip2* and 47% for *vpr*), the **or2** filter still eliminates a significant number of **srp**'s useless prefetches (a 52% reduction on average).

Our final filter, **srp+gdv**, exploits global rather than local density-vector correlation by using the last global density vector as a filter, as described in Equation 9 of Section 4.2. Overall performance is similar to **srp+ldv**, and slightly degraded relative to unfiltered **srp** and **srp+or2**. However, the performance impact is not uniform across the benchmarks: relative to **srp+ldv**, *ammp* and *facerec* degrade significantly, while *mgrid* and *fma3d* improve. These results are consistent with the L-Cor and G-Cor measurements for these benchmarks in Figure 2. *Equake* and *art* also improve slightly, in spite of having lower global than local correlations in Figure 2; apparently the inaccuracies of the **gdv** filter lead to more aggressive prefetching, which tends to improve performance with **srp**. The **gdv** filter also reduces traffic, though to a lesser extent than the **ldv** and **or2** filters, eliminating 30% of unnecessary prefetches on average. The key advantage of the **gdv** filter is that it requires minimal storage: only one density vector must be maintained in addition to those currently under construction.

Table 1 summarizes our performance results, showing the base non-prefetching IPCs for each benchmark and the harmonic mean IPC over all benchmarks. We also show the percentage change in IPC for **srp**, unfiltered and with the **ldv**, **or2**, and **gdv** filters. We see that **or2** has the best overall performance: slightly better than **srp** and **srp+gdv**, and 14% higher than **srp+ldv**. In addition, **or2** produces significantly less traffic than **srp**, and is the only policy for which there are no performance losses in any of our benchmarks. Thus, given infinite density-vector storage, the **or2** filter provides the best balance between successful prefetches and traffic increases.

4.4. Effects of finite density vector storage

Our results thus far have assumed infinite storage for recording density-vector history. To measure the impact of finite storage, we simulated the performance of the **or2** scheme using a four-way set-associative cache to store completed density vectors. We analyze three cache sizes that represent differing relationships between density-vector coverage and data-set sizes. The smallest cache has 2K entries; including two density vectors per entry (as required by the **or2** filter) plus tag overhead for a 48-bit physical address space, this cache requires approximately 39K bytes of storage. With our region size of 4K bytes, this cache can track the behavior of up to 8MB of memory. This coverage is less than the data-set sizes of all our benchmarks except *bzip* and *art*. The mid-size cache has 8K entries, requiring 153KB of storage and covering 32MB of memory—adequate for all our benchmarks except *equake*, *fma3d*, and *mgrid*. The largest cache has 32K entries, requiring 604K of storage and covering 128MB of memory. At roughly half the size of the L2 cache itself, this density-vector cache size is impractical, but its coverage is greater than the data set size for all our benchmarks.

We portray the results in the right-most two sets of bars in Figure 4. Each group of three bars shows the bus utilization and performance of the three density-vector cache sizes, from smallest to largest. The groups differ in their default policy for accesses that miss in the density-vector cache. The first group, **or2-def1**, uses a filter of all 1s (prefetch everything), while the second, **or2-def0**, uses a filter of all 0s (prefetch nothing).

The smallest density-vector caches have significant numbers of misses, so traffic and performance are largely determined by the default policy: results are similar to unfiltered **srp** for **or2-def1**, and to **no-pf** for **or2-def0**. As the cache grows, traffic and performance converge toward the infinite-storage **or2** values. Unfortunately, this convergence is not gradual: most benchmarks see little benefit from filtering (i.e., traffic reduction from **or2-def1**, or performance gain from **or2-def0**) until the density-vector cache is capable of tracking the entire data set.

For *equake*, *fma3d*, and *mgrid*, **or2-def0** fails to reach the performance of infinite-storage **or2** even with the largest density-vector cache size due to conflict misses in the

density-vector cache. These conflicts are exacerbated by the higher data-cache miss rate induced by the default no-prefetching strategy, which leads to a dramatic increase in density-vector cache accesses. In the most extreme case, *equake*, using the **or2-def0** policy rather than **or2-def1** leads to a the density-vector cache has a 12 times higher miss rate and 58 times more misses in the 32K-entry density-vector cache.

The impact of an overly small density-vector cache might be reduced by using an adaptive scheme to select an appropriate default policy dynamically; we leave this enhancement for future work.

5. Conclusions

The trade-off between speculative fetching to reduce cache misses and the increase in traffic caused by mistaken fetches is critical. In earlier work, we showed that aggressive prefetching can be made profitable by using novel mechanisms to avoid the overheads associated with bad prefetches. We incorporated these mechanisms into scheduled region prefetching, a design that effectively addresses memory-system stalls for applications with spatial locality. However, in many cases, that design generated copious amounts of unnecessary memory traffic. In this paper, we show how to retain the considerable performance benefits of scheduled region prefetching while removing the majority of the unnecessary traffic produced.

First, we showed that many regions have significant inter-epoch correlation. Then, for nine of the SPEC2000 benchmarks, we showed that we could reduce useless traffic by 70%, while improving mean performance by 1.3%, by adding the **or2** filter in addition to scheduled region prefetching. The benchmark with the worst performance loss was *art* (6%,) while the benchmark with the biggest gain was *facerec* (33%).

Unfortunately, the **or2** scheme requires significant storage to maintain two previous density vectors for each memory region. Caching vectors for only a portion of the memory space appears to have limited effectiveness. Given this practical constraint, the **gdv** filter, which requires maintaining only the last globally recorded density vector, may be more attractive. This scheme provides overall performance within 3% of the **or2** filter, and reduces useless traffic by 30% relative to unfiltered **srp**.

Memory-intensive benchmarks see one of three effects from our additions to **srp**. First, if they have little locality, our filters eliminate most of the useless prefetches. Second, if they have locality, but have already been brought close to the performance of a perfect level-two cache by **srp**, the filtering does not improve performance, but may reduce traffic. Finally, for applications that have locality and are still slowed by L2 misses, the filtering may improve their performance by eliminating bad prefetches, thus allowing more good prefetches to complete.

In previous work, we showed how to eliminate the memory performance gap—for most applications—if spatial locality exists. The price was an enormous increase in memory channel traffic. In this paper, we have shown how to provide the same performance benefits without the traffic explosion. The remaining—and extremely difficult—challenge is to extend this prefetch architecture to issue effective prefetches for those memory-intensive applications that have little spatial locality.

Acknowledgments

This work was supported by the National Science Foundation under Grants No. CCR-9734026 and CCR-9985109, by gifts from Intel and IBM, and an equipment grant from Compaq. Thanks to Rajani Parthasarathy for her help in generating the final results.

References

[1] Doug Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface*. PhD thesis, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706-1685, Dec. 1998.

[2] G. C. Driscoll, T. R. Puzak, H. E. Sachar, and R. D.Villani. Staging length table - a means of minimizing cache memory misses using variable length cache lines. *IBM Technical Disclosure Bulletin*, 25:1285, Aug. 1982. http://www.patents.ibm.com/tdbs/tdb?o=82A%2061160.

[3] Teresa L. Johnson and Wen-mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. *Proc. 24th Int'l Symp. on Computer Architecture*, pages 315–326, June 1997.

[4] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. *Proc. 25th Annual Int'l Symp. on Computer Architecture*, July 1998.

[5] Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. Reducing DRAM latencies with an integrated memory hierarchy design. *Proc. 7th Int'l Symp. on High-Performance Computer Architecture*, Jan. 2001.

[6] O. Temam and Y. Jegou. Using virtual lines to enhance locality exploitation. *Proc. 1994 Int'l Conf. on Supercomputing*, pages 344–353, July 1994.

[7] Peter Van Vleet, Eric Anderson, Linsay Brown, Jean-Loup Baer, and Anna Karlin. Pursuing the performance potential of dynamic cache line sizes. *Proc. 1999 Int'l Conf. on Computer Design*, pages 528–537, Oct. 1999.