TAP: Taxonomy for Adaptive Prefetching

Jayaram Bobba Michelle Moravan Umair Saeed

University of Wisconsin, Madison

Abstract

Data prefetching is an important technique for overcoming the increasing gap between the processor and memory speeds. Traditonal metrics for prefetching have been shown to be ineffective and misleading. Prefetch taxonomies provide interesting insights into the performance of prefetch algorithms. In this report, we present an analysis of two prefetch algorithms using a prefetch classification taxonomy PTMT. We also come up with the drawbacks of PTMT and propose an alternative taxonomy TAP. An adaptive tagged prefetch algorithm (A-TAG) that uses TAP is also proposed and analyzed.

Keywords: Data Prefetching, Prefetch Taxonomy, Tagged Prefetching, Sequential Prefetching, CDP

1 Introduction

Memory has traditionally been a bottleneck for achieving higher levels of ILP. The advent of out-of-order superscalar architectures alleviated this bottleneck to an extent by allowing computation to hide the memory access latency. But with the increasing difference between processor and memory speeds, memory is once again turning out to be a significant barrier to achieving higher speed-ups. Cache hierarchies have been designed to decrease the average memory access latency. However, as DRAM access latencies reach 128 processor cycles, an access to memory cannot be completely hidden by the processor. Amdahl's Law indicates that these cache misses limit the gains obtained by improving the processor core. Data prefetching is one techinque that tries to remove these misses by anticipating future data accesses and moving them closer to the processor in the cache hierarchies.

A good data prefetching technique should have the following properties. First, it should issue useful prefetches i.e. prefetch data that is likely to be used by the processor in the near future. Second, the prefetches should be timely. The data should not be brought in too early lest it displace some data that is required in the near future. A prefetch that does not hide a significant portion of memory access latency is also not going to be very useful. Finally, as a consequence of the first two properties, the prefetches should not lead to cache pollution, displacing data that is going to be used in near future with data that is prefetched but not accessed.

Traditional metrics for measuring the efficiency of prefetch algorithms have been shown to be inadequate and sometimes misleading. Srinivasan et. al [1] provide an excellent illustration of the deficiencies of these metrics. They introduce a taxonomy for classifying prefetches, PTMT, that does a better job of measuring the efficiency of prefetches. It also provides useful insights into the working of prefetch algorithms. However, their taxonomy has some drawbacks which we discuss later on in the report.

The following summarizes the contributions of our project. We

- Present a brief overview of some of the prefetching algorithms existing in literature.
- Discuss the prefetch taxonomy PTMT and use it to analyze two existing prefetch algorithms.
- Highlight the drawbacks of PTMT and come up with our taxonomy, TAP, which we consider more informative and simpler.
- Propose and evaluate an adaptive prefetch algorithm (A-TAG), A-TAG.

The rest of the report is organized as follows: In section 2, we talk of some of the prefetch algorithms proposed in literature. Section 3 discusses PTMT. Section 4 discusses two prefetch algorithms in the context of PTMT. Section 5 motivates and presents our new taxonomy, TAP. The next section discusses a simple application of this taxonomy for the development of an adaptive prefetch algorithm. Finally, section 7 presents the conclusions and future work.

2 Prefetch Algorithms

Existing data prefetch techniques can be classified into two main categories - software based and hardware based. Software prefetching is done via a specific *prefetch* instruction which specifies the address of the data word to be brought into the cache ([2], [3]). This address is passed to the memory system without waiting for a response and the processor continues execution. This way data can be brought into different cache levels ahead of its actual access in the code. However, for these techniques to be efficient a detailed knowledge of the data flow is needed, which in turn implies a complex and powerful compiler. The advantage of software prefetching is that the prefetch instruction is issued early enough to ensure that the data will already be present when it is accessed, hence hiding the complete latency. Also, fewer useless prefetches are issued since the compiler can use its knowledge about the program data flow in the code. This reduces the memory bandwidth requirements imposed by the prefetch mechanism.

Hardware techniques, on the other hand, rely on dynamic memory access information for predicting future data accesses. This speculation results in greater number of overall prefetches as compared with software prefetch mechanisms. The advantage of hardware techniques is that they incur little overhead on the processor itself and a powerful compiler is not needed. Several hardware based techniques have been proposed in literature. Vanderwiel and Lilja [4] provide an excellent survey of prefetch algorithms. We discuss three hardware prefetch algorithms - tagged prefetch, prefetch with arbitrary strides [4], and content data prefetching [5]. These algorithms were implemented and studied for the project. We give a brief summary of each of these below.

2.1 Tagged Prefetch (TAG)

Tagged prefetching is based on the simple sequential prefetching schemes which are variations on the one-block lookahead (OBL) schemes. A prefetch for a block $b + 1$ is initiated when block *b* is accessed. A *first-use* bit is associated with each cache block to indicate whether the block was demand fetched or is a prefetched block being referenced for the first time. When a block is demand fetched or prefetched, its *first-use* bit is set to 1. This bit is checked whenever the block is accessed. If it is found to be 1, it means that this block is being accessed for the first time, so the *first-use* bit is set to 0, and a prefetch for block *b + 1* is issued. This performs quite well for sequential streams of data. However, for a sequential stream resulting from a tight loop, the prefetch may not be initiated sufficiently in advance to avoid cache miss stalls. To overcome this deficiency, it is possible to increase the number of blocks which are prefetched. This number can be increased from 1 to K, and is called the degree of prefetch. This means that when a block *b* is accessed for the first time, the cache is checked whether blocks *b + 1, ... , b + k* exist in the cache. The missing blocks are then prefetched from memory.

2.2 Prefetch with arbitrary strides (STRIDE)

Strided prefetch prefetches constant stride array references which originate from looping structures. Special hardware logic is implemented to monitor the processor's address referencing pattern to detect the constant stride. This is done by comparing successive memory addresses issued by individual load or store instructions. If a memory instruction *m* references addresses $< a_1, a_2, a_3 >$ during three successive iterations, a prefetch would be initiated on the third iteration if $\delta = a_2 - a_1 = a_3 - a_2$. The prefetch address would be $a_3 + \delta$.

Strided prefetch requires the last memory reference to be stored along with the last stride for each instruction that accesses memory. This necessitates support from hardware in the form of a lookup table indexed by the PC. For each memory instruction, the address of the previous memory reference, the value of the stride, and the state are saved. Since it is impractical to store all the memory references in a program, only the most recent ones are stored in a structure known as the *reference prediction table* (RPT) [4]. When a memory instruction is executed for the first time, an entry for it is made in the RPT with the state set to *'initial'*. If it is executed again a stride value is calculated by subtracting the previous address stored in the RPT from the current address. This way the value of the stride is re-calculated on each reference and that value is used to predict the next prefetch.

2.3 Content-Directed Data Prefetching

Tagged and strided work for regular memory accesses, but their performance suffers in pointer intensive workloads. Content-Directed Data Prefetching (CDP) [5] was proposed to improved the performance of these pointer intensive workloads. When data is demand fetched from the memory, CDP borrows techniques from conservative garbage collection to check this data for a likely address. A word which has been identified as a likely address is converted from a virtual address to a physical one, and a prefetch request is issued using the physical address. As the prefetch returns from memory, its contents are also examined for likely addresses. This process then recurses. The prefetcher exploits the idea that if a pointer is being loaded from memory, that pointer will likely be used as a load address in the near future. The difficulty of this scheme lies in identifying a virtual address in a stream containing addresses, data, and random bits. Several heuristics have been proposed by Cooksey et al. [5]. The simplest one uses the fact that most virtual data addresses tend to share common high-order bits. Thus all data values found in a structure that share a common base address can be assumed to be pointers and can thus be used as prefetch addresses.

3 Prefetch Taxonomy

The proliferation of prefetch algorithms, touched upon above, necessitates a common set of metrics for their comparison. It is common knowledge [1] [6] traditionally, evaluating the quality of a prefetch algorithm has involved three terms: the number of good prefetches, G, the number of bad prefetches, B, and the total number of memory accesses, M. A prefetch is classified as good if the program uses the associated data before replacing the prefetched line in the cache. Any other prefetch falls into the bad category.

The two most common metrics, *coverage* and *accuracy*, are derived from these terms as follows.

$$C = \frac{G}{M} \tag{1}$$

$$A = \frac{G}{G + B} \tag{2}$$

High accuracy and coverage percentages are meant to indicate effective prefetching. However, [1] shows it trivial to construct a sequence of cache accesses and prefetches that, while achieving perfect accuracy and coverage, fails to decrease the number of misses. Furthermore, any unused prefetch incurs excess traffic. Thus the seeming neutrality of such an example - no fewer misses, but no more - belies the bandwidth cost of such useless prefetches. Furthermore, it is even possible for prefetching to increase the number of misses, either due to extremely bad prefetches, or as a result of the increased speculation allowed by reduced memory stalls. Coverage may not reflect these effects, depending on how M is calculated [6].

To address these deficiencies, [1] proposes the *prefetch traffic and miss taxonomy* (PTMT), which considers the performance effects of both misses and traffic, and attributes these to specific prefetches. To do so, Srinivasan et al. consider the next event on both the prefetched line, x, and the line that it replaces in the cache, y. The effects on each of these lines, and thus the ultimate consequences of any given prefetch, depend on what would have happened had no prefetching occurred. Thus implementation of PTMT requires simultaneous simulation of both a cache affected by some prefetch algorithm, *pf-cache*, and a cache affected by no such changes, the conventional cache, or *conv-cache*.

There are three distinct combinations of what happens to x in the pf-cache and the conv-cache. First, a hit might occur in both. The prefetch thus incurred additional traffic by bringing in a line that would have been there anyway, and no misses are avoided. Second, a hit may occur in the prefetch line while a miss occurs in the conv-cache. This case eliminates a miss without inducing any additional traffic, as the line would have been demand fetched regardless. Finally, if the prefetched line is replaced before it is accessed, the conv-cache event no longer matters. The

Table 1: Cost in Traffic and Misses for the Nine (x, y) Case Pairs

	pf-cache **outcomes**		*conv-cache* **outcomes**		**Extra**	
Cases	*x (prefetched)*	*y (replaced)*	*x (prefetched)*	*y (replaced)*	**Traffic**	**Misses**
1	hit	miss	hit	hit	2	1
2	hit	prefetched	hit	hit	1	0
3	hit	don't care	hit	replaced	1	0
4	hit	miss	miss	hit	1	0
5	hit	prefetched	miss	hit	0	-1
6	hit	don't care	miss	replaced	0	-1
7	replaced	miss	don't care	hit	2	1
8	replaced	prefetched	don't care	hit	1	0
9	replaced	don't care	don't care	replaced	1	0

line was prefetched, at the cost of an additional line of traffic, for nothing: it saved no misses.

The authors perform a similar analysis for y, which also has three combinations. First, a miss might occur in the pf-cache while a hit occurs in the conv-cache. This means the prefetch displaced a needed line, thus causing additional traffic and an additional miss. Second, the replaced line y might itself be prefetched, while it hits in the conv-cache. Third, y might be replaced in the conv-cache, making its future in the pf-cache irrelevant. In both of the latter two cases, the effects of prefetching are effectively nullified, as there is no additional traffic, but neither is there a change in the number of misses.

Finally, [1] puts all combinations of the above two sets of cases together to form nine cases which, they assert, completely describe the possible effects of any given prefetch. Table 1, duplicated from [1], summarizes this data.

However, Table 1 is only complete for direct-mapped caches. By introducing associativity, one more case must be taken into account, events that occur on z, a demand-fetched line not directly affected by any prefetch. A reference to z could lead to four combinations of events on the pf-cache and conv-cache. First, it might hit in both, or it might miss in both. Neither of these situations lead to additional traffic or misses. Second, z could hit in the pf-cache but miss in the conv-cache. [1] proves this can never occur. Finally, a reference to z might miss in the pf-cache and hit in the conv-cache, generating an extra line of traffic and an extra miss to demand fetch it back into the pf-cache. This occurs because a prefetch of line x that replaces line y can actually affect our third line z. Introduction of x alters the least recently used ordering tracked for replacement. This may lower the priority of z sufficiently so that it is eventually replaced in the pf-cache when it would not have been in the conv-cache. PTMT classifies this as Case 10.

To summarize, [1] divides prefetches into four categories. Useful prefetches, Cases 5 and 6, induce no extra traffic and reduce the number of misses by one. Useless prefetches, Cases 2, 3, 4, 8, and 9, do not change the number of misses, but do cost an extra line of bandwidth. Polluting prefetches, Cases 1 and 7, actually cause an additional miss at the cost of two extra lines of traffic. Finally there is the prefetch side-effect, Case 10, described above.

4 Analysis of TAG and STRIDE

In this section, we analyse two prefetch algorithms - TAG and STRIDE using PTMT. Since our implementation of CDP performs very poorly in that issues very few useful prefetches, we do not discuss it in the report.

SimpleScalar [7] was modified to implement PTMT and the prefetch algorithms. We retained the default simple-scalar parameters in most cases; Table 2 shows the parameters that we changed from the defaults. A few experiments revealed that such variation in IFQ and RUU parameters has little relative effect on the results; nevertheless, we present data for these more realistic values. Additionally, implementation of strided prefetch involves a table that tracks when to make a prefetch. We chose 128 lines as the size for this *reference prediction table*, but would have liked to explore other sizes. Use of 512 lines in [6] suggests our initial hunch may have been too miserly. Finally, SimpleScalar makes certain assumptions about the working of caches. These assumptions could lead to overly optimistic results in terms of performance. Perez et al. [8] present some details on the effects of these assumptions on performance results reported by SimpleScalar.

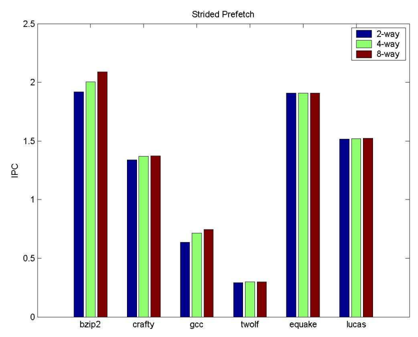
We use six SPEC CPU2000 benchmarks - 4 integer and 2 Floating Point, for our study. For faster simulations and more accurate results, we used the 100 million single SimPoint values for each benchmark [10]. After performing some basic validation experiments using SimPoints and

Table 2: Non-Default SimpleScalar Parameters

Parameter	**Value**
mem:lat	128 16
fetch:ifqsize	16
ruu:size	64

running a few benchmarks for different simulation lengths, we concluded that 100M instructions is sufficient to exercise the different configurations for L2 caches.

The performance of prefetch algorithms were studied while varying the set asscociativities between 2, 4 and 8. We observe that for a given prefetch algorithm, IPC always increases or remains constant as set-associativity increases, for any given benchmark. Figure 1 shows this for strided prefetch, but the graphs for no prefetching and tagged prefetch are similar, and can be viewed in the Appendix. (Whenever a representative graph is chosen for presentation here, graphs for all other parameters are made available in the Appendix.)



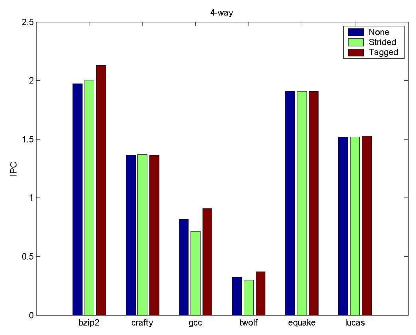


Figure 1: IPC for varying set associativities

Figure 2: IPC for various prefetch algorithms

Similarly, for any constant set-associativity, use of tagged prefetch generally retains or improves IPC. The sole exception is crafty at 4-way set-associativity, which shows a minuscule decrease in IPC. On the other hand, the performance of strided prefetch varies widely. It sometimes significantly degrades performance, as in the case of twolf, but other times it noticeably improves it, as with bzip2. Overall strided prefetch seems to have little effect. Figure 2 shows these trends for 4-way set-associativity.

Exploring these trends in terms of misses shows similar results. As expected, misses generally decrease as associativity increases, though occasionally there is little or no change. This most likely means associativity has grown large enough to eliminate most conflict misses. Figure 3, a representative sample, shows that some benchmarks, such as lucas, are quite sensitive to associativity, whereas others, namely bzip2, are not. Note that equake has so few misses it is difficult to discern trends; here it is better to refer to IPC.

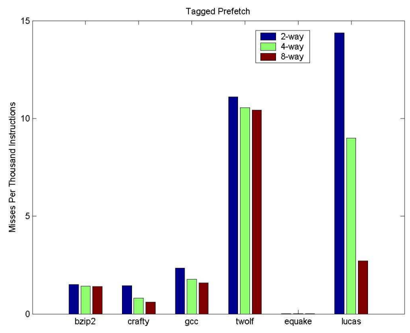


Figure 3: Misses/1000 instrns for varying associativity

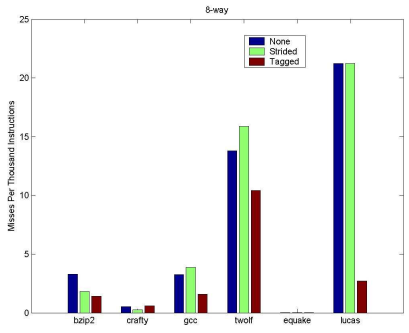


Figure 4: Misses/1000 instrns for varying prefetch algorithms

Miss results for varying prefetch algorithms track those for IPC. In Figure 4, we see that strided prefetching almost always maintains or increases the number of misses. The only exceptions occur with bzip2 and crafty in conjunction with the two higher associativities. In contrast, but as above, tagged prefetch almost always causes a large decrease in misses, with the one exception of crafty, which sports a slight increase. Finally we note that lucas seems less sensitive to the negative effects of strided prefetch than other benchmarks. A major drawback with PTMT, not mentioned in [1], is the problem of unclassified prefetches. Since the taxonomy requires observation of the next event on both the prefetched and replaced lines, if one or both is not accessed again, that prefetch will never be classified. Figures 5 and 6 add a fifth category, ’unclassified’, to the four types of prefetches reviewed in Section 3, and reveal the surprising dominance of this previously invisible class of prefetches. Our data illustrates that for all benchmarks and associativities, unclassified prefetches uniformly count for at least 50% of all prefetches.

On the other hand, parts of Figures 5 and 6 corroborate [1]. For instance (ignoring unclassified prefetches), tagged prefetch, which performs better than strided, has visible dark blue sections, which correspond to useful prefetches. In contrast, the strided bars comprise almost entirely light blue and orange sections, which designate useless and (harmful) Case 10 prefetches, respectively. As an aside, it is also interesting to note that strided, which performs worse, issues about 50% more prefetches per thousand instructions than tagged. The poor prefetch quality can perhaps be attributed to the release of tentative prefetches before stride confirmation.

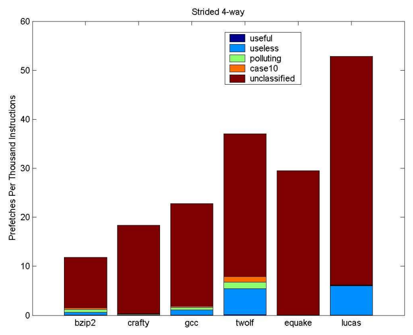


Figure 5: Prefetches/1000 instructions by category, a.

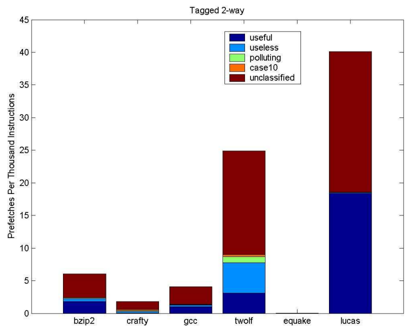


Figure 6: Prefetches/1000 instructions by category, b.

These insights suggest benefit from examining the case distribution in greater detail. Figures 7 and 8 normalize the distribution to the number of classified prefetches. A cursory comparison affirms PTMT by explaining why tagged performs better than strided: the strided graphs, dominated by the reds, oranges, and dark blues that indicate harmful or useless prefetches, contrast strikingly with the tagged graphs, which replace the harmful prefetches with yellow useful prefetches and a few green and light blue useless prefetches. Note that as equake classifies very few prefetches (none where the data is omitted!), the normalized values may not be statistically significant. However, a closer examination introduces some doubt. For instance, in the case of crafty, where strided performs better than tagged, the picture looks much the same. Scrutiny of these figures alone would never suggest this unusual conclusion. This inaccuracy may result from the large number of unclassified prefetches. Along the same vein, PTMT also fails to explain why there is little performance difference between tagged and strided performance for lucas. Like crafty, the figures would suggest a dramatic increase for tagged.

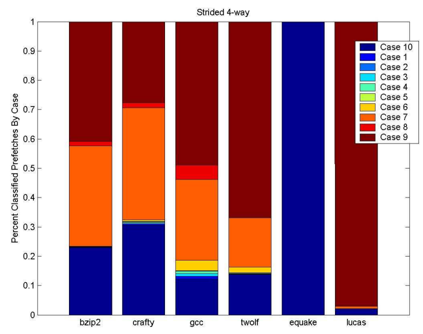


Figure 7: Case distribution, a.

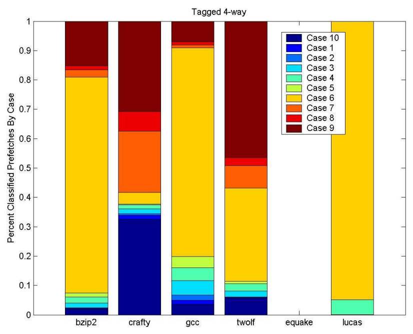


Figure 8: Case distribution, b.

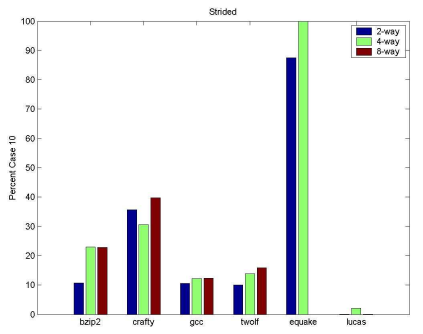


Figure 9: Percent Case 10, a.

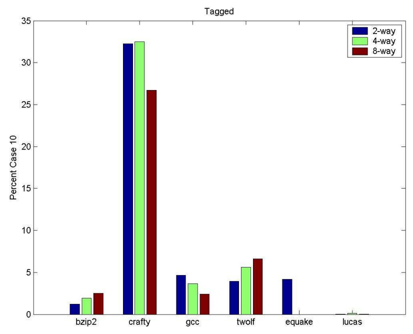


Figure 10: Percent Case 10, b.

We conclude by exploring the impact of Case 10. Srinivasan et al. assert that it is insignificant. Figures 9 and 10 show how the percentage of Case 10 prefetches varies with associativity. Except for lucas, our results show that it is almost never negligible. Furthermore, larger associativities tend to have higher percentages than 2-way; this is important because [1] only reports on PTMT for 2-way set associative caches. Case 10 events do not present any insight into the prefetches leading to them and thus limit the usefulness of PTMT.

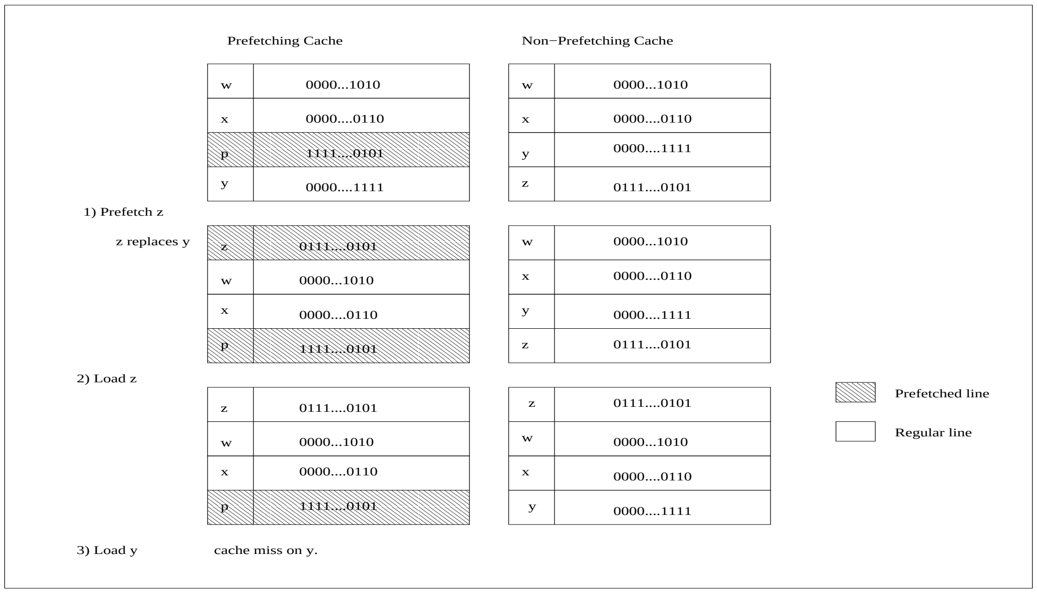


Figure 11: Motivation for TAP

5 Taxonomy for Adaptive Prefetching (TAP)

In this section, we present a new taxonomy for classifying prefetches. First, we present the motivation behind TAP and look at some of the drawbacks of PTMT. Then we present TAP and some issues regarding its implemention in hardware.

5.1 Motivation

An analysis of prefetches is generally based on observing *prefetch events*. A *prefetch event* is a cache hit or miss that causes a difference in the number of misses or memory traffic when compared to a cache where no prefetches are issued. *Good prefetch events* decrease the number of misses or traffic while *bad prefetch events* increase the number of misses or traffic. A prefetch that leads to *good prefetch events* can be considered *good*, while a prefetch that leads to *bad prefetch events* can be considered *bad*. However, notice that a single prefetch could lead to multiple *prefetch events* some of which could be *good* and some *bad*. The question of attributing a *prefetch event* to a prefetch is also non-trivial. This makes the task of classifying prefetches challenging.

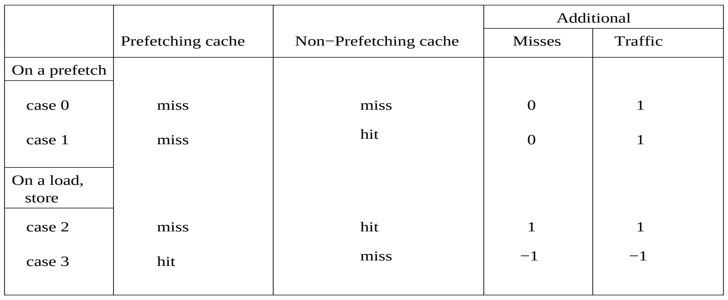
PTMT classifies prefetches by looking for *prefetch events* on the prefetched line and the line it replaces. Thus a prefetch is held responsible for all events on the prefetched line and the replaced line. However, we believe this assumption is rather simplistic and sometimes leads to erroneous conclusions. In Figure 11, we present an example that shows how we could draw misleading insights as a result of this assumption. Since the prefetch of line *z* leads to the replacement of line *y*, PTMT associates the cache miss on *y* with the prefetch of *z*. However, note that in the absence of the prefetched line *p*, we would not have had a cache miss on *y*. We would also not have prefetched *z* since it would have already been present in the cache.

In summary, we present four drawbacks of PTMT. First, in certain cases it wrongly attributes *prefetch events* to prefetches. Second, it is incomplete. As discussed in the previous section, a large number of prefetches do not get included in the taxonomy. Third, Case 10 events in PTMT do not provide useful insights into the prefetch performance. And finally, PTMT cannot be implemented in hardware and so cannot be used for dynamically adapting prefetch behaviour.

This leads us to TAP, where instead of classifying prefetches, we classify *prefetch events*. We then try to associate these events with a single prefetch or multiple prefetches. Intuitively, while PTMT makes a prefetch its reference point and tries to associate it with future *prefetch events*, TAP makes a *prefetch event* its reference point and then looks back to decide which prefetch(es) was/were responsible for it. This gives us more flexibility and allows us to implement TAP in hardware.

5.2 TAP

Figure 12 presents TAP. Cases 0 and 1 classify *prefetch events* that occur when a prefetch request is sent to the cache. Cases 2 and 3 occur when a normal reference (i.e a load or a store) is sent to the cache. Case 0 relates to prefetches that attempt to avoid *compulsory* misses. Cases 1 and 2 are the *bad prefetch events* caused by cache pollution. The easiest and most intuitive way



	Prefetching cache	Non−Prefetching cache	Additional	
			Misses	Traffic
On a prefetch				
case 0	miss	miss	0	1
case 1	miss	hit	0	1
On a load, store				
case 2	miss	hit	1	1
case 3	hit	miss	−1	−1

Figure 12: TAP

to attribute prefetches to these events would be to hold the LRU prefetched line responsible for them. More sophisticated approaches for attributing responsibility can be designed. Case 4 is the *good prefetch event*. It can be directly attributed to the prefetched line on which the cache request hits.

5.3 Hardware Support

To enable hardware implementation of TAP, we propose the use of a feedback cache using structures similar to the ones proposed by Bhavesh et al. [9]. For any given set, a feedback cache maintains information about the lines evicted due to prefetches. At any instant, these lines, together with the non-prefetched lines, give us a snap-shot of a non-prefetching cache. Bhavesh et al. [9] give more implementation details regarding the maintenance of this evicted line information. A k-way set associative feedback cache would need an additional k evicted tag entries per set. We believe that this is not too great a demand on the cache hardware considering increasing on-chip transistor counts. Alameldeen and Wood [11] use a similar structure to support adaptive cache compression.

Using a feedback cache, we could identify whether a cache request hits or misses both in the prefetch and non-prefetch cases. Thus we could use the taxonomy *on-line* to classify prefetches. This information can be used to design adaptive prefetch algorithms that can selectively enhance or throttle prefetching as program behaviour changes. Most of the adaptive prefetch algorithms

```
for every N prefetch events do
    if Number of good prefetch events $<$ T $\times$ (Number of bad prefetch events) then
        degree of prefetch, K = 1
    else
        degree of prefetch, K = 2
    end if
end for
```

Figure 13: A-TAG

proposed use indirect prefetch performance indicators to adapt their parameters. TAP provides a mechanism to directly use prefetch performance information to dynamically change the prefetch algorithm.

6 Adaptive TAG (ATAG)

In this section, we present a simple adaptive prefetch algorithm using TAP. We also present the results of running this algorithm for various benchmarks.

6.1 Algorithm

Tagged prefetch works by fetching the next sequential cache line whenever a given line is first accessed in the cache. As indicated in Section 2, its also possible to prefetch many lines ahead of the current accessed line. Adaptive tagged prefetch algorithms work by varying this parameter K, called the degree of prefetch. Ando and Knowles [12] present an adaptive tagged prefetch algorithm that works by varying the degree of prefetch based on the cache behavior with respect to the previous prefetches. We will not compare the results of our work with their algorithm.

Our Adaptive Tagged(A-TAG) Prefetch algorithm simply varies the degree of prefetch dynamically, using TAP. In our study, we set the degree of prefetch to be either 1 or 2. The algorithm begins with a degree of prefetch $K = 1$. If the cache observes many *good prefetch events*, it increases the degree of prefetch to 2. Converselty, if it observes many *bad prefetch events*, it

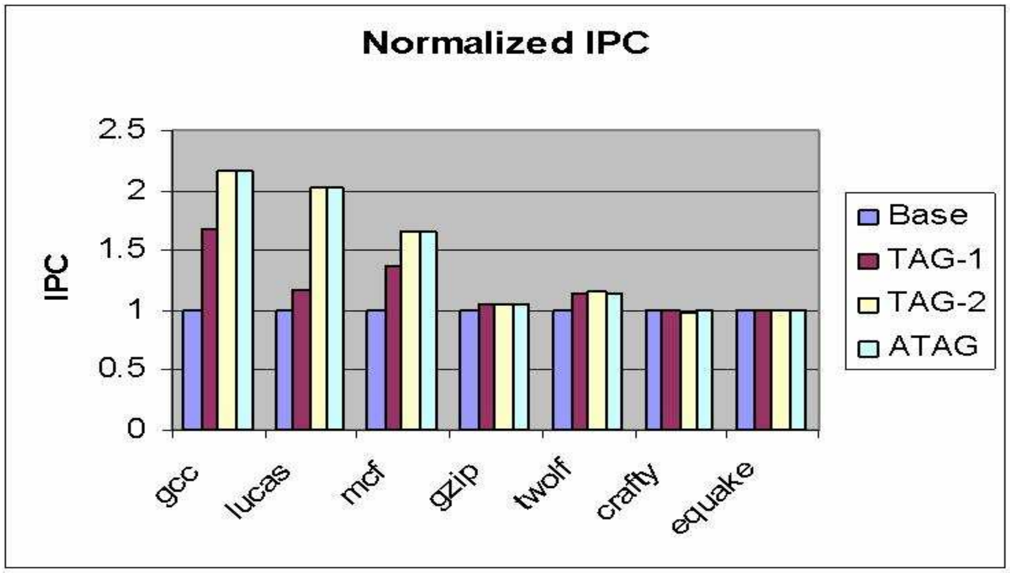


Figure 14: Comparison of normalized IPC

throttles prefetch by setting K back to 1. The algorithm is summarized in Figure 13. In our study, N was set to 200 and T was set to 3. We do not study the performance of A-TAG with different configurations, as the goal was to come up with an algorithm that uses TAP effectively, not necessarily the best algorithm.

6.2 Results

We present the results for running A-TAG on various SPEC2000 CPU benchmarks. In Figure 14, we compare the IPC's of various simulations run with and without prefetching. The first column for a benchmark presents the base case where we run the simulation without any prefetching. The next column (TAG-1) presents the IPC for tagged prefetching using a degree of prefetching $K = 1$. The third column (TAG-2) is for tagged prefetching with $K = 2$. The final column is for A-TAG. We notice that A-TAG adapts to the best degree of prefetching in all the cases. For the benchmark *twolf*, TAG-2 acheives a slightly higher IPC, but when we also consider Figure 15, we realize that this comes at the cost of significant additional traffic. In conclusion, where tagged

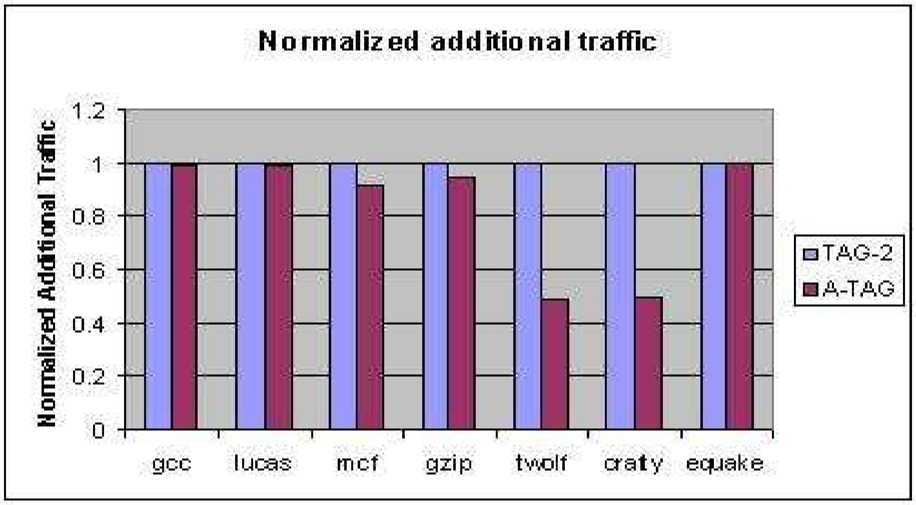


Figure 15: Comparison of normalized Additional Traffic

prefetching is effective, A-TAG adapts to a higher degree of prefetching, while in the other cases, it remains at a lower degree.

7 Future Work and Conclusions

The current work does not explore the performance of prefetch algorithms as we vary the algorithm parameters. We would also like to come up with an adaptive algorithm for CDP that uses TAP. CDP seems an ideal case for adaptive tuning because it is effective only when the program is involved in pointer chasing among heap data structures. It could be turned off for the rest of the program. Our current implementation of CDP does not perform too well. A better implementation could lead to the design of an adaptive algorithm. Prefetch timeliness is another issue not covered in this work. A prefetch initiated just before the line is accessed does not improve CPU performance significantly. Recent studies indicate that inability to issue timely prefetches could render prefetching totally useless [6]. IPCs are an indirect measure of prefetch timeliness, but a study that directly looks at this issue would provide more insights.

In conclusion, data prefetching is an important mechanism to bridge the growing gap between CPU and memory performance. We have emphasized the importance of classifying prefetches correctly, and towards this end, we have implemented PTMT by Srinivasan et al [1] and studied some prefetch algorithms using it. Our results reveal some drawbacks of PTMT. We have proposed TAP, our own taxonomy for classifying prefetches. Relative to PTMT, we believe TAP is simpler and provides a better classification of prefetches. Using TAP, we have proposed and evaluated the performance of an adaptive tagged prefetch algorithm.

References

[1] Viji Srinivasan, Edwards S. Davidson and Gary S. Tyson, ”A Prefetch Taxonomy,” *IEEE Transactions on Computers*, Vol 53, No. 2, Feb 2004.

[2] Santhanam. V, Gornish E. H, and Hsu W. C, ”Data Prefetching on the HP PA-8000,” *Proc. of the 24th International Symposium on Computer Architecture (ISCA)*, June 1997.

[3] Yeager K. C, ”The MIPS R10000 superscalar microprocessor,” *IEEE Micro*, vol. 16, No. 2, Apr 1996.

[4] Steven P. Vanderwiel and David J. Lilja, ”Data Prefetch Mechanisms”, *ACM Computing Surveys*, Vol. 32, No. 2, June 2000.

[5] Robert Cooksey, Stephan Jourdan and Dirk Grunwald, A Stateless, Content-Directed Data Prefetching Mechanism, *ACM SIGPLAN Notices*, Volume 37 ,Issue 10 (October 2002)

[6] P. G. Emma, A. Hartstein, T. R. Puzak, V. Srinivasan, ”Exploring the limits of prefetching,” *IBM J. Res. & Dev.*, Vol 49, No. 1, January 2005.

[7] D. Burger and T. Austin, The simplescalar toolset, version 3.0, Technical Report, Department of Computer Sciences, University of Wisconsin, 1997.

[8] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam, ”MicroLib: A Case for the Qualitative Comparison of Micro-Architecture Mechanism,” *Proceedings of the 3rd Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2004.

[9] Bhavesh Mehta, Dana Vantrease and Luke Yen, ”Cache Showdown: The Good, Bad and Ugly,” Project Report, Computer Sciences Department, University of Wisconsin Madison, 2004.

[10] http://www.cs.ucsd.edu/ calder/simpoint/single-sim-pionts.htm

[11] Alaa R. Alameldeen and David A. Wood, ”Adaptive Cache Compression for High-Performance Processors,” *Proc. of the Thirty First International Symposium on Computer Architecture (ISCA),* June 2004.

[12] Ando Ki and Alan E. Knowles, ”Adaptive Data Prefetching Using Cache Information,” *Proc. of the Eleventh International Coference on Supercomputing*, pp 204-212, 1997.

A Appendix

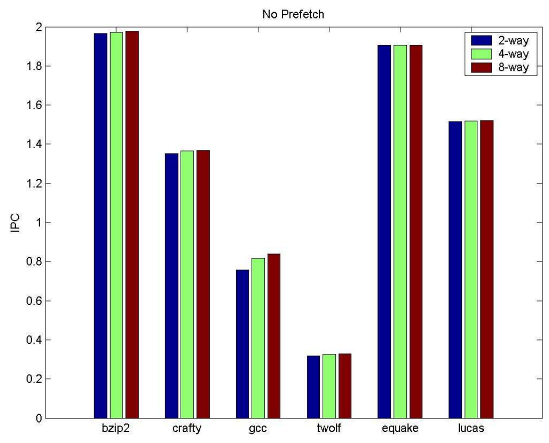


Figure 16: IPC, no prefetching

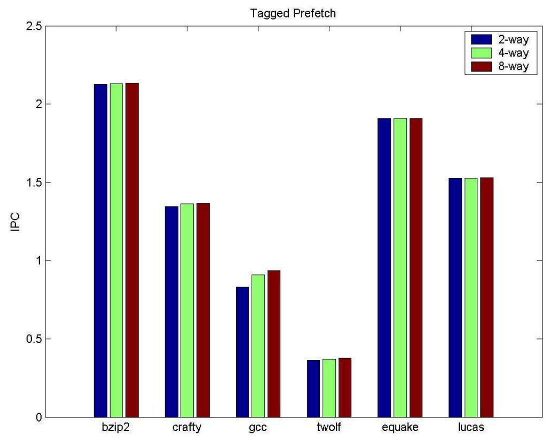


Figure 17: IPC, tagged prefetching

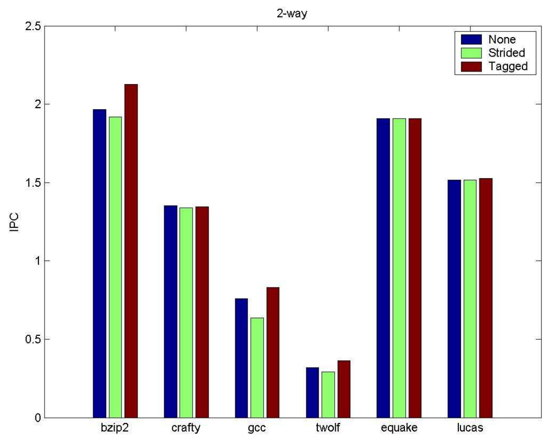


Figure 18: IPC, 2-way

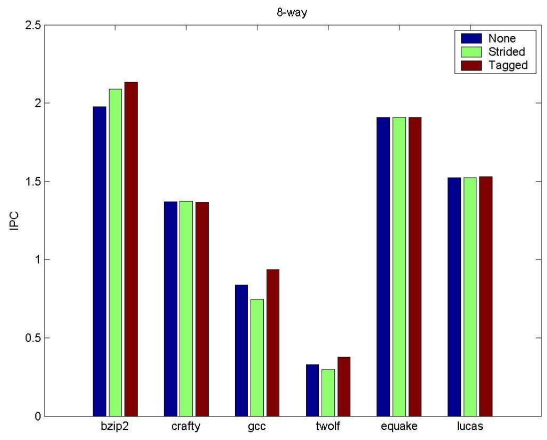


Figure 19: IPC, 8-way

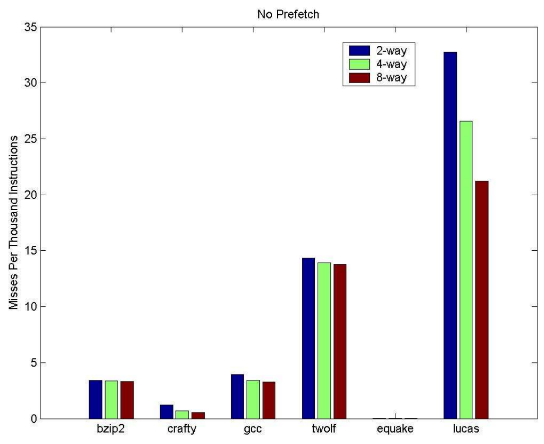


Figure 20: Misses, no prefetching

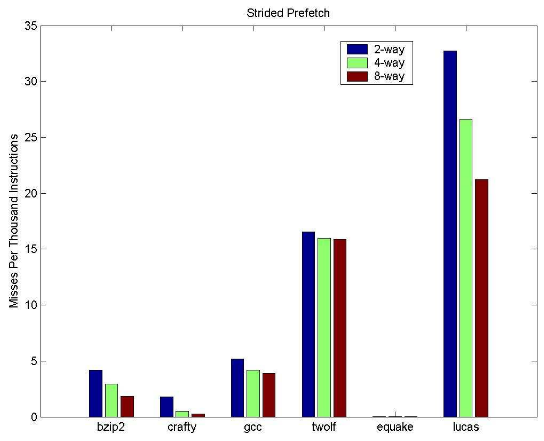


Figure 21: Misses, strided prefetching

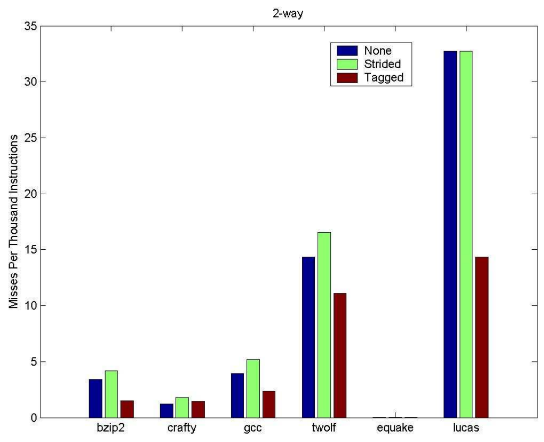


Figure 22: Misses, 2-way

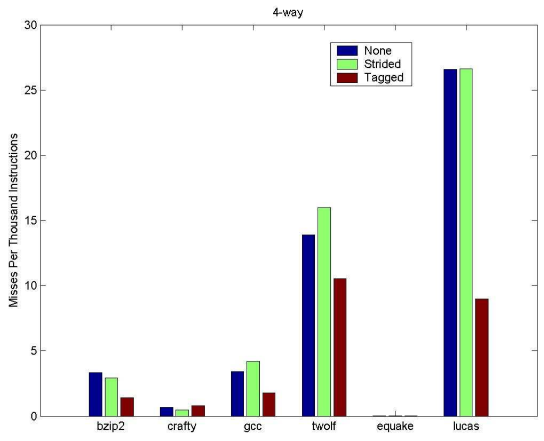


Figure 23: Misses, 4-way

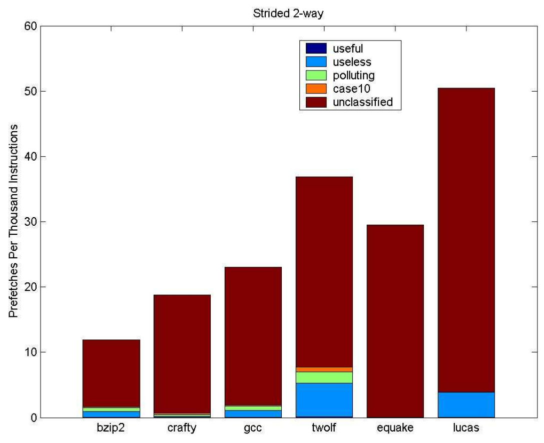


Figure 24: Prefetches, strided 2-way

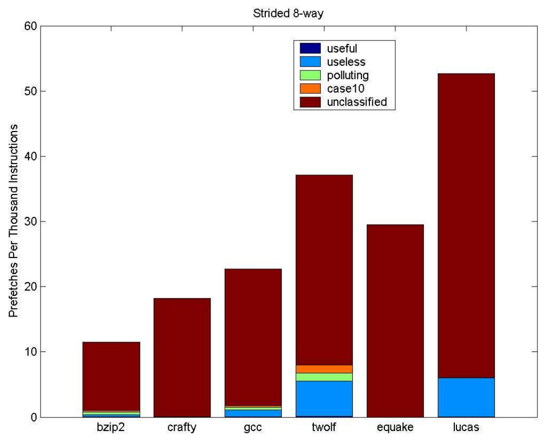


Figure 25: Prefetches, strided 8-way

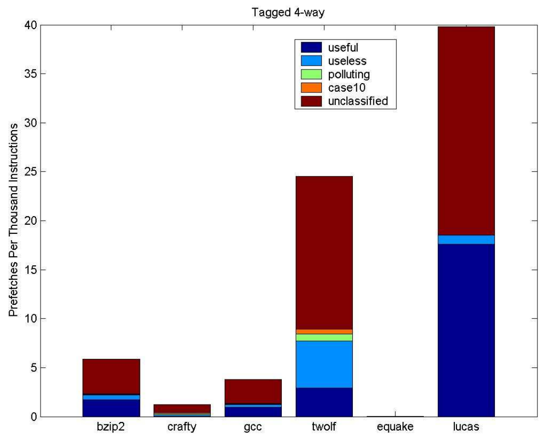


Figure 26: Prefetches, tagged 4-way

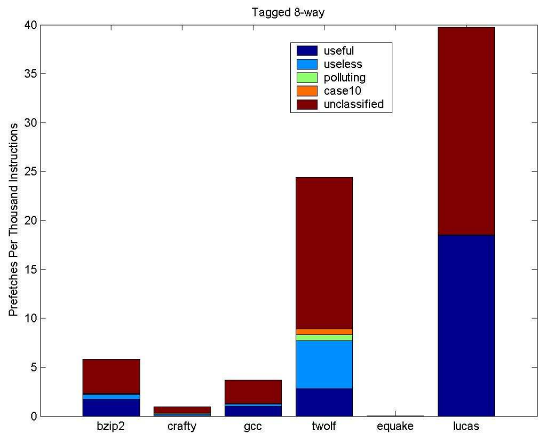


Figure 27: Prefetches, tagged 8-way

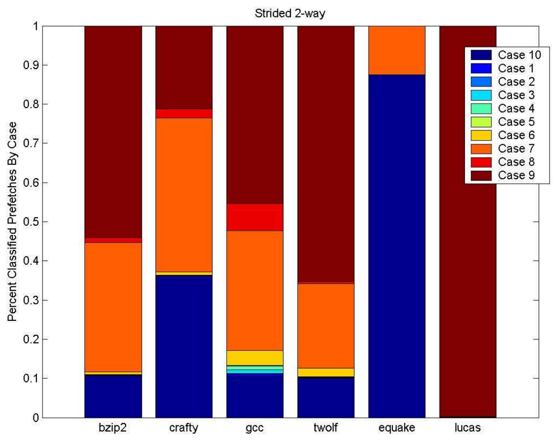


Figure 28: Cases, strided 2-way

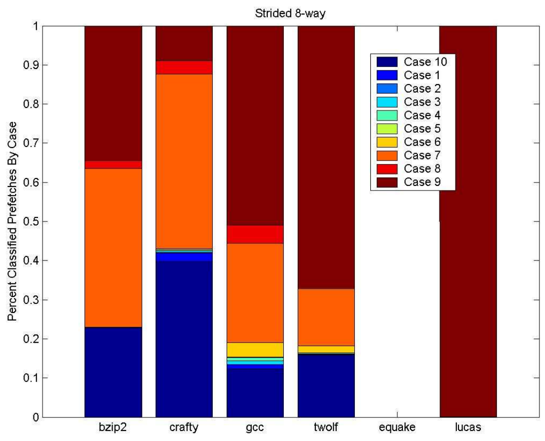


Figure 29: Cases, strided 8-way

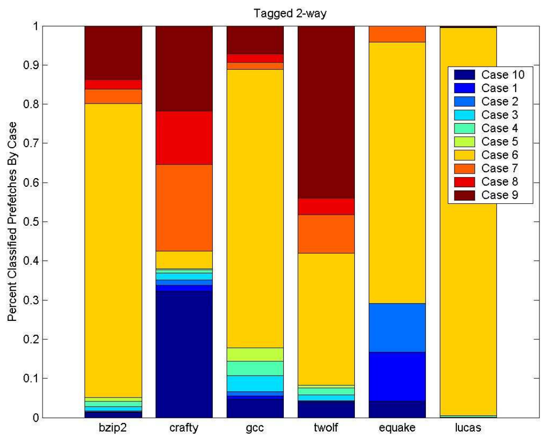


Figure 30: Cases, tagged 2-way

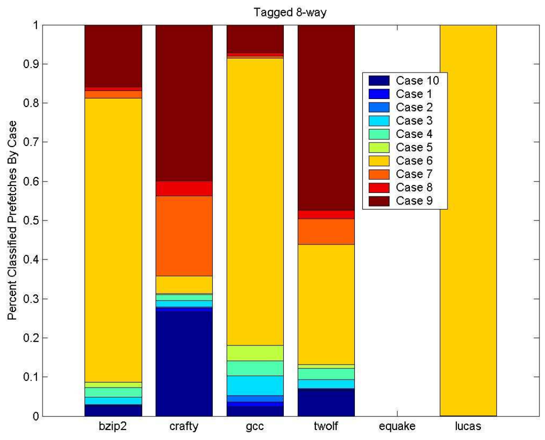


Figure 31: Cases, tagged 8-way