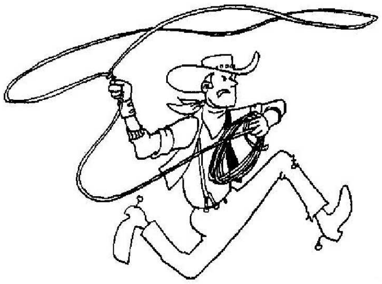

Cache Showdown: The Good, Bad, and Ugly

Bhavesh Mehta, Dana Vantrease, Luke Yen

Computer Sciences Department

University of Wisconsin, Madison

Abstract

Prefetching algorithms have been mainly studied in the context of the Coverage and Accuracy metrics. While this is an appropriate metric for prefetching into separate stream buffers, it is a poor assessment of prefetching into a shared cache structure where cache pollution can become a serious factor. Traditionally, prefetches have been categorized as "good" or "bad" if they are accessed or are evicted without being accessed, respectively. We propose a new categorization of "good", "bad", and "ugly" that takes into account prefetches that cause harmful cache pollution.

This paper proposes a novel structure called the Evict Table (ET) that gauges the amount of cache pollution caused by prefetching into a shared data structure, such as a cache. We show the value of the ET in the context of a chip-multiprocessors, where prefetching among several processing nodes may further increase the contention for cache real-estate. Specifically, we use the ET as an aid in evaluating the effects of Unistride and Czone prefetching algorithms on a chip-multiprocessor's shared L2 cache across a varying number of cache sizes.

1 Background

While processor speeds have been increasing dramatically, memory systems have not been able to keep pace, causing a steadily increasing gap between processor cycles and memory access cycles. This has given rise to various latency tolerance mechanisms such as multilevel cache hierarchies and out-of-order instruction execution. With the advent of aggressive superscalar processors where multiple instructions are in flight at any point of time, the role of cache hierarchies have become more critical and demanding. *Prefetching* [14] is a technique to reduce memory latency by obtaining data before potential accesses to it occur. Cache prefetching refers to bringing cache lines into the cache preemptively before a demand miss occurs to those lines. This technique has actually always been used in cache memories. When we miss on a particular word, we not only bring that word into the cache but also fetch the whole cache line containing that word. This is called *Implicit prefetching*. The effectiveness of this technique generally increases as the cache line size is increased, up to the point of high bandwidth costs in uniprocessors and false sharing in multiprocessors. Exploiting the same technique more aggressively gives rise to what is called *Explicit prefetching*. For example, upon a miss we could not only fetch the cache line containing the word missed upon, but also choose to fetch a larger set of cache lines. When to start and stop prefetching, what lines to prefetch, and where to put prefetch lines are some of the trade-offs to consider when evaluating prefetch strategies.

Prefetched data might be placed into named locations, such as processor registers. This kind of prefetch is called *binding prefetch*. Alternatively, using the idea of *non-binding prefetch* we can place the prefetched data into a cache in the memory hierarchy or into separate structures like stream buffers [9]. When prefetching into a cache we must be weary of *cache pollution*. Cache pollution is when a prematurely prefetched block displaces useful data in the cache. That is, if the block had not been displaced, the processor would have hit to it, but because of prefetching, a miss results.

In order for prefetches to be useful, they must be timely. In one case, if a useful prefetch is prematurely allocated, it may be evicted from the prefetch structure before its use. In another case, an unused prefetch may be poorly timed in such a way that it pollutes the cache, evicting live cache data. There are several prefetch policies that try to alleviate these situations. One of them is *tagged prefetch* [13], which provides timeliness by keeping a tag associated with all prefetch lines, and only issuing additional prefetch requests when a previous prefetch line's tag has been set, which occurs when that prefetch line is accessed. Another policy is prefetching only from the miss address stream, which is less aggressive than initiating prefetches by watching the entire address stream, and is based on the idea of miss addresses being localized (misses to an address usually lead to additional misses around that address).

There are two measures that have traditionally been used to quantitatively measure prefetch schemes. The first, accuracy, refers to the fraction of all prefetches that are actually used. The second, coverage, is the fraction of all memory requests that can be elminated by a particular prefetching scheme. To measure coverage and accuracy all prefetches have been in the past categorized into “Good”(G) and “Bad”(B) prefetches. According to Srinivasan et. al, a ”Good” prefetch is one that is accessed before it is replaced, while a ”Bad” prefetch is one that is replaced before it is accessed. Quantitatively [6], if the total number of misses are M without prefetching,

$$Coverage = G/M$$

$$Accuracy = G/(G+B)$$

2 Introduction

Prefetching has been effective in uniprocessors [1, 4, 5, 7, 8, 9, 10]. It is also important in Chip-Multiprocessors (CMPs) (e.g. Power4 [3]). We consider the case of CMPs with private L1 caches and a shared L2 cache that is shared amongst prefetches and cached values. There are extra benefits to prefetching into a CMP’s shared L2 over a uniprocessor’s private L2. For example, good prefetching will not only help avoid misses from one processor but also potentially avoid misses from other processors, if there is sharing. However, CMPs are may be susceptible to inaccurate prefetches because resources wasted by inaccurate prefetches are shared among all processors and hence all the processors might be affected. Such effects, in particular the effects of prefetch-provoked cache pollution across the processing nodes, are the focus of this study.

We use an Evict Table 3 to evaluate different prefetching policies in terms of cache pollution. In particular we compare *Unistride* and *Czone* prefetching techniques. The unistride scheme, or the fixed-stride scheme, is based upon the idea that if we miss to a particular cache line we are likely to access the next cache line also (spatial locality). Czone [12] prefetching is an arbitrary stride prefetching scheme, and is shown in figure 1.

According to Palacharla et. al, in Czone we partition each word address into two parts: the Concentration zone (Czone) and the address tag. In addition, there is a filter table associated with the prefetcher, which keeps some state for each address that initiates any prefetch requests and is used by the prefetcher’s finite state machine (FSM) whenever it tries to decide whether to issue prefetches to that address or not. Each entry of the non-unistride filter table, in addition to the tag of the Czone partition, has a few state bits, and last address and stride fields which are required to implement the stride detecting FSM. At the end of three consecutive strided references a stream is allocated and the entry in filter is freed.

The rest of this paper is as follows. In section 3 we introduce Evict Table and its functionality, in section 4 we present the simulation methodology. Section 5 discusses the results of our simulation. Section 6 talks about related work. In section 7 we conclude and in section 8 we discuss some of the issues we want to handle in future.

3 Evict Table

3.1 Description

The Evict Table (ET) is a mechanism by which we can track the effects a given prefetch policy has on a cache that shares its real-estate with prefetches. Its primary function is to keep track of the fraction of all prefetches

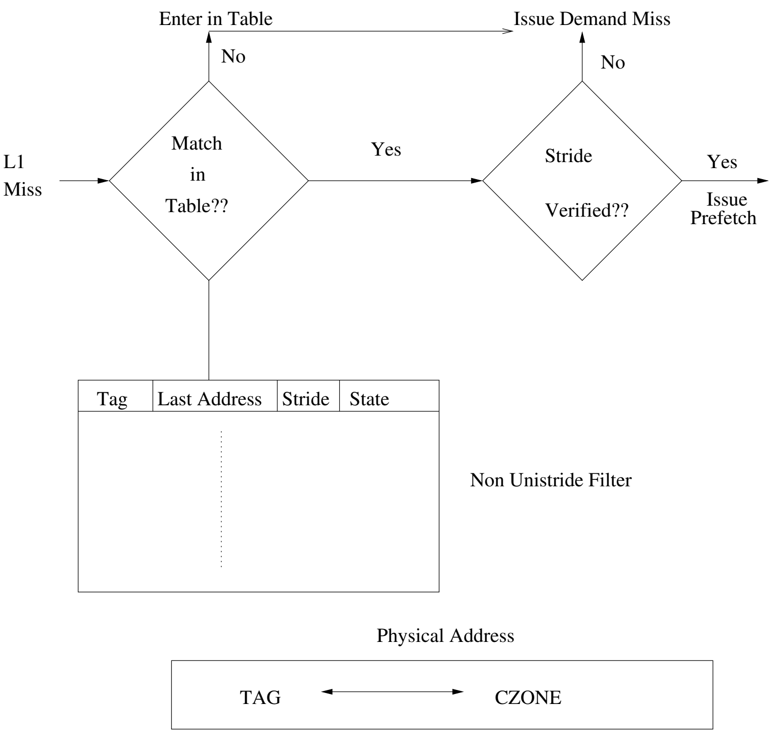

Figure 1: **Czone scheme.**

issued that are useful, harmful, and useless (also known as ”good”, ”bad”, and ”ugly” prefetches). Useful prefetches are those that are actually accessed, while harmful prefetches are those that are never accessed and are a direct cause of misses to cache lines that were displaced by that prefetch line. Finally, useless prefetches are those that are never accessed. In this sense, the primary negative effect of this type of prefetch line is consuming unnecessary memory bandwidth. We have not yet explored quantitatively the amount of memory bandwidth consumed, but rather we measure the negative effects of useless prefetches indirectly by comparing the fraction of useless prefetches to useful and harmful prefetches.

The definitions of ”good”, ”bad”, and ”ugly” prefetches differ from the traditional senses of ”good” and ”bad” prefetches in key ways. In particular, ”good” prefetches are still described with accesses to prefetch data, however they have been adjusted to take into account cache pollution. Thus, whenever prefetch induced cache pollution is a problem, the number of ”good” prefetches, as we define it, will always be less than the traditional naive definition. In fact, the traditional definition is susceptible to overestimating the usefulness of prefetches and can actually lump our notion of ”bad” prefetches in with the ”good” prefetches. ”Bad” prefetches, as we define them, have no equivalent terminology in the realm of prefetching. ”Ugly” prefetches are equivalent in meaning to traditional ”bad” prefetches. We avoid using the term ”bad” here and prefer using ugly because a prefetch going unused is (in most cases) much less harmful than one that evicts soon-to-be-accessed data.

In implementation, we calculate ”good”, ”bad”, and ”ugly” in the following way:

 # Good Prefetches = # Hits to Prefetched Data - # Hits to Evicted Data

 # Bad Prefetches = # Hits to Evicted Data

 # Ugly Prefetches = Total # of Prefetches - # Hits to Prefetched Data

The ET functions by keeping an additional structure which holds the address tags of the cache line victims which have been displaced due to prefetch lines. At any point in execution, the union of the non-prefetched tags in the cache and the evict tags in the ET give the state of of cache as if prefetching had never occurred. In order to ensure the latter, time-stamps must accompany all tags in the cache and the ET. Since the ET acts solely as a monitor indicating whether or not a prefetch was harmful, it does not provide a recovery mechanism (such as the cache line’s data) when misses occur. Currently, our ET has as many entries as there are total numbers of cache blocks. This was chosen so that we could keep track of all the last victims displaced by prefetch lines, and so that we could easily ensure that several ET invariants could be maintained.

The first invariant is that for each prefetch line in the cache, there must be a corresponding victim in the ET. The term victim does not necessarily mean a true cache line that was displaced by the prefetch line, since a prefetch into an invalid block does not displace a victim at all. Rather, for this scenario we maintain the invariant by inserting a special NULL entry, which does not keep any real information except a valid timestamp, which is used in order to participate in LRU replacement of ET entries. Thus the first invariant can be stated as follows:

# of Prefetch Lines in the Cache Set = # of Victim Entries in the ET Set

The second invariant for the ET is as follows: at any given time, the number of victims in a given ET set plus the number of non-prefetch cache lines for that set in the cache should always be less than or equal to the associativity of the cache:

# of Victim Entries in ET Set + # of Valid Non-Prefetch Cache Lines in Set $\leq$ *Cache Set Associativity*

This invariant is true because the number of valid non-prefetch cache lines must always be less than or equal to the cache associativity, and the remaining cache blocks for that set can be taken up by prefetch lines, which must have a corresponding victim entry in the ET.

We now describe, in several different scenarios, the operations of the ET, which depend on the state of the cache at any given snapshot in time.

3.2 Inserting Prefetches

The first scenario involves prefetch lines being inserted into invalid cache blocks (see Figure 2). As mentioned earlier since there is no valid data line being evicted we will simply insert a NULL entry in the ET set in order to satisfy our two invariants.

The second scenario (see Figure 2) is when a prefetch line evicts a valid cache line. In this case, we must insert the victim's address tag into an ET entry in the same set, and give the entry the current timestamp. Note that we must have space available in the ET for this set, since the victim we are replacing is a cache line; hence there must be at least one unoccupied ET entry for this set corresponding to the cache line. In this case we still maintain both of our invariants because each prefetch has a corresponding victim ET entry for the set, and the sum of the victims plus the number of valid non-prefetch cache lines is less than or equal to the cache set associativity.

The third scenario involves an incoming prefetch line evicting another prefetch line in the cache (see Figure 2). In this case we do not add or remove entries in the ET for this set because we have not evicted any non-prefetch cache lines. Therefore, the current victim entry in the ET corresponding to the victim prefetch line is sufficient to maintain our two invariants. It is worth noting that the prefetch evicted might have been a useful prefetch, and though our ET does not capture this chain of prefetches replacing prefetches, it could be extended to do such.

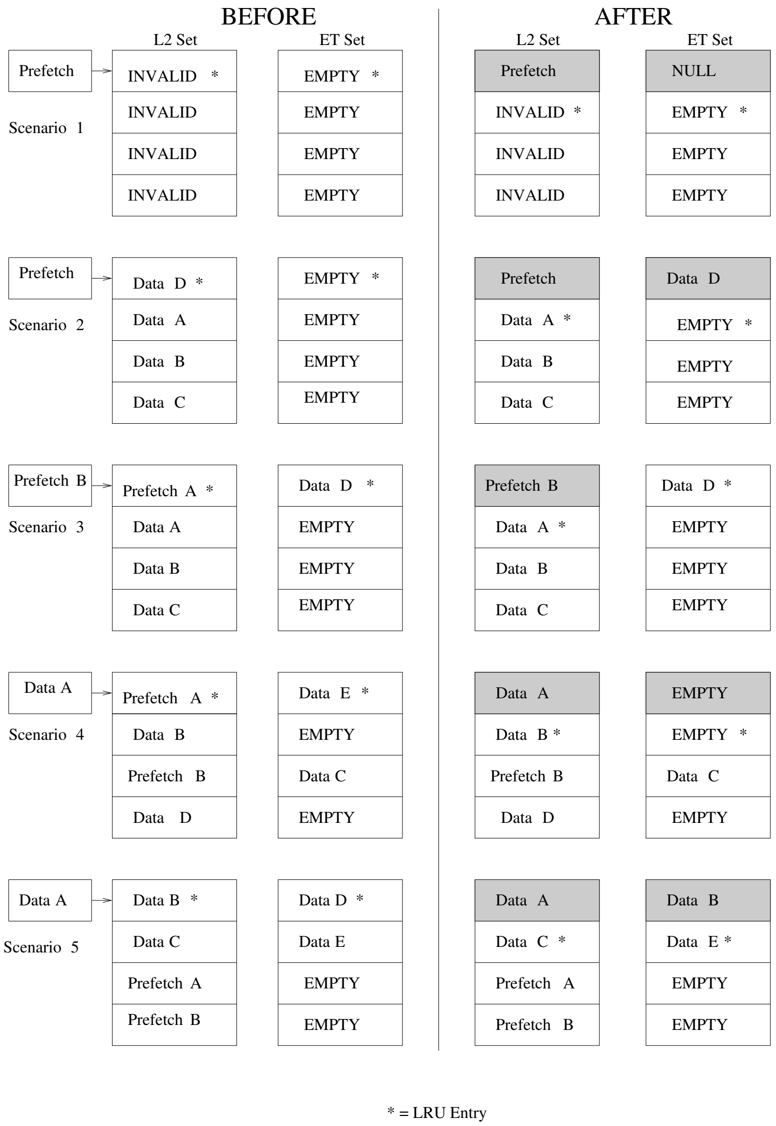

Figure 2: **How Evict Table works**

The fourth scenario (see Figure 2) involves an incoming non-prefetch cache line replacing a prefetch cache line. In this case, due to our first invariant, there is a victim ET entry for this set and when evicting the prefetch cache line we must also evict its corresponding victim ET entry. Therefore our two invariants are still maintained after these actions since we have one less ET entry and one less prefetch cache line than before, but we also increased the number of non-prefetch cache lines.

The final scenario for our ET occurs when an incoming non-prefetch cache line replaces another non-prefetch cache line, and there exists prefetch cache lines in the (see Figure 2). In this case, we must evict the Least Recently Used(LRU) ET entry, allocate a new ET entry and fill it with the address tag of the cache line which is going to be evicted. The reason for doing these two steps can be explained in the following way. If at least one of the prefetch cache lines in the cache had not existed, there would be a possibility of the incoming non-prefetch line co-existing with the victim non-prefetch cache line. However we have eliminated this possibility due to the existence of at least one prefetch cache line. Therefore we must update our victim list to account for this possibility, which is done by evicting the LRU ET entry and allocating a new entry based on the address tag of the evicted non-prefetch cache line. Note that it is important that we follow these two steps because of the need to maintain our two invariants.

3.3 Servicing Requests

Table 1 describes how the ET goes about servicing requests in a manner that ensures all of the invariants are held. Though there is a one-to one correspondence between the prefetch entries and the evict table entries, no single prefetch entry is bound to an evict table entry. This allows LRU to continue in a natural progression. Thus, all entries from the ET are removed in LRU order. In some cases, such as a hit to the ET, a specific entry is to be removed. We accomplish this by calling a setLRU function to ensure that when the miss comes back the entry which is hit to is removed. By simply setting the ET entry to LRU, rather than removing it before the miss is serviced, this prevents race conditions between the prefetch port and the request/response port from violating the invariants.

4 Simulation Setup

To explore various CMP prefetching techniques, we used the Multifacet simulation infrastructure to simulate a multiprocessor server [2] running scientific and commercial workloads. Our target system is a 4-processor

Table 1: **Interactions with the Cache and its Evict Table**

Cache Non-prefetched Hit	Cache Prefetched Hit	Evict Table Hit	Cache Action	ET Action	Comment
No	No	No	Issue Miss	No action	No changes
No	No	Yes	Issue Miss	Bad Prefetch, set LRU	Prefetch evicted useful data
No	Yes	No	Hit, goto non-prefetched state	Good Prefetch, remove LRU	Prefetch hit before data
No	Yes	Yes	Hit, goto non-prefetched state	setLRU, remove LRU	Useful data evicted and prefetched backed in
Yes	No	No	Hit	No Action	Nothing changes
Yes	No	Yes	Hit	setLRU	Useful data evicted, prefetched back in and hit to
Yes	Yes	No	No	No action	Impossible: Cannot have identical tags in cache
Yes	Yes	Yes	No	No action	Impossible: Cannot have identical tags in cache

system (1 CMP with 4 processors), running Solaris v9. Each processor has its own L1 Data and Instruction caches. The L1 Cache is writeback and maintains inclusion with L2. The four processors on a chip share an L2 cache. Each chip has a point to point link with a coherence protocol controller, and a memory controller for its part of the globally shared memory. The system implements sequential consistency using directory-based cache coherence. The L1 uses the MSI cache coherency protocol, and the shared L2 uses MOSI cache coherency. We evaluated two prefetching schemes, Czone and unistride prefetching at the L2. Prefetches are allocated in the L2 and the ET resides at the L2. We used Kyle Nesbit and Nick Lindberg's [11] implementation of the Czone prefetcher, and modified it to also allow unistride prefetching. For both schemes we set the prefetch degree to 1. In all cases we used infinite bandwidth for the links so that they would not be the bottleneck in the system. The system configurations is shown in Table 2.

5 Results and Analysis

We compared the two prefetching schemes, Czone and unistride, by fixing the prefetch degree at 1, and varying the sizes of the shared L2 cache into which the prefetch lines go into. We then ran simulations using

Table 2: System Parameters

Total Number of Processors	4
Processors per Chip	4
L1 I-Cache	64KB 2-way set associative
L1 D-Cache	64KB 2-way set associative
Shared L2 cache	1-16MB 4-way set associative
Total TBEs	128 entries

the Apache, Barnes, Ecperf, JBB, Ocean, OLTP, and Zeus benchmarks (using short runs). In all instances we collected data using our Evict Table as well as using the statistics from Ruby in order to compare the two prefetching schemes and their performance differences as we varied the cache size.

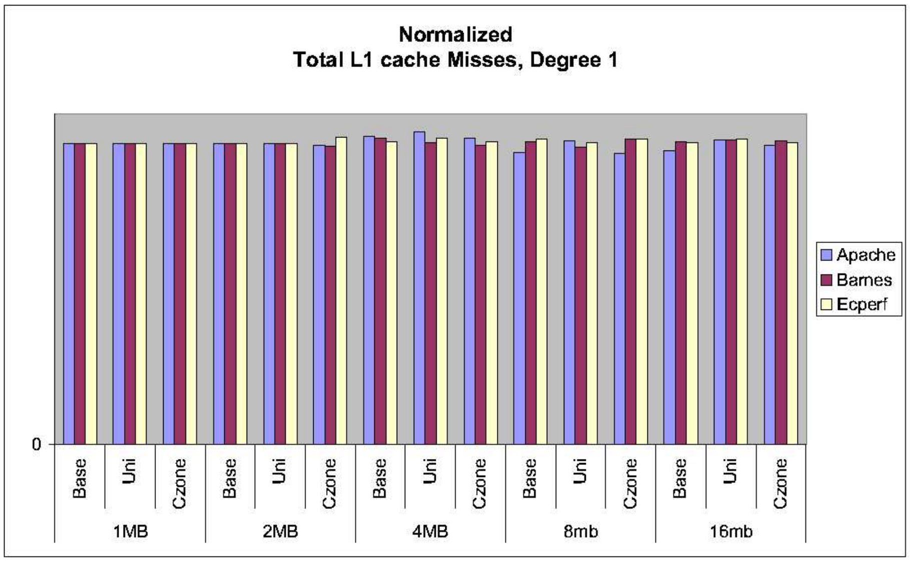

Figure 3: **Normalized L1 misses 1.**

First, we examined the total number of L1 cache misses, which includes both instruction and data cache misses. We normalized the results for our two prefetching schemes against the Base configuration for the same cache configuration but with no prefetching at all. As can be seen in Figures 3 and 4, for the Apache, Barnes, ECperf, JBB, Ocean, and Zeus the total L1 cache misses did not differ much between the Base, Czone, and Unistride prefetching schemes. OLTP experience some variations in L1 cache misses as we varied the cache size and also used different prefetching schemes. On one hand, these results indicate that cache pollution affecting inclusive data in the L2 cache is not a major problem for these benchmarks, since the L1 cache misses stay relatively constant in light of prefetching. However, according to data in Figure 5 as we increase the cache size the total number of prefetches decreases. This increase of prefetches compared to total number of prefetches in bigger cache sizes can be explained by the following. Our prefetch scheme currently initiates prefetches by observing the L1 miss address stream or by hits to prefetches which haven't received its data from main memory (hence it is in some intermediate cache coherency state). Since we are not issuing many more L1 misses as compared to the Base case, we assume the miss traffic in both quantity and content is similar. Thus we hypothesize that the large prefetch number comes from another source: hitting to the prefetches in the intermediate cache state. According to our protocol, such hits invoke more prefetches. We have not extensively explored this phenomenon, and this should be investigated further in future work.

Likewise, we see that in Figures 6 and 7, all of the benchmarks with the exception of OLTP and ECperf

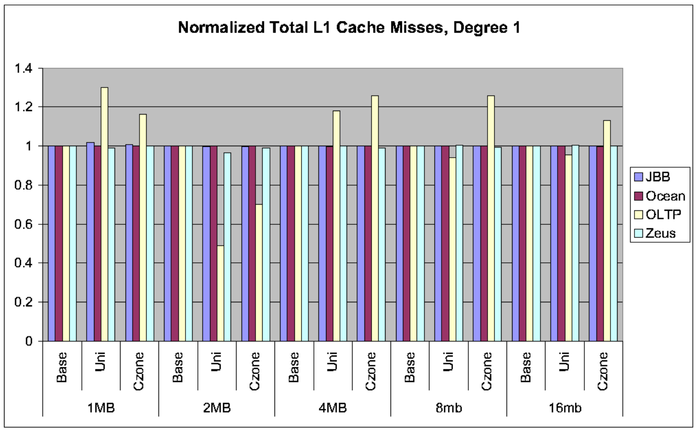

Figure 4: **Normalized L1 misses 2.**

experienced relatively the same quantity of L2 misses. OLTP was unlike the others due to its inconsistent behaviors, which does not mesh well with prefetchers. Similarly, due to the short simulations of ECperf we could not concretely categorize its behavior as consistent as we could with the remaining benchmarks. In summary, there was a slight decrease in the total number of L2 cache misses, but it was not significant and could be attributed to the randomized nature of the simulator.

The effects of the cache misses, particularly the L2 cache misses, directly translates to the performance data we see in Figure 8. As we stated earlier, OLTP was not amenable to prefetching, and therefore experienced a little more than 20% increase in cycles per transaction. The other benchmarks had a slight decrease in cycles per transaction, which results in some performance improvement (around 1%).

We next discuss how our ET focuses on the categorization of prefetches into “good”, “bad”, and “ugly”. Some benchmarks, such as Barnes, ECperf, and Ocean are less sensitive to bad prefetches than others and experience few bad prefetches. This phenomenon is likely the result of the application being mostly read-only or the benchmarks having a small working set that is unaffected by the area prefetches consume.. In

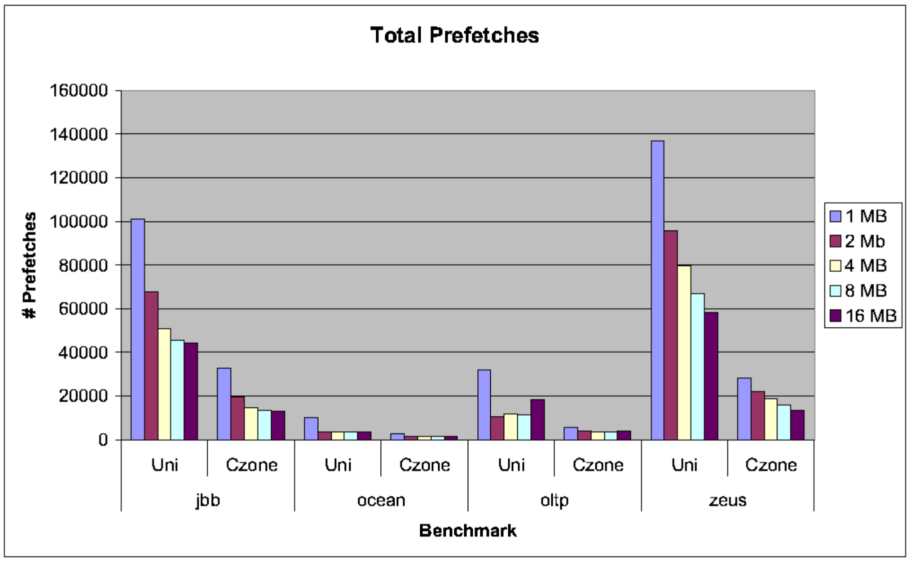

Figure 5: **Total Number of Prefetches.**

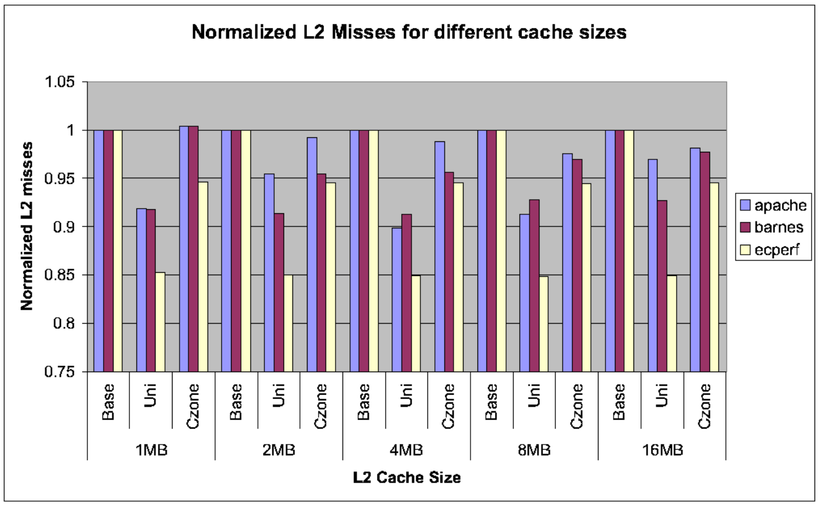

Figure 6: **Normalized L2 misses 1.**

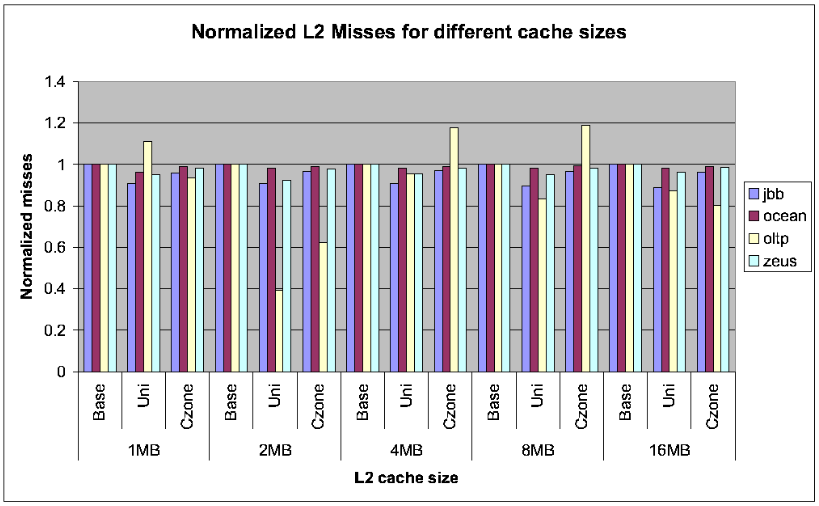

Figure 7: **Normalized L2 misses 2.**

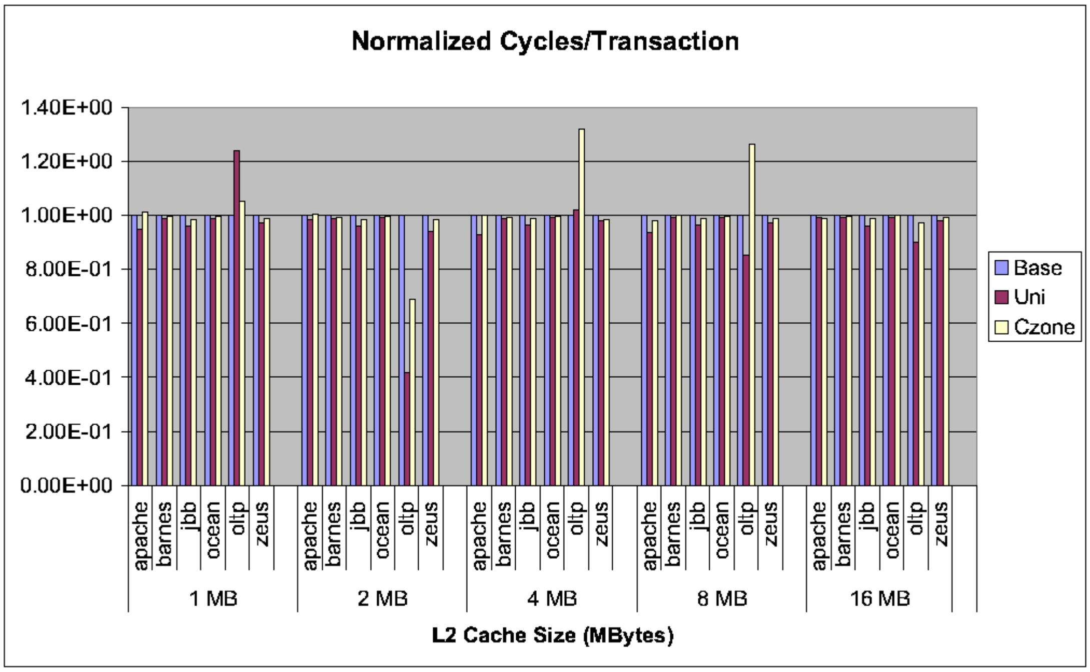

Figure 8: **Cycles per Transaction Performance.**

other benchmarks, including Apache, Jbb, OLTP, and Zeus, this was not the case. The percentage of bad prefetches is directly linked to the cache size, and as the cache becomes larger, the pollution is less serious. As a side effect experecing less cache-pollution it becomes more probable that prefetches will be categorized as “good’.

Cache pollution in smaller caches, compounded with prefetching more at smaller cache sizes (Figure 5), contributes to the cache pollution. Despite cache-pollution being a problem, in terms of misses, more prefetches are “good” than “bad” in simulations. This is not to say, however, that the number of “bad” misses is insignificant. Note that the “bad” prefetches make up as much as 20% of the prefetches and these quantify the mis-classification of prefetches which traditionally have been considered “good”. This is a significant enough percentage to give the wrong impression of a prefetching scheme’s effectiveness.

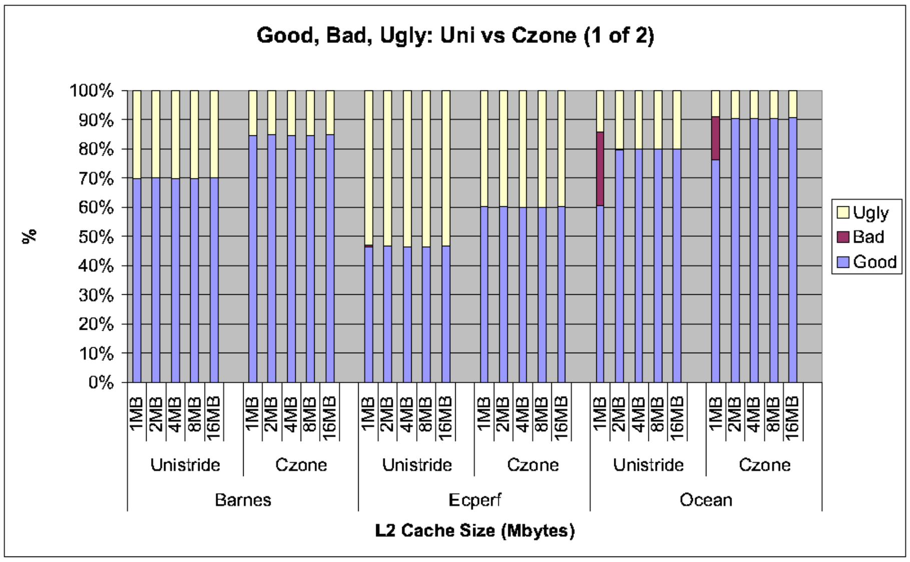

Figure 9: **Good, Bad, Ugly Distribution 1.**

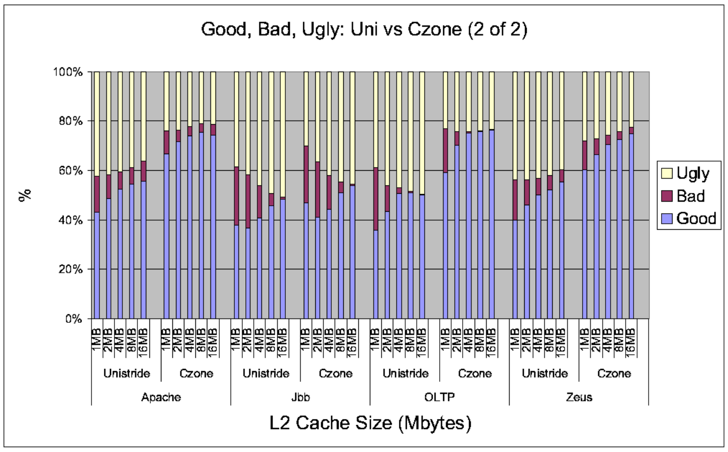

Figure 10: **Good, Bad, Ugly Distribution 2.**

6 Related Work

Prefetching has been explored and discussed in great detail in literature [1, 4, 5, 7, 8, 10]. Recently, Viji Srinivasan et. al. [6] introduced a rigorous way to evaluate the "goodness" of a prefetch scheme by introducing costs to every memory action. They quantified these costs at a high level by categorizing the amount of extra requests and extra cache misses prefetching introduces. They provided a similar analysis to ours by comparing a cache which had prefetching to a cache, of the same size and configuration, that did not have prefetching. They provided analysis based on the different scenarios that could occur, depending on whether the evicted cache line is accessed later on. In this manner their coverage of the different scenarios can be mapped directly to the scenarios covered by our ET, albeit at a more simplistic setting.

In contrast, our simulated system is much more complex than the one they simulated on. Their results were for a single cache system that did not have inclusion, and using a prefetcher that issued prefetches regardless of whether they already existed in the cache. Our simulation system uses a cache hierarchy that maintains inclusion, withe the private L1 caches being writeback caches, and also using a prefetcher that drops prefetch requests if the address tags already exist in the L2 cache. Srinivasan et. al evaluates the effectiveness of prefetching in a mathematical way, which would be very complex and tedious for our simulated system due to the overwhelming number of factors that need to be accounted for when making that type of analysis. In addition, their analysis assumes an arbitrary prefetch scheme, whereas we performed our analysis with two specific prefetching schemes, Czone and unit-stride.

7 Conclusions

In this paper we introduce a novel structure, the Evict Table, which captures the side effects a particular prefetching scheme has on caches. It captures the "good", "bad", and "ugly" prefetches (corresponding to useful, harmful, and useless prefetches, respectively) for any given prefetch scheme, and allows accurate feedback on how that scheme performs for a given system configuration.

We also compared the performance of two prefetching schemes, the Czone arbitrary stride prefetcher and the unistride prefetcher, under different shared L2 cache configurations varying from 1-16MB. Both prefetching schemes perform similarly for a prefetch degree of 1, and while we have not presented graphs of the schemes using prefetch degree 8 we have found that both schemes degrade drastically at this prefetch degree. Regardless of what prefetch degree is used for the prefetcher we feel that the Evict Table can provide an accurate analysis of the effectiveness of the prefetching scheme and its effects on the memory hierarchy.

We have preliminary results showing that cache-associativity is a factor in how consequential cache pollution is when prefetching into a shared cache. In general as cache size increases the fraction of "good" prefetches increases and the fraction of "bad" prefetches decreases for all the benchmarks. In addition, the fraction of useless prefetches fluctuates across different benchmarks, so no conclusive trend exists as we increase the cache size. However, by categorizing prefetches into these three categories we have provided a valuable tool into how prefetches affect the memory hierarchy.

8 Future Work

There are a number of issues we need to explore in future work. The first issue has to do with memory bandwidth. Although we have captured the number of useless ("ugly") prefetches for both Czone and Unistride prefetching schemes for varying cache sizes, we have not quantitatively measured this excess bandwidth as a fraction of the total memory bandwidth in the system. In order to do this precisely we need to explore different bandwidth configurations and to see the impact of Czone and Unistride prefetching assuming a fixed cache configuration. Related to this is how we might use this information as feedback into the prefetcher. Since the ET can potentially be a hardware structure, we might be able to use the fraction of harmful prefetches (as gathered by the ET) as iterative feedback into the prefetcher, and to use this information to gouge when to turn off prefetching and when to restart it (either by some intelligent scheme or by using random restart).

The second has to do with communication invalidates. This is a problem in with multiple chips in a CMP because if multiple prefetchers on multiple CMP chips issue prefetches to the same address, and one of the chips issues a GETX request because it wants to write to that block, the other chips now have to invalidate that block. This would not have occurred if there had not been prefetches to that same block. Although this has always been a problem in multiprocessor systems in which individual chips have private prefetchers, it might be more or less of a problem in CMPs due to shared caches and smaller wire delays. In order to examine this problem in greater detail we need to collect data on how often communication invalidates due to prefetches occur, and if there is potential in using some hardware in order to alleviate these invalidates.

Thirdly, we have assumed during our simulations that we prefetch into a shared L2 cache, which has the potential of prefetching for other processing nodes at the expense of causing cache pollution. To expand our analysis further we can see the effectiveness of prefetching into a victim cache or stream buffer [9]. We can then compare the results against our ET results to see whether the additional hardware overhead of using these separate structures to avoid cache pollution is justified.

Finally, we need to explore using other cache coherency protocols to examine its effects on cache pollution when prefetching into a shared cache. We have utilized an inclusive cache coherency protocol in our simulations. However, there are some subtle effects that may cause cyclic dependencies to form when we are using inclusive caches. This is because we are initiating prefetches by examining the miss address stream, and the combination of prefetching into a shared L2 cache and possibly evicting inclusive L1 data might cause additional L1 cache misses. Since the prefetcher acts upon miss addresses this effect initiates the prefetcher, which might once again cause additional L1 misses, which forms the cyclic dependency between the L1 cache and the prefetcher at the L2 cache. This situation might be entirely avoided by utilizing exclusive caches along with an exclusive cache coherency protocol, or by using separate prefetch structures as mentioned above.

References

[1] T.F. Chen and J.L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, pages 609–623, May 1995.

[2] Alaa R. Alameldeen et. al. Simulating a $2M commercial server on a $2K PC. *IEEE Computer*, February 2003.

[3] J. M. Tendler et. al. Power4 system microarchitecture. IBM Technical White Paper, 2001.

[4] John W. C. Fu et. al. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110, 1992.

[5] Timothy Sherwood et. al. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, 2003.

[6] Viji Srinivasan et. al. A prefetch taxonomy. *IEEE Computer*, pages 126–140, February 2004.

[7] Z. Martonosi Hu and S. M. Kaxiras. TCP: tag correlating prefetchers. In *The Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[8] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[9] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[10] S. Kim and A. V. Veidenbaum. Stride–directed prefetching for secondary caches. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 314–321, August 1997.

[11] Kyle Nesbit and Nick Lindberg. CMP prefetching. http://www.cs.wisc.edu/ david/courses/cs838/projects/nesbit.pdf, 2004.

[12] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Int'l Symp. Computer Architecture*, pages 24–33, 1994.

[13] Alan J. Smith. Cache memories. *Computing Surveys*, pages 473–530, September 1982.

[14] Steven VanderWiel and David Lilja. Data prefetch mechanisms. ACM Computing Surveys, 32(2):174–199, June 2000.