# Increasing the Cache Efficiency by Eliminating Noise

**Prateek Pujara and Aneesh Aggarwal**
**Department of Electrical and Computer Engineering**
**Binghamton University, Binghamton, NY 13902**
**{prateek,aneesh}@binghamton.edu**

## Abstract

*Caches are very inefficiently utilized because not all the excess data fetched into the cache, to exploit spatial locality, is utilized. We define cache utilization as the percentage of data brought into the cache that is actually used. Our experiments showed that Level 1 data cache has a utilization of only about 57%. In this paper, we show that the useless data in a cache block (cache noise) is highly predictable. This can be used to bring only the to-be-referenced data into the cache on a cache miss, reducing the energy, cache space, and bandwidth wasted on useless data. Cache noise prediction is based on the last words usage history of each cache block. Our experiments showed that a code-context predictor is the best performing predictor and has a predictability of about 95%. In a code context predictor, each cache block belongs to a code context determined by the upper order PC bits of the instructions that fetched the cache block. When applying cache noise prediction to L1 data cache, we observed about 37% improvement in cache utilization, and about 23% and 28% reduction in cache energy consumption and bandwidth requirement, respectively. Cache noise mispredictions increased the miss rate by 0.1% and had almost no impact on instructions per cycle (IPC) count. When compared to a sub-blocked cache, fetching the to-be-referenced data resulted in 97% and 44% improvement in miss rate and cache utilization, respectively. The sub-blocked cache had a bandwidth requirement about 35% of the cache noise prediction based approach.*

**Keywords:** Cache Noise, Block Size, Cache Organization, Spatial Locality, Hardware Prefetching

## 1. Introduction

Caches exploit spatial locality by also fetching the words adjacent to the word for which the miss occurred. This can result in considerable noise (words that are not required) being brought into the cache. *We define cache utilization as the percentage of the useful words out of the total words brought into the cache.* Our studies show that with a block size of 32 bytes, cache utilization for the Level 1 data cache is only about 57% for the SPEC 2K benchmarks (a result also observed previously by Burger et. al. [2] and others). Lower cache utilization results in energy, bandwidth, and precious cache space being spent for useless words. Improving cache utilization, by fetching only the to-be-referenced words can lower energy consumption (by avoiding wastage of energy on useless words), improve performance (by making more cache space available for compression), and facilitate complexity-effective bus design (by reducing the bandwidth requirement between different levels of cache).

Spatial locality in programs can be further exploited by increasing the cache block size. Larger cache blocks also reduce the tag overhead. However, they increase the bandwidth requirement (between the different levels of cache) and the cache noise. On the other hand, smaller cache blocks reduce the cache noise, however, at the expense of larger tag overheads and lower spatial locality exploitation. Sub-blocking is used to mitigate the limitations of larger blocks. In sub-blocked caches, sub-blocks (which are portions of the larger cache block) are fetched on demand. However, sub-blocking can result in significant increase in cache miss rate and cache noise if words are accessed from different sub-blocks. Other alternatives [18, 20] are to dynamically adapt the block size to the spatial locality exhibited by the application, still bringing in contiguous words. In this paper, we investigate prediction techniques (called cache noise prediction) that fetch only the to-be-referenced words (which may be in non-contiguous locations) in a cache block. This technique is an attractive alternative to sub-blocking for larger cache blocks because it has the benefit of reducing the bandwidth requirement, while avoiding the miss rate and cache noise impact of sub-blocking.

Cache noise predictor considers the usage history of the words (defined by the offsets of the words that are actually used) in the cache blocks. We investigated three different kinds of cache noise predictors. The first predictor – *phase context predictor (PCP)* – bases its prediction on the words usage history in the most recently evicted cache block. The second predictor – *memory context predictor (MCP)* – bases its prediction on the words usage history of the most recently evicted cache block in a particular memory context, defined as contiguous memory locations. The third predictor – *code context predictor (CCP)* – bases its prediction on the words usage history of the most recently evicted cache block that was fetched by a program code context, defined as a contiguous set of instructions. Even though the cache noise predictor can be used for any cache, we focus on Level 1 data cache. Our experiments showed that, even a simple last words usage CCP predictor (which we found to be the best performing predictor) gives a correct cache noise predictability of about 95%. We implemented the *CCP* cache noise predictor for L1 data cache and observed that its utilization increased by about 37% and the bandwidth requirement between the L1 and L2 cache reduced by about 28%. Cache noise mispredictions increased the miss rate by only about 0.1%, resulting in almost zero reduction in IPC.

The contributions of this paper are:

1. Illustrating the high predictability of cache noise, proposing efficient cache noise predictor implementations, and showing the benefits of cache noise prediction based fetching of words in terms of cache utilization, cache power consumption, and bandwidth requirement.
2. Illustrating the benefits of cache noise prediction based fetching of words in reducing the bandwidth requirement and cache pollution effects of hardware prefetching mechanisms.
3. Investigating cache noise prediction based fetching of words as an alternative to sub-blocking.
4. Investigating profiling to further improve the prediction capabilities of the predictors.

The rest of the paper is organized as follows. Section 2 presents the motivation behind increasing the cache utilization and discusses the intuition behind the prediction mechanisms proposed in this paper.

Section 3 discusses the implementation of the *CCP* predictor. Section 4 presents the related work. Section 5 presents the experimental setup and the results. Section 6 presents the application of cache noise prediction to hardware prefetching. Section 7 investigates cache noise prediction based fetching of words as an alternative to sub-blocking. Section 8 presents the results of profiling to improve predictor efficiency. Section 9 presents the prediction accuracy results for different sized cache blocks. Finally, we conclude in Section 10.

## 2. Cache Utilization and Cache Noise Prediction

### 2.1 Cache Utilization

*We define cache utilization as the percentage of words brought into the cache that is actually used.* To motivate why techniques increasing the cache utilization should be investigated, we present the level 1 data cache (in this paper, we focus only on level 1 data cache) utilization statistics for the SPEC 2K benchmarks. For the experiments, we ran trace-driven simulations for 500M instructions after skipping the first 3B instructions, for a 32-bit PISA architecture. We used a 4-way set associative, 16KB L1 data cache with a block size of 32 bytes. To measure cache utilization, we divide the total number of unique words touched in the cache by the total words brought into the cache. Figure 1 presents the results of the cache utilization measurement. Figure 1 shows that the cache utilization is bad for almost all the benchmarks (except gcc, applu, wupwise, and mgrid). The average cache utilization is about 42% for the integer benchmarks and about 69% for the floating point benchmarks. Overall, the average cache utilization is only about 57%. An interesting point to be noted in Figure 1 is that the average cache utilization is noticeably lower for integer benchmarks than the floating point benchmarks. This is because the floating point benchmarks access data more sequentially than integer benchmarks and because the floating point benchmarks load and store a considerable number of *doubles* (which occupy two words), resulting in more words being accessed and a better cache utilization.

### 2.2 Cache Noise Prediction

We predict the cache noise based on the history of words accessed in the cache blocks, assuming that programs may repeat the pattern of memory references. This is the same intuition behind the various memory address predictors [1, 6]. First, we discuss the advantages and the disadvantages of the two different words usage histories that can be used for prediction – *eviction time history* and *current history*. In *eviction time history*, the words usage history used for prediction is the words accessed in the cache block that was most recently evicted. In the *current history* method, the words usage history used for prediction is the words currently accessed in the cache block that was most recently fetched. *Eviction time history* has the advantage that the entire words usage history for a cache block is available before a prediction is made. However, the disadvantage of *eviction time history* is that the cache blocks brought into the cache may use a stale history for prediction – because a long time may have passed since the last cache block eviction, or may use an outdated history – because the cache blocks being evicted currently have a words usage pattern that is no longer valid. *Current history* has the advantage of more up-to-date words usage history, but suffers from not knowing the entire set of words that will be used in a cache block. Our experiments showed that an *eviction time history* predictor performed better than a *current history* predictor.

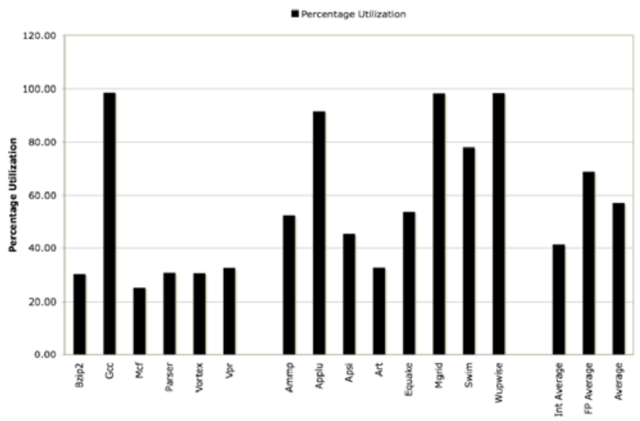


**Figure 1: Level 1 Data Cache Utilization for Spec 2K benchmarks**

In *eviction time history* predictors, the words usage of the cache blocks is recorded as they are accessed and the predictor is updated when a cache block is evicted. We experimented with three cache noise predictors. The first predictor – *phase context predictor (PCP)* – records the words usage history of the most recently evicted cache block and makes the prediction based on that. The second predictor – *memory context predictor (MCP)* – bases its prediction on the assumption that the data accessed from a contiguous location of memory (called memory context) will always be accessed in the same fashion. C*ode context predictor (CCP)* – bases its prediction on the assumption that any instruction in a particular portion of code (called code context) will always access the data in the same pattern. A code context is defined by the upper order bits of the instruction PC, and a memory context is defined by the upper order bits of the memory address. Here, we explain the functioning of the CCP predictor, which can be easily modified for other predictors.

Figure 2 shows the working of the *CCP* predictor for 3 code contexts X, Y, and Z, given by the upper address bits 100110, 101001, and 101110, respectively. A cache block belongs to a code context if it was fetched into the cache (on a cache miss) by a memory instruction in that code context. For instance, any cache block fetched due to a cache miss generated by an instruction with higher order PC bits as 100110 will belong to code context X. *CCP* predictor records the words usage history of the most recently evicted cache block in a code context, and uses it to make predictions for any cache misses originating out of that code context. For instance, in Figure 2, when instruction $I_n$ (which belongs to code context X) generates a miss, the words usage history of the code context X is checked. Accordingly, only the first two words of the cache block A are fetched into the cache. However, the cache block A evicts a cache block in the code context Z. Hence, the words usage history of the evicted cache block is used to update the predictor table for code context Z. Subsequently, when instruction $I_x$ (which belongs to code context Z) misses in the cache, only the first word in the block is fetched.

To measure predictabilities of the predictor, the predicted words usage history for a cache block is remembered when it is brought into the cache. Words usage history may not be available if no cache blocks have been evicted in a particular context, resulting in "no predictions". We observed that the average "no prediction" was almost zero for all the benchmarks. When a cache block is evicted, the words accessed for the block are compared with the remembered words usage history. If the words accessed in a cache block are a subset of the predicted words in the cache block (for instance the prediction is 1011, and the usage is 1001), then we consider it as correct prediction because no additional misses are incurred for that cache block. Figure 3 gives the predictability of the CCP predictor for context sizes of 26, 28, and 30 bits. Figure 4 gives the predictability of the PCP predictor and the MCP predictor with context sizes of 20 and 22 bits. We experimented with many different context sizes. For the *CCP* predictor, the predictability reduced further as the number of bits defining the context is reduced (beyond 26 bits). The *MCP* predictor (with a maximum context size of 27 bits for a 32 byte cache block and 32 bit addresses), on the other hand, behaved differently for different benchmarks. As also observed in Figure 4, as the context size is increased for the *MCP* predictor, the predictability increases for some benchmarks, whereas remains the same or decreases for others. Not much difference is observed in the predictabilities of the CCP predictor with context sizes of 28 and 30 bits because not all the instructions are memory instructions. The predictability of the CCP predictor for floating point benchmarks (an average of about 90% for 28 bits) is much higher than that for the integer benchmarks (an average of about 50% for 28 bits). An average prediction accuracy (correct predictions out of all the new cache blocks fetched) of about 70% is observed. Overall, the CCP predictor has a better predictability than the MCP and the PCP predictors. Next, we discuss techniques to further improve the predictability.

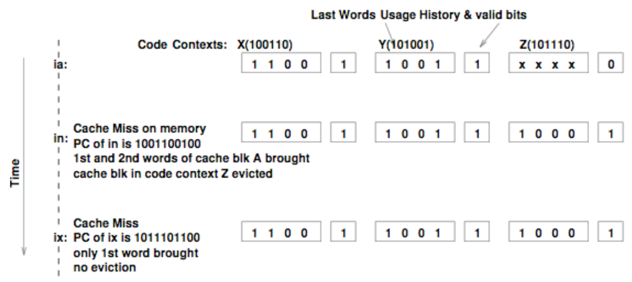


**Figure 2: Working of the CCP cache noise predictor**

## 2.3 Improving Predictability

Here, we discuss 2 techniques that can be employed to improve the predictability of the predictors.

**Miss Initiator Based History (MIBH):** The *MIBH* technique further records the words usage history of cache blocks based on the offset (in the cache block) of the word that initiated the cache miss. We call the word that initiated the miss as the *miss initiator word*. For instance, the words usage history for the same code context in the *CCP* predictor may differ if the offset of the word that initiated the miss is different. The *MIBH* technique will further improve the predictability if the relative position of the words accessed in a cache block remains the same, however, the location of the words in the block may differ. For instance, a loop may be accessing all the elements of an array, where each element of the array is a *struct*, and all loop iterations access only the first field in the *struct*. To measure the predictabilities in the *MIBH* technique, a maximum of 8 (for an 8-word cache block) different words usage histories are recorded. When a cache block is evicted, the words usage history that is updated depends not only on its code context but also on the miss initiator word for the cache block. Similarly, on a cache miss, cache noise prediction is made based on the context of the cache access and also on the word offset of the access.

**ORing Previous Two Histories (OPTH):** In the *OPTH* technique, words usage histories of two, instead of one, previously evicted cache blocks are recorded for each context. The prediction is then made by bitwise oring the two words usage histories. The intuition behind the *OPTH* technique is that there may be multiple valid words usage histories for a context, and basing the prediction on just one history can result in a misprediction. The *OPTH* technique will result in better predictability because more words will be brought into the cache for each cache block. However, the *OPTH* technique may reduce the cache utilization by still bringing in more words than required. In fact, we observed that if the number of previous histories that is "ored" for prediction is increased beyond 2, the improvement in utilization reduces significantly.
**Results:** Figure 5 shows the predictabilities of the 28 and 30 bits *CCP* predictor with the *MIBH* and the *OPTH* techniques. Figure 5 shows that the *MIBH* technique performs much better than the *OPTH* technique. The average prediction accuracy for the *CCP* predictor with the *MIBH* technique is about 90% for the integer benchmarks and almost 100% for the floating point benchmarks (the total average is about 95%). The best average prediction accuracy for the *PCP* predictor with both the *OPTH* and the *MIBH* techniques was about 68% and for the *MCP* predictor was about 75%, compared to 95% for the *CCP* predictor. Henceforth, we only consider the *CCP* predictor. Combination of the *MIBH* and the *OPTH* techniques gives the best performance, but only slightly higher than the *MIBH* technique. Henceforth, we only consider the 28-bit CCP predictor with the MIBH technique, as *OPTH* + *MIBH* results in lower cache utilization than just *MIBH*. With this high prediction accuracy, we observed that the cache utilization can potentially increase from about 57% to about 92%.

## 3. CCP Cache Noise Predictor Implementation

With the cache noise predictors, the cache miss request is also accompanied by the pattern of words (if available) in the cache block to be fetched. If the words usage history is not available, then all the words in the cache block are fetched. Each cache block in the cache is provided with 2 bit-vectors (of sizes equal to the number of words in the cache block). The *valid words* bit-vector indicates the valid words in the cache block, and the *words usage* bit-vector records the words accessed in the cache block. For every cache block access, the bit corresponding to the offset of the access is set in the *words usage* bit-vector. In addition, each cache block is also provided with space to store the code context and the miss initiator word offset (for the MIBH technique) for the cache block. When the cache is accessed, the hit or a miss depends on the validity of the cache block, the result of the tag comparison, and also the validity of the word accessed. If the cache access is a miss because the word is invalid, then only the remaining words are brought into the cache. The rest of the discussion only considers cache misses due to invalid cache blocks or due to tag mismatch. In a writeback cache, only the valid words are written back into the lower level cache. In a write through cache, cache miss on a write to an invalid word can be implemented with a write-allocate or with a write-no-allocate policy. Our L1 data cache uses writeback with write allocate policy.
In the predictor, the most recently seen code contexts and their words usage histories are maintained in a predictor table. On a cache miss the words usage history (if available) of the code context is read from the table and sent to the lower level memory. When a

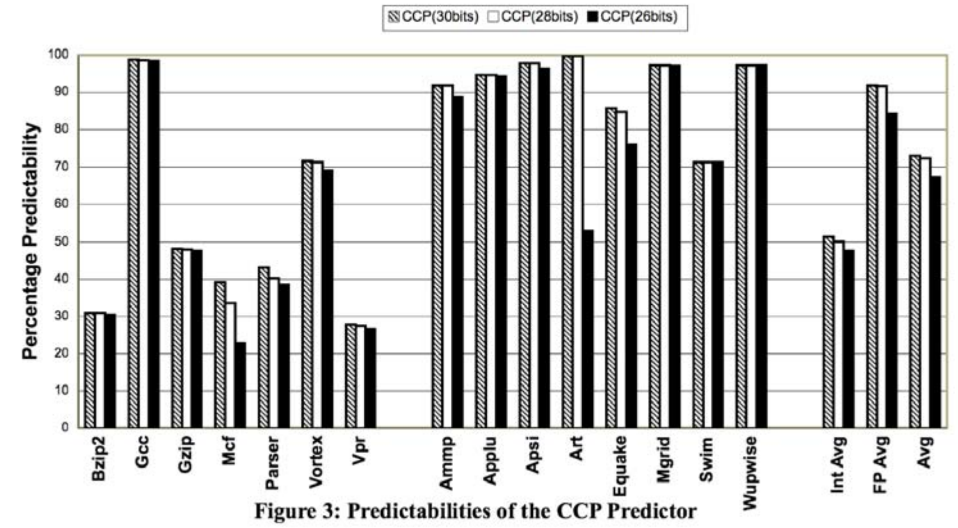


**Figure 3: Predictabilities of the CCP Predictor**

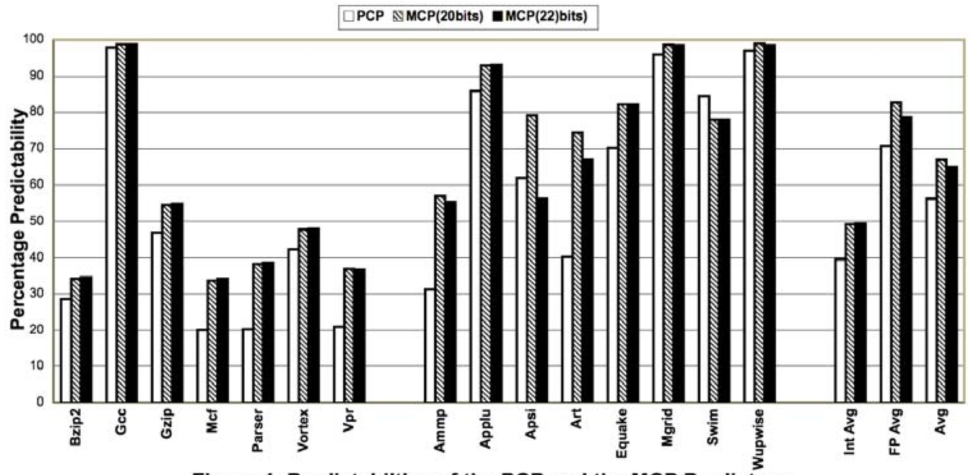


**Figure 4: Predictabilties of the PCP and the MCP Predictors**

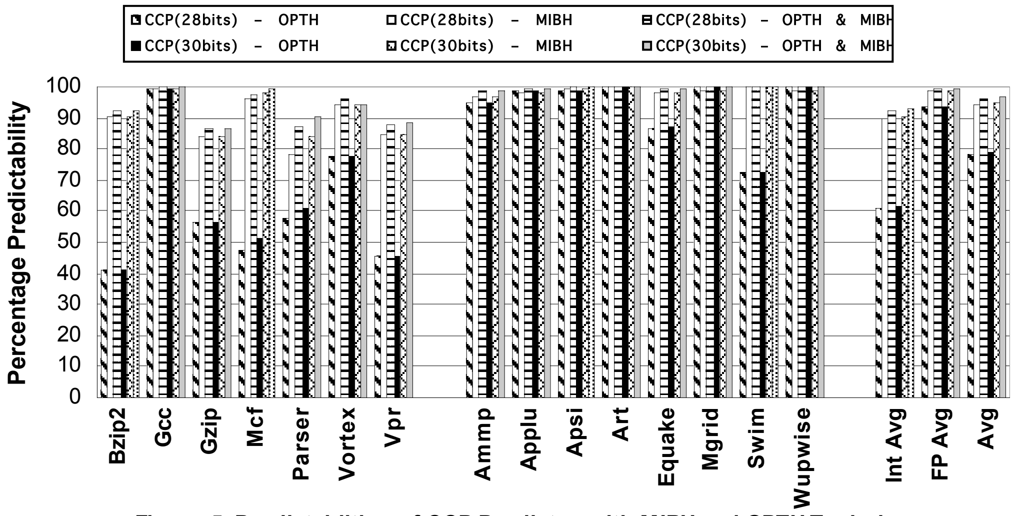


**Figure 5: Predictabilties of CCP Predictor with MIBH and OPTH Techniques**

valid block is evicted from the cache, the table is updated with the words usage history of the evicted cache block. The code context table can be implemented as a direct-mapped, set associative, or a fully associative CAM/RAM structure.

The CAM portion stores the code contexts, and the RAM structure stores the words usage histories. The code context of the cache miss or of the evicted cache block is broadcast in the CAM structure. This is followed by comparisons of the valid code contexts. On a match, the corresponding words usage history is either used or updated. The replacement policy used is the Least Recently Used (LRU) policy. The *CCP* predictor can be alternately implemented by storing the code context along with the instruction cache.

For the *MIBH* technique, each entry in the context table stores multiple words usage histories for the different *miss initiator words*. However, if space is provided for the histories of only a few *miss initiator words*, then each history also stores the initiator word offset. In this case, *miss initiator word offsets (MIWOs)* are also compared along with the code contexts to determine a hit in the table. Figure 6(a) shows the structure of the predictor table entry for the base case, and Figure 6(b) with the *MIBH* technique (with space for just two *miss initiator words*). In Figure 6(b), each CAM portion of the entry has a single broadcast tag, which is used to broadcast the context and the *MIWO*. The comparison result and the validity of the entry drive the word-line of the RAM portion of the structure, which has a single read/write port. If the word-line is activated, then either the history on the port is written into the entry or the history is read from the entry. The context table requires only a single broadcast tag and a single read/write port, because the table is accessed only on a cache miss and on eviction of a cache block. This also prevents the access to the context table from being in the critical path. On a rare case that there are multiple accesses to the table in a single cycle, either the requests can be arbitrarily dropped or a small buffer can be maintained to serialize the requests. We observed that a 2-entry buffer is enough to avoid any loss of requests. Additionally, we did not observe any variation in the performance of the predictor with the two alternatives.

**Better Space Utilizing *MIBH* Predictor:** In the *MIBH* predictor of Figure 6(b), predictor space will be wasted if a code context has only a few *miss initiator words*. To avoid wastage of predictor space, we discuss a novel code context predictor shown in Figure 6(c), where each table entry is provided with space for two *MIWOs*. In this predictor, each context is divided into two parts: *common context* and *different context*. *Common context* is the upper order bits of the context, and *different context* is the remaining lower order bits. In Figure 6(c), each context table entry can hold histories of two contexts provided that they have the same common context. For a hit in the table, the *common context*, the *different context*, and the *MIWO* of a cache block should match with the values in an entry. The number of bits required to implement this MIBH predictor will be slightly more than that required for the predictor in Figure 6(b), which will depend on the size of the *common context*. The replacement policy replaces the least recently used entry, and if the LRU entry has the same common context, then the least recently used history in that entry is replaced.

**Predictor size:** A larger table will have a lower "no prediction" rate resulting in better cache utilization and lower cache energy, however at the expense of a higher predictor table energy consumption. Note that performance is only impacted by the misprediction rate and not by the "no prediction" rate. We only use small predictors in our experiments that may increase the "no prediction" rate, but will limit the energy consumption in the predictor. We measured the percentage of correct, mis, and no predictions (out of the cache blocks fetched) for predictor tables with 16 entries with 4 MIWOs (16/4), 16 entries with 8 MIWOs (16/8), and 32 entries with 4 MIWOs (32/4). The better utilizing MIBH predictor uses a common context of 25 bits and a different context of 3 bits. The experimental parameters are given in Table 1. We observed that, for the 16/4 predictor, the percentage of cache misses for which prediction is made by the better space utilizing predictor of Figure 6(c) is about 10% more than that by the conventional predictor of Figure 6(b). The 16/4 predictor predicted the pattern for almost 80% of the cache misses. We also observed that a 64/4 predictor is enough to predict for almost all the cache misses.

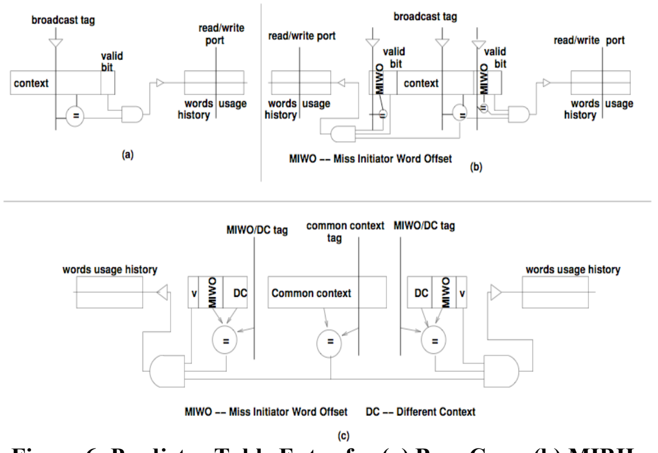


**Figure 6: Predictor Table Entry for (a) Base Case; (b) MIBH; (c) Better Space Utilizing MIBH**

**First Access Check (FAC):** To further reduce the misprediction rate, the *FAC* checks whether the *miss initiator word* (for which a cache miss has occurred) is in the predicted words usage history for the block. If the miss initiator word is not in the history, then either all words in the cache block are fetched or the words indicated by the predicted history plus the miss initiator word are fetched. We follow the first alternative where all the words in the cache block are brought into the cache. For instance, if the predicted words usage history for a block is "0110" and the miss initiator word is the first word in the block then the entire cache block is brought.

**Early Misprediction Detection (EMD):** Another alternative to reduce the misprediction rate is to avoid successive mispredictions due to stale or outdated words usage history. Since the predictor table is updated at eviction time, the words usage history in the table could be stale or outdated, as discussed in Section 2. The EMD technique is implemented on top of the FAC technique. In the EMD technique, once a misprediction is detected (which could be for the miss initiator word), the words usage history for that context and *MIWO* is invalidated in the predictor table to prevent other cache misses from using the outdated/stale history. However, in this technique a single glitch in the words usage history can result in converting correct predictions to "no predictions".

We experimented with the CCP predictors with both FAC and FAC/EMD techniques, and observed that the EMD technique does not result in significant reduction in misprediction rate. However, the FAC reduced the misprediction rate by more than 50%. Figure 7 presents the prediction accuracy results of the CCP predictor with the FAC technique for the 16/4, 16/8, and 32/4 CCP predictors. Figure 7 shows that the 16/4 and 16/8 techniques are able to predict for almost 75% of the cache misses, and the correct predictions are about 97% (out of the total predicted). In the rest of the paper, we experiment with the 16/4 predictor table.

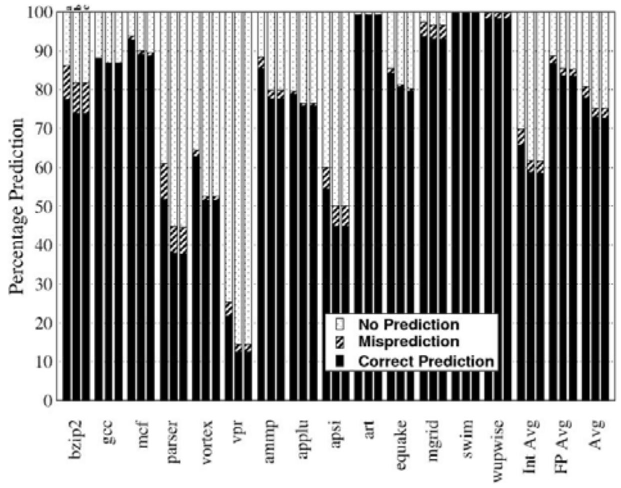


**Figure 7: Prediction accuracies with (a) 32/4; (b) 16/8; (c) 16/4 CCP predictors**

## 4. Related Work

Sub-blocked caches reduce the cost of tagging a cache with small blocks at cache block granularity by associating a single tag with a cache block consisting of multiple sub-blocks. The sub-blocks have their own status bits. With sub-blocking, memory traffic is reduced by transferring only a single sub-block on a cache miss. Seznec [16] proposed decoupled sectored caches (DSC), where the address tag location associated with a cache sub-block is dynamically chosen among several possible tag locations, emulating variable cache block sizes. However, the non-fetched sub-blocks within a cache block result in higher miss rates. With our prediction mechanisms, the cache can be designed as a single-word sub-blocked cache. However, the important difference is that all (and only) the required sub-blocks are brought into the cache simultaneously.

Veidenbaum et al. [18] propose adapting the cache line size to application behavior by using the words usage history of a cache line. However, unlike our scheme, their scheme still brings contiguous words (which can result in cache noise in the cache), the cache line size can only be halved or doubled, and only uses the words usage history of a cache line when bringing the same cache line again. Zhang et al. [20] propose a different implementation of a sub-blocked cache, where each cache line is of size equal to the sub-block and a counter in the cache controller specifies the number of contiguous sub-blocks to be read from the memory. Inoue et al. [9] propose a sub-blocks usage history based fetching of sub-blocks forming a cache line. Their scheme is somewhat similar to our PCP predictor, with the major difference being that they use the sub-block usage history of the cache block that is currently being evicted. For this, on a cache miss, they first determine which cache block will be replaced, and then send the sub-block usage pattern for that block to bring the corresponding sub-blocks in the new cache block. This can increase the cache miss penalty. Dubnicki and Leblanc [5] proposed adjustable block size caches to prevent false sharing in shared memory multiprocessor systems.

Johnson et al. introduce the notion of a macroblock of memory where the memory access pattern is consistent across the macroblock. This concept is somewhat similar to the memory context discussed in this paper. However, they use the access pattern within a macroblock for optimal placement of data in the memory hierarchy (for instance bypassing the data cache) [10], and for varying the size of the cache block [11], where multiple adjacent cache blocks are fetched if the adjacent cache blocks in a macroblock are accessed simultaneously. The main purpose of their study was to reduce the conflict misses. Their techniques can still result in lower cache utilization as they still bring contiguous data into the cache. Kumar and Wilkerson [12] reduce the miss rate of the decoupled sectored cache by prefetching cache lines based on the spatial footprint of a sector. Chen et al. [21] use prediction of to-be-referenced sub-blocks to save leakage energy by turning off the sub-blocks that will not be referenced. They extend the prediction mechanism to also selectively prefetch cache lines based on the usage history of these lines.

Nicolaescu [14] uses profile information to guide a compiler in inserting line size configuration information into a program. Witchel [19] proposed a software-controlled cache line size, where the compiler specifies how much data to fetch on a data cache miss. Several studies [7, 8, 15, 17] have also been performed to measure the trade-offs between miss ratio and traffic ratio by varying block and sub-block sizes. McNiven et al. [13] also looked at reducing traffic between adjacent levels of the memory hierarchy.

In summary, to our knowledge, no techniques have been proposed that attempt to reduce the cache noise by predicting and fetching all and only the required words in a cache block. Techniques have been proposed that vary the cache line size to match the spatial locality of an application, but the main difference compared to our approach is that they still bring in contiguous words into the cache which can still result in cache pollution. In addition, the techniques can only increase the cache line size if the adjacent sub-blocks (smaller cache lines) are accessed simultaneously.

## 5. Experimental Setup and Results

### 5.1 Experimental Setup

We use a modified version of the SimpleScalar simulator [3], simulating a 32-bit PISA architecture. For benchmarks, we use a collection of 6 integer (vpr, mcf, parser, bzip2, vortex, and gcc), and 8 FP (equake, ammp, mgrid, swim, wupwise, applu, apsi, and art) benchmarks, using ref inputs, from the SPEC2K benchmark suite. The statistics are collected for 500M instructions, after skipping the first 3B instructions. In our experiments, we focus only on L1 data cache. Table 1 gives the hardware parameters for the experiments. Our *CCP* predictor has a *common context* of 25 upper order PC bits for a context of 28 bits. The results are only for a fully associative predictor table with 16 entries with 4 *MIWO*s.

### 5.2 Results

We present the L1 data cache miss rates, the bandwidth requirement in terms of the average number of words fetched per cache block, Instructions per cycle (IPC) count, and the cache utilization of the L1 data cache in Table 2. Table 2 shows that the miss rate increases minimally due to mispredictions (an average of about 0.1%), resulting in almost zero reduction in IPC. On the other hand, the bandwidth requirement in terms of average number of words fetched per cache block reduces by about 28%, and the L1 cache utilization increases by about 37%. Words fetched per cache block reduce significantly for benchmarks (such as art, mcf, and bzip2) that have lower cache utilization for the base case and significantly higher utilization with cache noise prediction. In addition, for benchmarks such as vpr, a large percentage of no prediction results in almost no improvement in cache utilization.

### 5.3 Power Measurements

By fetching only the to-be-referenced words in a cache block, cache energy consumption can be reduced, by reducing the energy consumed in reading the words from the lower level caches, in the bus to transfer the words to the upper level caches, and in writing the words in the upper level cache. Furthermore, writing only the valid words back into the lower level cache for a dirty cache block, and shutting down invalid portions of cache blocks will also result in reducing the energy consumption. The disadvantage of this method is the additional energy consumption in the predictor and additional bits in the cache. This is in addition to the savings in leakage energy consumption obtained due to a reduction in the number of active transistors. In this section, we only measure the savings obtained in dynamic energy consumption. Figure 8 shows the percentage reduction in dynamic energy consumption of the new cache + predictor as compared to the conventional cache. We use a modified version of the cacti tool [22](0.18um) to perform the energy consumption of each access to the cache and the predictor table. Note that Figure 8 also includes the energy consumption on cache hits and in the additional bits. Figure 8 shows that an average of about 23% savings in dynamic energy consumption can be obtained with cache noise prediction. Higher energy savings is obtained for benchmarks (such as bzip2, mcf, and art) whose cache utilization increases significantly with the use of a cache noise prediction. On the other hand, negative energy savings are obtained for benchmarks (such as gcc, applu, mgrid, and wupwise) that are using most of the words in a cache block and have higher cache utilization for the base case. Hence, the prediction overhead results in higher energy consumption in these benchmarks.

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| *Fetch/Commit Width* | 8 instructions | *Floating-point FUs* | 3 ALU, 1 Mul/Div |
| *Unified Register File* | 128 Int/128 FP 1 cycle inter-subsys. Lat | *Integer FUs* | 4 ALU, 2 AGU, 1 Mul/Div |
| *Issue Width* | 5 Int/ 3 FP | *Issue Queue Size* | 96 Int/64 FP |
| *Branch Predictor* | 4K Gshare | *BTB Size* | 4K entries, 2 way assoc. |
| *L1 Icache* | 16K direct mapped 2 cycle latency | *L1 Dcache* | 16K, 4-way assoc., 2 cycle latency |
| *Memory Latency* | 100 cycles 1st word, 2 cycle inter-word | *L2 cache* | Unified 512K 8-way assoc. 10 cycle lat. |
| *ROB Size* | 256 instructions | *Load/store buffer* | 64 entries |

**Table 1: Default Experimental Parameters**

| | IPC | | Miss Rate | | Words/Block | | Utilization | |
|---|---|---|---|---|---|---|---|---|
| | Base | CCP | Base | CCP | Base | CCP | Base | CCP |
| Bzip2 | 1.029 | 1.02 | 4.1 | 4.4 | 8 | 3.6 | 27.9 | 61.3 |
| Gcc | 0.942 | 0.94 | 10.4 | 10.4 | 8 | 7.9 | 98.4 | 98.5 |
| Mcf | 0.733 | 0.73 | 24.5 | 24.7 | 8 | 2.6 | 25.1 | 76.1 |
| Parser | 0.935 | 0.93 | 4.3 | 4.6 | 8 | 5.6 | 29. | 42.2 |
| Vortex | 0.994 | 0.99 | 1.7 | 1.7 | 8 | 5.1 | 30.3 | 47.0 |
| Vpr | 0.957 | 0.95 | 5.7 | 5.8 | 8 | 7.2 | 31.4 | 34.8 |
| | | | | | | | | |
| Ammp | 1.048 | 1.04 | 7.2 | 7.3 | 8 | 4.7 | 50.6 | 85.0 |
| Applu | 1.081 | 1.08 | 3.0 | 3.0 | 8 | 7.8 | 90.8 | 92.0 |
| Apsi | 1.100 | 1.10 | 1.5 | 1. | 8 | 6.1 | 41.8 | 54.4 |
| Art | 1.006 | 1.00 | 45.9 | 45.9 | 8 | 2.6 | 32.6 | 98.1 |
| Equake | 1.058 | 1.05 | 6.5 | 6.6 | 8 | 5.2 | 53.1 | 80.6 |
| Mgrid | 1.068 | 1.06 | 3.7 | 3.8 | 8 | 7.6 | 94.0 | 98.3 |
| Swim | 1.062 | 1.06 | 1 | 14.0 | 8 | 6.2 | 77.7 | 99.8 |
| Wupwise | 1.052 | 1.05 | 0.5 | 0.5 | 8 | 7.9 | 96.6 | 99.2 |
| | | | | | | | | |
| Int Avg | 0.93 | 0.93 | 8.50 | 8.6 | 8 | 5.38 | 40.5 | 60.03 |
| FP Avg | 1.05 | 1.05 | 10.32 | 10.38 | 8 | 6.067 | 67.19 | 88.47 |
| Average | 1.00 | 1.00 | 9.54 | 9.64 | 8 | 5.77 | 55.76 | 76.28 |

**Table 2: IPC, Miss rate; Bandwidth in terms of words per cache block; and cache utilization for the 16/4 CCP predictor**

## 6 Hardware Prefetching

Processors employ hardware prefetching to improve the cache miss rates of L1 caches. The most common hardware prefetching mechanism is to also fetch the next cache block on a cache miss to exploit more spatial locality. Prefetching reduces the miss rate, however at the expense of increased cache pollution and increased pressure on the bandwidth. Cache noise prediction can alleviate these two limitations of the prefetching mechanisms. When applying cache noise prediction to a prefetching mechanism, only the required words are fetched from the prefetched cache block. Note that this is different from predicting and prefetching only the to-be-referenced cache blocks as discussed in [21].

With prefetching, the prefetched cache block has not been accessed in the current cache miss. Hence, its code context and its *MIWO* are not known. One alternative is to also predict the word pattern for the prefetched cache block, either using a separate predictor table or using the same predictor table with minor modifications. In our current implementation, the fetched and the prefetched cache blocks are predicted to have the same pattern of words. We experiment with two variants. In one approach, the prefetched cache block does not update the predictor table when it is evicted. In the other approach, the prefetched cache block is updated with its code context and *MIWO* when it is accessed for the first time. The cache block then updates the predictor table on eviction benefiting future misses in its code context. We experiment with prefetching all the words of the next cache block (base case) and cache noise prediction based prefetching with the above two variants.

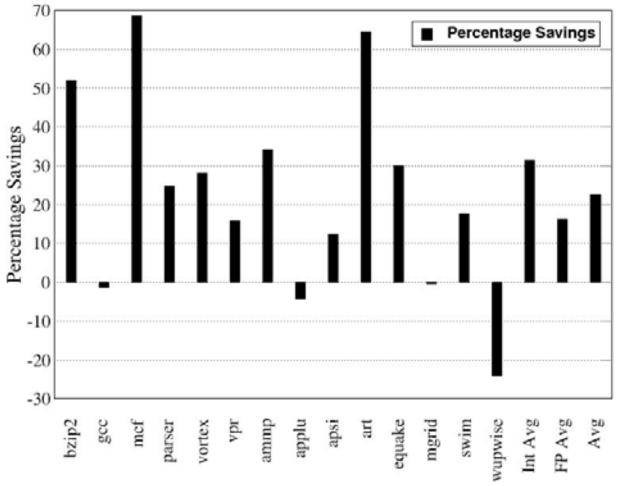


**Figure 8: Percentage dynamic energy savings of predictor + cache wrt to the traditional cache**

Figure 9 shows the prediction accuracy results for the cache noise based hardware prefetching along with the prediction accuracies without hardware prefetching. The results are for the 16/4 predictor. It can be observed that the prediction rate reduces with prefetching. This is because cache misses reduce with hardware prefetching. We observed that the reduction in cache misses that hit in the predictor table is more than the cache misses that miss, resulting in lower prediction rate. Nevertheless, the prediction accuracy of cache noise prediction is still very high, an average of about 93% when the prefetched cache block does not update the predictor table and about 96% when the prefetched cache block updates the predictor table. When comparing the two variants of prediction, the prediction rate usually increases and the misprediction rate usually decreases when the prefetched cache block updates the predictor table. This is because a more up-to-date words usage history for the context of the prefetched cache block is now available. In some benchmarks, especially gcc, the prediction rate decreases when the prefetched cache block updates the predictor because the updated history evicts critical history and never gets used. Figure 10 gives percentage decrease in energy consumption and percentage increase in cache utilization with respect to the base case prefetching mechanism. It can be observed that, when the prefetched cache block updates the predictor, cache energy consumption reduces by about 22% and the cache utilization increases by about 70%. We also observed that the miss rate increased by only about 2% with respect to the base cache prefetching mechanism.

## 7 Cache noise prediction as an alternative to sub-blocking

Sub-blocking is used to have the lower tag overhead and higher spatial locality exploitation benefits of a larger cache block, while reducing the higher cache noise and bandwidth requirements associated with them. These are also the benefits of cache noise prediction based fetching of words. In this section, we investigate the use of cache noise prediction based fetching of words as an alternative to sub-blocking. For this, we compare the miss rates, cache utilization, bandwidth requirements, and energy consumption of a cache using the cache noise predictor with that of a sub-blocked cache. We use the 16/4 prediction for the prediction based fetching of words, and sub-blocks of two words for the sub-blocked cache. Table 3 gives the results of our measurements. The energy measurements in Table 3 are percentage reduction in energy with respect to a conventional non-sub-blocked 32-byte cache. Table 3 shows that the average bandwidth requirement of the sub-blocked cache is about 35% of that of the prediction based approach. However, the energy consumption and the miss rate of sub-blocked cache is about 44% and 97% more, respectively, than the prediction based approach. In addition, the cache utilization of the prediction based approach is about 10% more than the sub-blocked cache. This suggests that cache noise prediction based approach can be a good alternative to sub-blocking if slightly higher bandwidth is available.

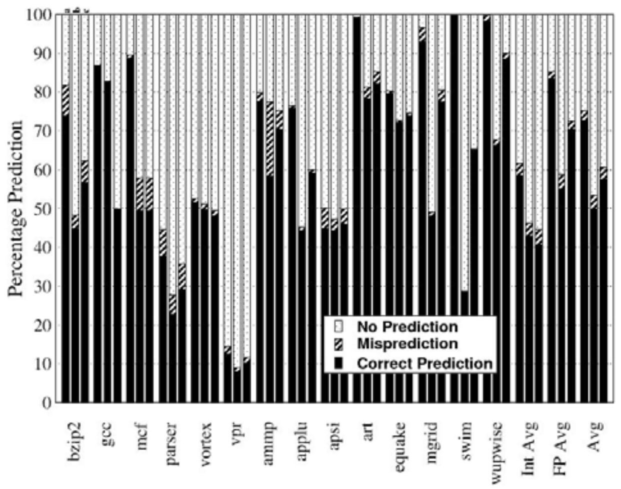


**Figure 9: Prediction accuracies with (a) no prefetching; (b) no updates for prefetched block; and (c) updates for prefetched block; for the 16/4 predictor with 32-byte block size**

## 8 Profiling to improve total prediction

For some of the benchmarks, especially vpr, parser, vortex, and apsi, the number of predictions is significantly low, inspite ot their higher predictabilities as shown in Figure 3. The primary reason for this decrease in prediction rate is that the predictor table is too small to accommodate all the words usage histories. An alternative to increasing the predictor table size to alleviate this limitation of small predictor tables is to use the table more efficiently. In particular, there may be many word usage histories in the predictor that are evicted without getting used. However, when the predictor table is updated with these histories, they could replace histories that may be required in the future (even with LRU replacement policy). In this section, we investigate using profiling to improve the efficiency of the predictor table. Figure 11(x) shows the distribution of histories based on its usage for a 32-byte cache block and the 16/4 predictor table. The figure shows that there are a significant percentage of histories that are evicted either without getting used or with getting used only a few times, especially for benchmarks with lower prediction rates. Hence, we explore using profile information to avoid allocation of predictor table entries to those histories that are not used. For this, we run the benchmarks once to collect the combination of code contexts and *MIWO*s for the histories that are not used and utilize the collected information to prevent them from updating the predictor table in another run. The prediction results are shown in Figure 11 (y). Figure 11(y) shows that profiling improves the prediction rate by about 7%. However, the improvement in prediction rate is not as high as expected, suggesting that the predictor table may have to be increased to get better prediction rates.

## 9 Sensitivity

In this section, we present the prediction accuracy results of the 16/4 *CCP with FAC* predictor as the cache block size is increased to 64 bytes and 128 bytes. The increase in cache block size can have two implications on the prediction accuracy: (i) the predictor misprediction rate can increase because of an increase in the probability that a word is used that is not present in the predicted words usage history, (ii) the predictor "no prediction rate" can increase because of an increase in the probability that a new cache block brought into the cache has a *miss initiator word* for which the words usage history has not been recorded. Figure 12 gives the prediction results for the different sized cache blocks. As expected, the "no prediction" rate increases with an increase in the cache block size, suggesting that larger predictor tables may be required for larger block sizes. This also suggests that profiling will be much more beneficial for larger cache blocks. On the other hand, the increase in misprediction rate is negligible, an average of about 1%, with wupwise being the only benchmark with a noticeable increase in misprediction rate (by about 20% for 128-byte block size). Hence, cache miss rate will remain largely unaffected because the miss rate is only affected by the misprediction rate of cache noise prediction. Even with decreased prediction rate of the cache noise predictor, the cache utilization increased by about 79% for 64-byte cache block and by about 91% for 128-byte cache block.

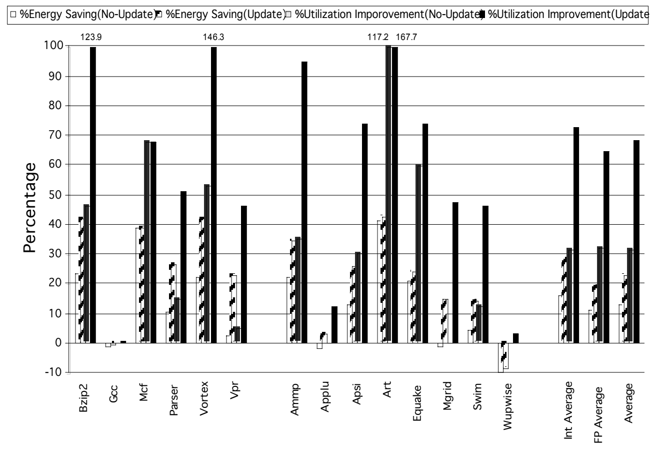


**Figure 10: Percentage reduction in energy consumption and percentage improvement in cache utilization for hardware prefetching with 16/4 CCP cache noise predictor**

| | Energy | | Miss Rate | | Words/Block | | Utilization | |
|---|---|---|---|---|---|---|---|---|
| | **Sub-block** | **CCP** | **Sub-block** | **CCP** | **Sub-block** | **CCP** | **Sub-block** | **CCP** |
| Bzip2 | 46.92 | 52.01 | 8.96 | 4.48 | 2 | 3.64 | 60.8 | 61.36 |
| Gcc | 26.14 | -1.17 | 27.81 | 10.41 | 2 | 7.99 | 74.13 | 98.52 |
| Mcf | 56.69 | 68.59 | 45.85 | 24.79 | 2 | 2.64 | 54.95 | 76.19 |
| Parser | 35.44 | 24.56 | 10.63 | 4.67 | 2 | 5.66 | 60.99 | 42.27 |
| Vortex | 34.57 | 28.15 | 4.27 | 1.76 | 2 | 5.16 | 49.36 | 47.03 |
| Vpr | 37.03 | 15.91 | 14.14 | 5.85 | 2 | 7.23 | 61.99 | 34.85 |
| | | | | | | | | |
| Ammp | 45.16 | 34.02 | 15.2 | 7.37 | 2 | 4.76 | 75.11 | 85.08 |
| Applu | -12.87 | -4.29 | 13.17 | 3.09 | 2 | 7.89 | 77.98 | 92.08 |
| Apsi | 42.54 | 12.36 | 3.33 | 1.6 | 2 | 6.15 | 87.65 | 54.44 |
| Art | 64.01 | 64.36 | 61.24 | 45.95 | 2 | 2.66 | 80.57 | 98.19 |
| Equake | 39.31 | 29.93 | 14.35 | 6.61 | 2 | 5.28 | 69.42 | 80.61 |
| Mgrid | -19.04 | -0.32 | 17.56 | 3.88 | 2 | 7.65 | 73.62 | 98.31 |
| Swim | 47.31 | 17.64 | 27.05 | 14.02 | 2 | 6.23 | 69.31 | 99.86 |
| Wupwise | 13.58 | -23.92 | 1.88 | 0.59 | 2 | 7.92 | 73.82 | 99.25 |
| | | | | | | | | |
| Int Avg | 39.47 | 31.34 | 18.61 | 8.66 | 2 | 5.387 | 60.37 | 60.037 |
| FP Avg | 27.5 | 16.22 | 19.22 | 10.389 | 2 | 6.0675 | 75.94 | 88.478 |
| Average | 32.63 | 22.7 | 18.96 | 9.648 | 2 | 5.776 | 69.26 | 76.289 |

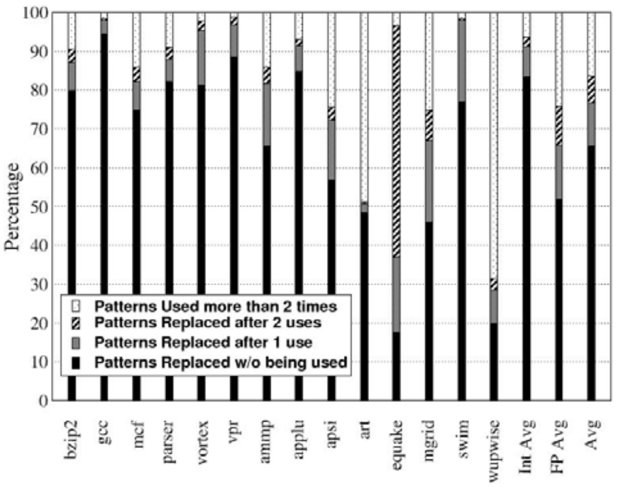
**Table 3: Miss rate; Bandwidth in terms of words per cache block; power consumption overhead and cache utilization for the 16/4 CCP predictor compared with a sub-blocked cache**

## 10 Conclusion

In this paper, we proposed prediction mechanisms to reduce the number of words that are fetched into the cache and never used. The prediction mechanism uses the words usage history of the last evicted cache block. We experimented with three different kinds of predictors – *PCP*, *MCP*, and *CCP* – and observed that the predictability of the code context predictor (*CCP*) is the best. With the help of predictability improving techniques such as *MIBH*, the *CCP* predictor had about 95% correct prediction for an infinite predictor. We also explored techniques to improve the prediction rate of smaller predictor tables (better space utilizing *MIBH*), and reduce the misprediction rate of cache noise prediction (*FAC*). With all these techniques, even a small predictor CCP predictor with 16 entries and space for only four miss initiator words' usage histories (16/4 predictor), achieved a prediction rate of 75% with correct prediction of about 97% out of the total predictions.

The 16/4 predictor improved the cache utilization of the L1 data cache by about 37%, reduced the bandwidth requirement between the L1 and the L2 caches by about 28%, and reduced the cache energy consumption (including the consumption in the additional hardware) by about 23%. All these benefits were obtained with a minimal 0.1% increase in cache miss rate and almost zero impact on IPC. We also investigated the applicability of cache noise prediction mechanisms for hardware prefetching, and observed that prediction accuracy of about 96%(out-of total prediction) was achieved with hardware prefetching as well. However, about 70% cache utilization improvement is observed for the hardware prefetching mechanism. When compared to sub-blocking with two word sub-blocks, cache noise prediction based fetching of words improves miss rate and cache utilization by 97% and 10%, respectively, and reduces energy consumption by 44%. We also explored using profiling to improve the cache noise predictor efficiency.

(x)



(y)

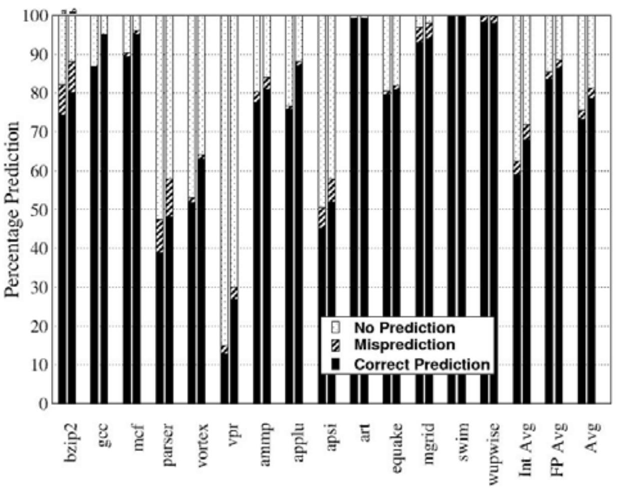


**Figure 11: (x) Percentage distribution of histories in the predictor w.r.t their usage; (y) Prediction accuracies (a) w/o profiling; (b) with profiling w/o updates for not used histories**

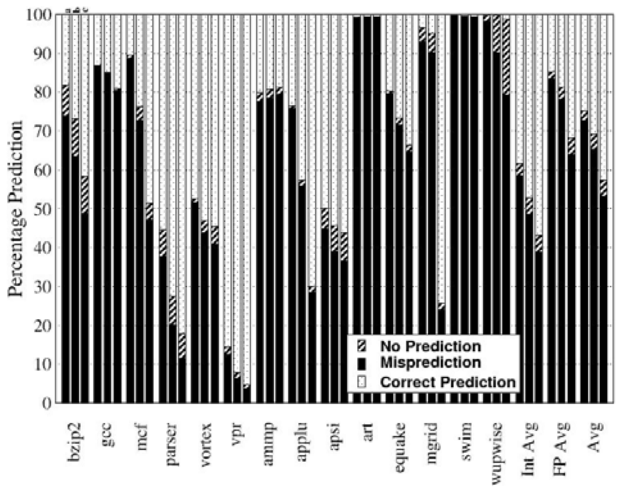


**Figure 12: Prediction results for a 16/4 CCP predictor for cache block size of (a) 32 bytes; (b) 64 bytes; and (c) 128 bytes**

**References**

[1] M. Bekerman, et. al., Correlated Load Address Predictors, Proc. ISCA-26, 1999.

[2] D. Burger, et. al., Memory bandwidth limitations of future microprocessors, Proc. ISCA-23, 1996.

[3] D. Burger and T. Austin, The SimpleScalar Tool Set, Version 2.0, Computer Arch. News, 1997.

[4] A. Dhodapkar and J. Smith, Comparing Program Phase Detection Techniques, Proc. Micro-36, 2003.

[5] C. Dubnicki and T. LeBlanc, Adjustable block size coherent caches, Proc. ISCA-19, 1992.

[6] J. Gonzalez and A. Gonzalez, Speculative execution via address prediction and data prefetching, Proc. ICS, 1997.

[7] M. Hill and A. Smith, Experimental evaluation of on-chip microprocessor cache memories, Proc. ISCA-11, 1984.

[8] A. Huang and J. Shen, A limit study of local memory requirements using value reuse profiles, Micro-28, 1995.

[9] K. Inoue, et. al., Dynamically Variable Line-Size Cache Exploiting High On-Chip Memory Bandwidth of Merged DRAM/Logic LSIs, Proc. HPCA, 1999.

[10] T. Johnson and W. Hwu, Run-time adaptive cache hierarchy management via reference analysis, Proc. ISCA-24, 1997.

[11] T. Johnson, et. al., Run-time spatial locality detection and optimization, Proc. Micro-30, 1997.

[12] S. Kumar and C. Wilkerson, Exploiting spatial locality in data caches using spatial footprints, Proc. ISCA-25, 1998.

[13] G. McNiven and E. Davidson, Analysis of memory referencing behavior for design of local memories, Proc. ISCA-15, 1988.

[14] D. Nicolaescu, et. al., Compiler-directed Cache Line Size Adaptivity, IMS, 2000.

[15] S. Przybylski, The performance impact of block sizes and fetch strategies, Proc. ISCA-20, 1993.

[16] A. Seznec, Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio, Proc. ISCA-21, 1994.

[17] A. Smith, Line(block) size choice for cpu cache memories, ACM Transactions on Computer Systems, C-36:1063-1075, 1987.

[18] A. Veidenbaum, et. al., Adapting Cache Line Size to Application Behavior, Proc. ICS, 1999.

[19] E. Witchel and K. Asanovic, The Span Cache: Software Controlled Tag Checks and Cache Line Size, Proc. ISCA-28, 2001.

[20] C. Zhang, et. al., Energy Benefits of a Configurable Line Size Cache for Embedded Systems, Proc. Int'l Symp. on VLSI, 2003.

[21] C. F. Chen et al., Accurate and complexity-effective spatial pattern prediction, *Proc. HPCA-10*, 2004.

[22] P. Shivakumar and N. Jouppi, CACTI 3.0: An Integrated Cache timing, Power, and Area Model, *Technical Report, DEC Western Lab*, 2002.