Enabling Partial Cache Line Prefetching Through Data Compression *

Youtao Zhang
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083

Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

Hardware prefetching is a simple and effective technique for hiding cache miss latency and thus improving the overall performance. However, it comes with addition of prefetch buffers and causes significant memory traffic increase. In this paper we propose a new prefetching scheme which improves performance without increasing memory traffic or requiring prefetch buffers. We observe that a significant percentage of dynamically appearing values exhibit characteristics that enable their compression using a very simple compression scheme. The bandwidth freed by transferring values from lower levels in memory hierarchy to upper levels in compressed form is used to prefetch additional compressible values. These prefetched values are held in vacant space created in the data cache by storing values in compressed form. Thus, in comparison to other prefetching schemes, our scheme does not introduce prefetch buffers or increase the memory traffic. In comparison to a baseline cache that does not support prefetching, on average, our cache design reduces the memory traffic by 10%, reduces the data cache miss rate by 14%, and speeds up program execution by 7%.

1 Introduction

Due to increasing CPU and memory performance gap, off-chip memory accesses have become increasingly expensive and can take hundreds of cycles to finish. Since load instructions usually reside on the critical path, a single load miss could block all of its dependent instructions and stall the pipeline. To improve the memory performance, hardware prefetching [3, 2, 7, 1] has been proposed for use in high performance computer systems. It overlaps long memory access latency with prior computations such that at the time the data is referenced, it is present in the cache.

Different prefetching approaches vary in *where* they hold the prefetched data, *what* data they prefetch, and *when* they prefetch the data. A prefetch scheme can simply prefetch the next cache line, or with additional hardware support prefetch cache lines with a dynamically decided stride. Since hardware speculatively prefetches data items, these data items may and may not be used by later accesses. In order to avoid the pollution of data caches, prefetched data is usually kept in a separate prefetch buffer. A cache line is moved from the prefetch buffer to the data cache if a memory access references data in the cache line. Since prefetch buffer is of limited size, new prefetched cache lines have to kick out old ones in the buffer if the buffer is full. If a cache line is prefetched too early, it might be replaced by the time it is referenced. On the other hand, if a cache line is prefetched too late, we are unable to fully hide the cache miss latency. If a prefetched cache line is never moved from the prefetch buffer to the data cache, the memory bandwidth used in bringing it into the prefetch buffer is wasted. Although prefetching is a simple and effective technique, it results in increased memory traffic and thus requires greater memory bandwidth.

In this paper we propose a prefetching technique that does not increase memory traffic or memory bandwidth requirements. By transferring values in compressed form, memory bandwidth is freed and whenever possible this extra bandwidth is used to prefetch other compressed values. In addition, the scheme we propose does not require introduction of extra prefetch buffers.

The compression scheme is designed based upon characteristics of dynamically encountered values that were observed in our studies [8, 9, 6]. In particular, dynamic values can be categorized as small values and big values. Positive small values share the prefix of all zeros and negative small values share the prefix of all ones. Also pointer addresses that account for a significant percentage of big values share the same prefix if they are in the same memory chunk of cer-

*Supported by DARPA award no. F29601-00-1-0183 and National Science Foundation grants CCR-0220262, CCR-0208756, CCR-0105535, and EIA-0080123 to the University of Arizona.

tain size. Using small amount of space to remember these prefixes, we can store the values in compressed form and easily reconstruct the original values when they are referenced.

The prefetching scheme works as follows. With each line in memory, another line which acts as the prefetch candidate is associated. When a cache line is fetched, we examine the compressibility of values in the cache line and the associated prefetch candidate line. If the i-th word of the line and the i-th word from its prefetched candidate line are both compressible, the two words are compressed and transferred using up bandwidth of one word. This is done for each pair of corresponding words in the two lines. This approach clearly does not increase the memory bandwidth requirements. However, in general, it results in prefetching of a *partial cache line*. By studying a spectrum of programs from different benchmark suites, we found the compressible words are frequent and prefetching a partial cache line helps to improve the performance. In addition, we derive a parameter to characterize the importance of different cache misses. We found that a cache miss from a compressible word normally blocks more instructions than that from an incompressible word. Thus, prefetching of compressible words shortens the critical path length and improves the processor throughput.

The rest of this paper is organized as follows. We motivate our design by a small example in section 2. Cache details and access sequences are discussed in section 3. Implementation and experimental results are presented in section 4. Related work is reviewed in section 5. Finally, section 6 summarizes our conclusions.

2 Motivation of Partial Cache Line Prefetching

We first discuss the representation of compressed values used by in our hardware design and then illustrate how the cache performance is improved by enabling prefetching of partial caches lines.

2.1 Value Representation

While a 32-bit machine word can represent 2^{32} distinct values, these values are not used equally frequently. Memory addresses, or pointer values, account for a significant percentage of dynamically used values. Recent study shows that dynamically allocated heap objects are often small [11] and by applying different compiler optimization techniques [10, 11] these objects can be grouped together to enhance spatial locality. As a result, most of these pointer values point to reasonably size memory region and many share a common prefix. For non-address values, studies show that many of them are small values, either positive or negative, close to the value zero [9]. The higher order bits of small positive values are all zeros while the higher order bits of small negative values are all ones.

Given the above characteristics of values, it is clear that they can be stored in compressed formats in caches and reconstructed into their uncompressed forms when referenced by the processor. Figure 1(a) shows the case when the prefix of a pointer value can be discarded. If an address pointer stored in memory and the memory address at which the address pointer is stored share a prefix, then the prefix need not be stored in memory. When a shortened pointer is accessed from memory, by concatenating it with the prefix of the address from which the pointer is read, the complete address pointer can be constructed. For example, in Figure 1(a), when we access pointer Q using pointer P, we could use the prefix of pointer P to reconstruct the value of Q. Figure 1(b) shows the case in which the prefix of a small value can be discarded if these bits are simply sign extensions. We save only the sign bit and could extend this bit to all higher order bits when reconstructing the value.

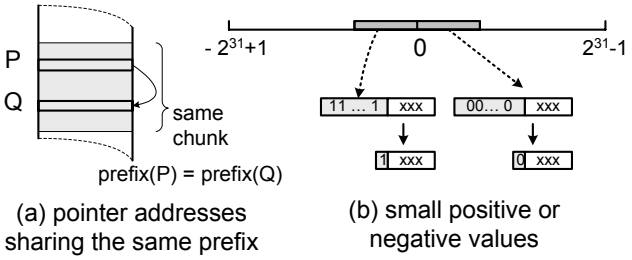

Figure 1. Representing a 32-bit value using fewer bits.

According the above observations compression is achieved by eliminating higher order bits of the values. The next question we must answer is how many of the higher order bits should be eliminated to achieve compression. Through a study of a spectrum of programs we found that compressing a 32 bit value down to 16 bits strikes a good balance between the two competing effects described above [16]. We use the 16th bit to indicate whether the lower order 15 bits represent a small value or a memory address. The remaining 15 bits represent the lower order bits of actual values. Thus, pointers within a 32K memory chunk and small values within the range $[-16384, 16383]$ are compressible.

Figure 2 shows in more detail the value representation we use. A value could be stored in either compressed or uncompressed form and if it is stored in compressed form, it could be a compressed pointer or a compressed small value. Thus, two flags are used for handling compressed values. The flag "VC" indicates whether the stored value is in compressed form. When it is set, which represents a compressed value, the second "VT" flag is used to indicate if the original value is a small value or pointer address. The "VT" flag is stored as part of the compressed value in the cache while the "VC" flag is stored separately from the value.

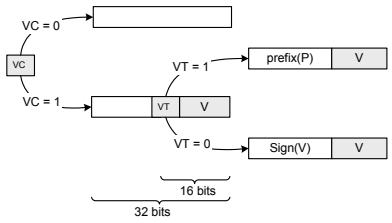

Figure 2. Representing compressed values.

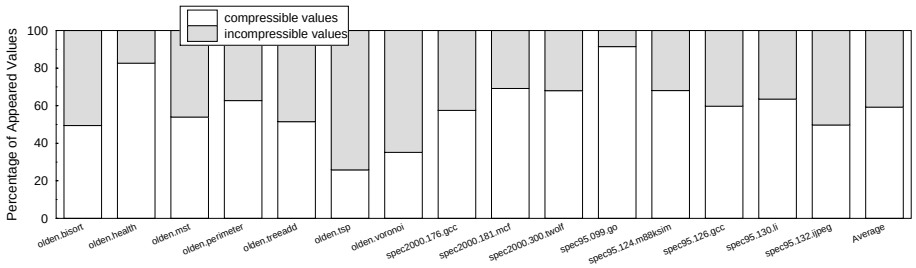

Figure 3. Values encountered in memory accesses.

Albeit the above compression scheme is very simple, it is very effective in practice. We examined all accessed values, from Olden, SPEC95Int, SPEC2000Int benchmark suites, as a result of word level memory accesses and categorized the values as compressible and uncompressible according to our scheme. If the higher order 18 bits are all 0s or 1s, we consider it as a compressible small value. If a value and its address share the same 17-bit prefix, we consider it to be a compressible address pointer. Otherwise, the value is categorized as a non-compressible value. From Figure 3, we see on average, 59% of dynamic accessed values are compressible under the definition of this compression scheme.

2.2 Partial Cache Line Prefetching

Consider the commonly used *prefetch on miss* policy. If a referenced cache line l is not in the cache, line l is loaded into the data cache and line $l+1$ is brought into the prefetch buffer. Thus, the demand on the memory bandwidth is increased. On the other hand, by exploiting the dynamic value representation redundancy, we can perform hardware prefetching which exploits the memory bandwidth saved through data compression. Our method stores values in the cache in compressed form and in the space freed up by compressing values to 16 bits, additional compressible values are prefetched and stored. If a word in a fetched line l at some offset is compressible, and so is the word at the same offset in the prefetched line $l+1$, then the two words are compressed and held in the data cache at that offset. On the other hand, if the word at a given offset in line l or line $l+1$ is not compressible, then only the word from from line l is held in the cache. Thus, when all words in line l are fetched into the cache, some of the words in line $l+1$ are also prefetched into the cache.

Let us consider the example shown in Figure 4 where, for illustration purposes, it is assumed three out of four words are compressible in each cache line. The space made available by compression in each cache line is not enough to hold another cache line. Therefore, we choose to prefetch only part of another line. If the compressible words from another cache line with corresponding offsets are prefetched, then three additional compressible words can be stored which covers 7 out of 8 words from two cache lines.

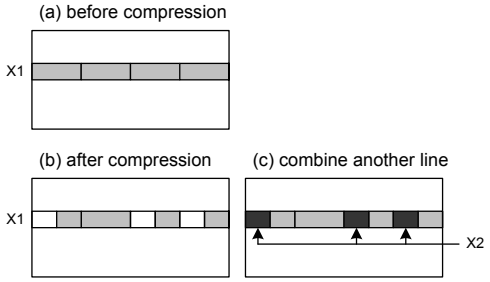

Figure 4. Holding compressed data in cache.

The example in Figure 5 illustrates how compression enabled prefetching can improve performance. Figure 5(b) shows a code fragment that traverses a link list whose node structure is shown in Figure 5(a). The memory allocator would align the address allocation and each node takes one cache line (we assume 16 bytes per line cache). There are 4 fields of which two are pointer addresses, one is a type field and the other one contains a large value. Except for this large information value field, the other three fields are identified as highly compressible. The sample code shown in Figure 5(b) calculates the sum of the information field for all nodes of type T. Without cache line compression, each node takes one cache line. To traverse the list, the next field is followed to access a new node.

```
struct node {
        int type;
        int info;
        struct node *prev;
        struct node *next;
};
```
(a) node declaration

```
        ...
(1)     while ( p ) {
(2)       if (p→type == T)
(3)           sum += p→info;
(4)       p = p→next;
        }
        ...
```
(b) sample code

Figure 5. Dynamic data structure declaration.

A typical access sequence for this piece of code would generate a new cache miss at statement (2) for every iteration of the loop (see Figure 6(a)). All accesses to other fields in the same node fall into the same cache line and thus are all cache hits. However, if all compressible fields are compressed, a cache line would be able to hold one complete node and three fields from another node. Now an access sequence will have cache hits at statements (2) and (4) plus a possible cache miss at statement (3), as shown in Figure 6(b). Partial cache line prefetching can improve perfor-

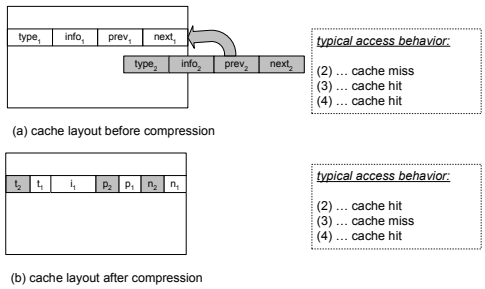

Figure 6. Cache layout before and after compression.

mance in two ways. First, if the node is not of the type T, we do not need to access the large information field. This saves one cache miss. Second, even in the case that we do need to access the information field, the cache miss happens at statement (3). Although the new and old scheme generate the same number of cache misses, the miss at statement (3) is less important. The critical program execution path is "(1)(2)(4)" and (3) is not on this path. Thus, a miss at (3) will have less impact on the overall performance.

3 Cache Design Details

In this section, we will first discuss the new design and then present the fast compression and decompression logic. Handling of data accesses to our new cache design will also be discussed.

3.1 Cache Organization

In this work we consider a two level cache hierarchy. Both L1 and L2 caches are on chip. Moreover partial cache line prefetching is implemented for both caches. At the interface between the CPU and L1 cache compression and decompression is performed so that the CPU always sees values in uncompressed form while the cache stores the values in compressed form. Similarly the off-chip memory holds values in uncompressed form but before these values are transferred on-chip, they are compressed. A value is considered to be compressible if it satisfies either of the following two conditions:

- If the 18 higher order bits are all ones or all zeros, the 17 higher order bits are discarded.
- If the 17 higher order bits are the same as those of the value's address, the 17 higher order bits are discarded.

Compressible words are stored in the cache in their compressed forms. Potentially, one physical cache block could hold content from two lines, identified as the *primary* cache line and the *affiliated* line in the paper. The primary cache line is defined as the line mapped to the physical cache line/set by a normal cache of the same size and associativity. Its affiliated cache line is the unique line that is calculated through a single operation as shown below:

$$< Tag_{affiliated}, Set_{affiliated} >= \\ < Tag_{primary}, Set_{primary} > \oplus mask$$

where $mask$ is a predefined value. The mask is chosen to be 0x1 which means the primary and affiliated cache lines are consecutive lines of data. Thus, this choice of the mask value corresponds to the next line prefetch policy. Accordingly, given a cache line, there are two possible places it can reside in the cache, referred to as the primary location and the affiliated location. The cache access and replacement policy ensure that at most one copy of a cache line is kept in the cache at any time.

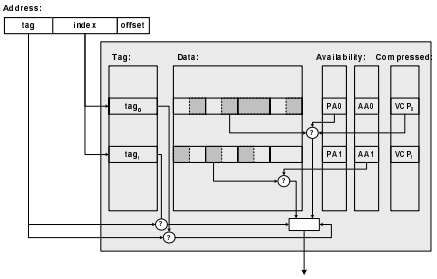

Figure 7. Compression cache.

The major difference between a standard two level cache and the new design is at the interface between the L1 and L2 cache. The requests from the upper level cache are traditionally line based. For example, if there is a miss at the L1 cache, a request for the whole line is issued to the L2 cache. In the compression cache, the requested line might appear as an affiliated line in the L2 cache and thus only be partially present in the L2 cache. To maximize the benefits from the partially prefetched cache line, we do not always enforce a complete line from the L2 cache as long as the requested data item is found. That is, the requests to the L2 cache are still word based and a cache hit at the L2 cache returns a partial cache line. The returned line might be placed as a primary line or an affiliated line. In either case, flags are needed to indicate whether a word is available in the cache line or not. A flag PA (Primary Availability) for the primary cache line is associated with one bit for each word and another flag AA (Affiliated Availability) for the affiliated cache line is provided. As discussed, a value compressibility flag (VC) is used to identify if a value is compressible or not. For the values stored in the primary line, a one-bit VCP flag is associated for each word. On the other hand, if a value can appear in the affiliated line, it must be compressible and thus no extra flag is needed for these values. The design details of the first level compression cache are shown in Figure 7.

When compared to other prefetching schemes, partial cache line prefetching adds 3 bits for every machine word.

It is about 10% cache size increase, however, it completely removes the prefetch buffer. Thus the hardware cost introduced from the extra flags are not high. In the next section, we will compare our scheme to hardware prefetching whose prefetch buffer is of comparable size.

3.2 Dynamic Value Conversion

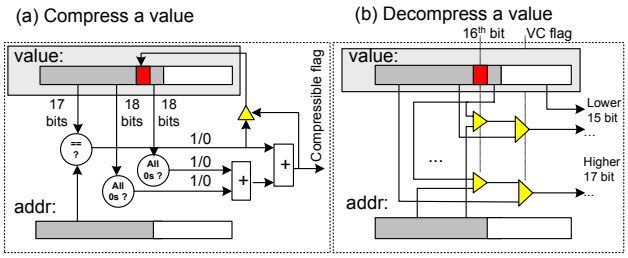

Figure 8. Value Compression and Decompression.

Since a value can dynamically change from an incompressible one to a compressible and vice versa, it is important to support fast compression and decompression. Dynamic values are compressed before writing to L1 cache and decompressed before sending back to CPU. In Figure 8, we present the hardware implementation of the compressor and decompressor. To compress a value, three possible cases are checked in parallel: (i) are the higher order 17 bits of value and address the same; (ii) are the higher order 18 bits all ones; and (iii) are the higher order 18 bits all zeros. Each of the checks can be performed using $log(18) = 5$ levels of 2 input gates. In addition, extra delay is introduced in form of 3 levels of gates to distinguish these cases. The total delay is 8 gate delays. Since compression is associated with write instructions and the data is usually ready before the pipeline reaches the write back stage. As a result, compression delay can be hidden before writing back to the cache.

It is more critical to quickly decompress a value which is associated with a read instruction. As shown in Figure 8(b), we need at least two levels of gates to decompress the higher order 17 bits. Each gate is enabled by a flag input. The delay associated with decompression can be hidden. Typically the delay associated with the reading of the data array is smaller than the delay associated with tag matching. Therefore after data has been read, some time is available to carry out the decompression while the tag matching is still in progress. This approach for hiding decompression delay is essentially similar to the approach used in [15] to hide the delay associated with decoding of read values.

3.3 Cache Operation

Next we discuss further details of how the cache operates. First we describe the interactions at interface points (CPU/L1, L1/L2, and L2/Memory) and next we discuss how the situation where a value stored in a location changes from being compressible to uncompressible is handled.

CPU - L1 interface. When a read request is sent from the CPU, both the primary cache line and its affiliated line are accessed simultaneously. The set index of the primary cache line is flipped to find its affiliated line. If found in the primary cache line, we return the data item in the same cycle and if it is found in the affiliated line, the data item is returned in the next cycle (with one extra cycle latency). A compressed word is decompressed and sent back to CPU. In the case of writing a value to the cache, a write hit in the affiliated cache line will bring the line to its primary place.

L1 - L2 interface. For cache accesses from L1 cache to L2 cache, if the accessed word is available in L2, it is a cache hit and only the available words in the cache line are returned. Since the block size of L2 cache is 2 times that of L1 cache, the primary and affiliated cache line in L1 cache reside in the same cache line block in L2 cache. Since they are already organized in their compressed format, words from the affiliated line are returned only when they and their corresponding words in primary line are compressible.

When a new cache line arrives to the L1 from L2 cache, the prefetched affiliated line is discarded if it is already in the cache (it must be in its primary place in this situation). On the other hand, before discarding a replaced cache line, we check to see if it is possible to put the line into its affiliated place. If the dirty bit is set, we still write back its contents and only keep a clean partial copy in its affiliated place.

L2 - memory interface. For accesses from L2 cache to memory, both the primary and the affiliated lines are fetched. However, before returning the data, the cache lines are compressed and only available places from the primary line are used to store the compressible items from the affiliated line. The memory bandwidth is still the same as before. The arrival of a new line to the L2 cache is handled in a manner similar to the arrival of a new cache line to L1 cache.

Changes in values from compressible to uncompressible. When a word in primary cache line changes from a compressible word to an incompressible word, and the corresponding word in affiliated cache line already resides in the word, we have a choice between keeping either the primary line or the affiliated line in the cache line. Our scheme gives priority to the words from the primary line. The words from affiliated line are evicted. The affiliated line must be written back if the dirty bit is set.

When a word in an affiliated line changes from compressible to incompressible, we move the line to its primary place and update its corresponding word. The effect is the same as that of bringing a prefetched cache line into the cache from the prefetch buffer in a traditional cache.

It might increase memory traffic if value changes fre-

quently between these two categories. However, our experimental results show dynamic values do not change that frequently and thus justify our design choice.

4 Experimental Results

In this section we first briefly describe our experimental setup and then present the results of our experimental evaluation. To evaluate the effectiveness of our cache design, we compare its performance with a variety of other cache implementations.

4.1 Experimental Setup

We implemented compression enabled partial cache line prefetching scheme using Simplescalar 3.0 [4]. The baseline processor is a four issue superscalar with two levels of on-chip cache (Figure 9). Except the basic cache configuration, we use the same parameters for implementations of all different cache designs.

Parameter	Value
Issue width	4 issue, OO
IFQ size	16 instr.
Branch Predictor	Bimod
LD/ST Queue	8 entry
Func. units	4 ALUs, 1 Mult/Div, 2 Mem ports 4 FALU, 1 FMult/FDiv
I-cache hit latency	1 cycle
Icache miss latency	10 cycles
L1 D-cache hit latency	1 cycle
L1 D-cache miss latency	10 cycles
Memory access latency	100 cycles (L2 cache miss latency)

Figure 9. Baseline experimental setup.

We chose a spectrum of programs from Olden [13], SPEC2000, and SPEC95 [14] benchmark suites. Olden benchmarks were executed with representative input sets provided with the benchmark. SPEC programs were run with the reference input set.

We compare the performances of cache configurations described below. The comparisons are made in terms of overall execution time, memory traffic, and miss rates.

- **Baseline cache (BC).** The L1 cache is 8K direct mapped and 64 bytes/line. The L2 cache is 64K 2-way associative and 128 bytes/line.
- **Baseline cache with compression (BCC).** The L1 and L2 caches are the same as BC. We add compressors and decompressors at the interfaces of the CPU and the L1 cache, the L2 cache and the memory. BC and BCC have the same performance since BCC only changes the format in which the data is stored and transmitted.
- **Higher associative cache (HAC).** The L1 cache is 8K 2-way associative and 64 bytes/line. The L2 cache is 64K 4-way associative and 128 bytes/line. Since two cache lines may be accessed if the required word is in the affiliated cache, we model a cache with double the associativity at both cache levels for comparison.
- **Baseline cache with prefetching (BCP).** The L1 and L2 caches are the same as BC; *however, we invest the hardware cost in BCC/CPP to cache prefetch buffers*. A 8-entry prefetch buffer is used to help the L1 cache and a 32-entry prefetch buffer is used to help the L2 cache. Both are fully associative with LRU replacement policy.
- **Compression enabled partial line prefetching (CPP).** The L1 cache is 8K direct mapped, 64 bytes per cache line. The L2 cache is 64K 2-way associative, 128 bytes per cache line. Partial cache lines are prefetched as we discussed.

4.2 Memory Traffic

The memory traffic comparison (Figure 10) for different configurations are normalized with respect to BC which is always 100% in the figure. From the graph, we find (1) the simple *data compression* technique greatly reduces the memory traffic – the reduction as shown by BCC is on average 60% of BC configuration; (2) *hardware prefetching* increases memory traffic significantly with an average about 80% increase.

On the other hand, the CPP design is not a simple combination of prefetching and data compression at the memory bus interface. It stores the prefetch data inside the cache which effectively provides a larger prefetch buffer than the scheme that puts all hardware overhead into supporting a prefetch buffer (BCP). As a result, the memory traffic for CPP, which on average is 90% of the traffic of the BC configuration, is lower than the average of BCC and BCP's traffic ((60%+180%)/2=120%). Thus, our CPP design reduces traffic even though it carries out prefetching.

As discussed, the only access in CPP that could increase memory traffic happens if a store instruction writes to the primary place or the affiliated place and changes a compressible value to an incompressible one. Either it will generate a cache miss (if value is written to the affiliated place) or it will cause the eviction of a dirty affiliated line (if value is written to the primary place). However, this situation does not occur often enough in practice.

4.3 Execution Time

The overall execution time comparisons of different cache configurations, normalized with respect to BC, are shown in Figure 11. The difference between the BC bar and other bars gives the percentage speedup.

First we observe that *data compression* itself affects neither the memory access sequence nor the availability of cache lines in the data cache. As a result, it has the same performance results as baseline. It is also expected that HAC consistently does better than BC. Hardware prefetching is very effective and gives better performance than HAC for 11 out of 14 programs.

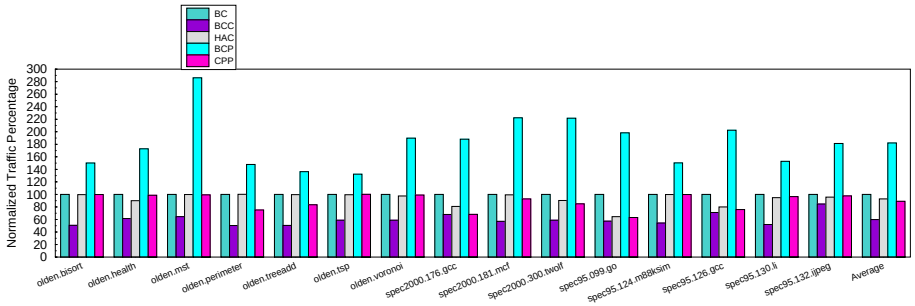

Figure 10. Comparison of memory traffic.

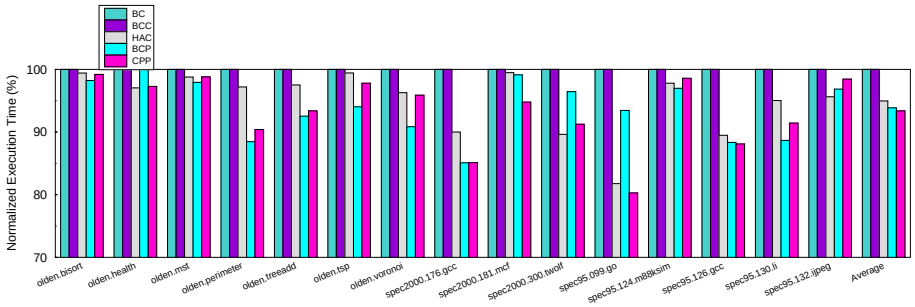

Figure 11. Performance comparison.

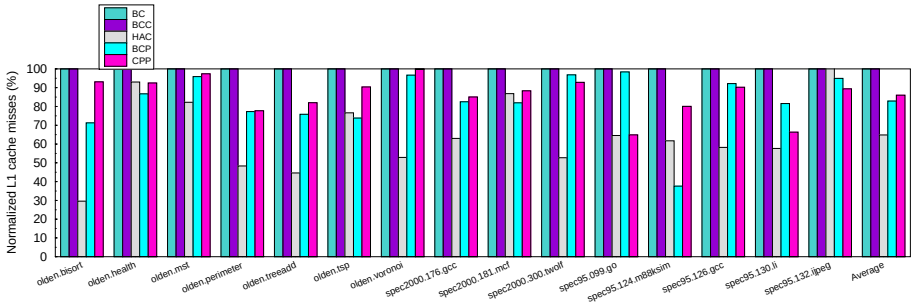

Figure 12. Comparison of L1 cache misses.

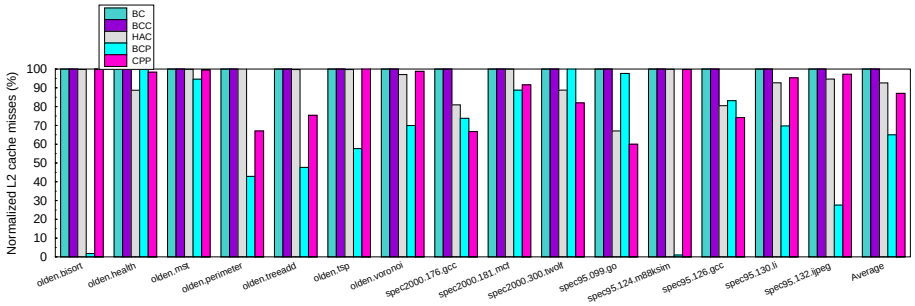

Figure 13. Comparison of L2 cache misses.

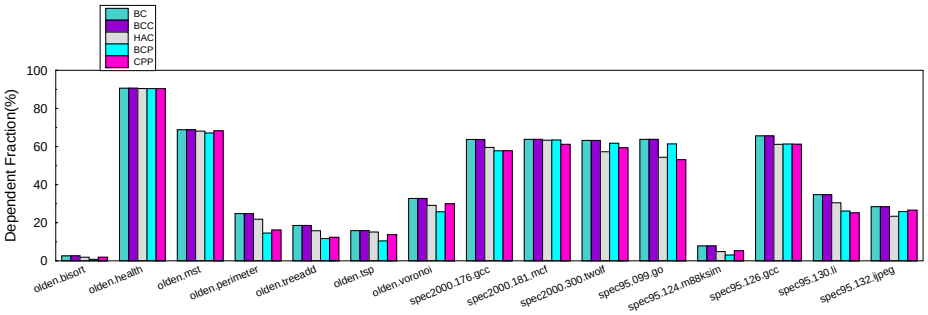

Figure 14. Importance of cache misses (estimated using % of directly dependent instructions).

The proposed CPP design does consistently better than the baseline cache. This is expected since CPP never kicks out a cache line in order to accommodate a prefetched line and thus prefetching in CPP can never cause cache pollution. On average programs run 7% faster on the CPP configuration when compared to the baseline configuration. While the HAC has a better replacement policy, CPP can hold more words in the cache. For example, although a two-way associative cache can hold two cache lines in a set, CPP can hold the content from 4 lines in these two physical lines. Thus, CPP reduces the capacity misses in comparison to HAC and improves the performance. From the figure, we can also see that CPP does better than BCP for 5 out 14 programs. Generally, CPP does slightly worse than BCP since CPP only prefetches partial cache lines and thus is less aggressive in its prefetching policy in comparison to BCP. However, if conflict misses are dominant, i.e. a higher associative cache has better performance than BCP (e.g., olden.health and spec2000.300.twolf), CPP performs better than BCP. CPP reduces the conflict misses and thus improves the effectiveness of prefetching.

4.4 Cache Miss Comparison

The comparisons of L1 and L2 cache misses are shown in Figure 12 and 13 respectively. To be clear, it is not considered as a cache miss in BCP if an access can find its data item from prefetch buffer.

Compared to BC, *prefetching techniques (BCP and CPP)* greatly reduce cache misses. Compared to HAC, *prefetching techniques* generally have comparable or more L1 cache misses but in many cases fewer L2 cache misses. HAC greatly reduces the conflict misses. For BCP, since the prefetch buffer of L1 cache is small, new prefetched items sometimes replace old ones before they are used. For CPP, a new fetched cache line kicks out a primary line and its associated prefetched line. Thus, conflict misses are not effectively removed. For L2 cache, BCP sometimes performs better than CPP since it has a larger prefetch buffer and can hide the miss penalty more effectively.

An interesting phenomenon is that although CPP sometimes has more L1 or L2 cache misses than HAC, it still achieves better overall performance (e.g., for 130.li from SPECint95 although CPP has more L1 and L2 cache misses than HAC, the overall performance using CPP is 6% better than HAC). As was dicussed, this suggests that different cache misses have different performance impacts.

Additional experiments were designed to further analyze this phenomenon. We first derive a new parameter for this purpose. Given a set of memory access instructions m, the importance of this set is defined as the percentage of total instructions that directly depend on m. In case that m is the set of cache miss instructions from a program execution, its importance parameter indicates how many directly dependent instructions are blocked by the cache misses. A higher number means that the cache misses block more instructions and thus can hurt the performance more. The method to approximately compute this percentage is as follows. According to Amdahl's law, we have overall speedup

$$\begin{aligned} S_{overall} &= \frac{Execution_{old}}{Execution_{new}} \\ &= \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{S_{enhanced}}} \\ \therefore Fraction_{enhanced} &= \frac{S_{enhanced}(1 - \frac{1}{S_{overall}})}{S_{enhanced} - 1}. \end{aligned}$$

In the Simplescalar simulator, by varying only the cache miss penalty and running the program twice without speculative execution, we observe the same cache misses happen at the same instructions. Since their directly dependent instructions are also fixed, the main change to the execution is the reduced dependence length from a cache miss instruction to its directly dependent instructions, the enhanced fraction could thus be considered as the percentage of the instruction that are directly depending on these cache misses.

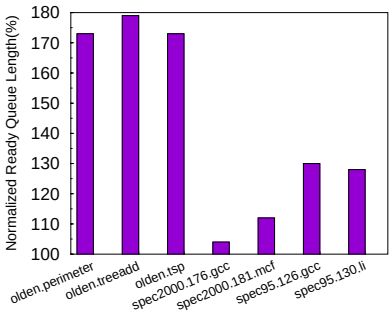

Figure 15. Average ready queue length in miss cycles.

Now, for different cache configurations, this fraction is computed as follows. First, the cache miss latency is reduced in half, which means $S_{enhanced}$ =2. Second, the overall performance speedup is measured, which is $S_{overall}$. It is computed from the total number of cycles before and after changing the miss penalty. Now, the value of $Fraction_{enhanced}$ can be obtained. The results for different configurations are plotted in Figure 14 from which we can find that CPP reduces the importance of the cache misses for most benchmarks. For the benchmarks that are slower than HAC, it is seen that they have larger importance parameters. This estimation is consistent with the result shown in Figure 11.

For the benchmarks with significant importance reduction, we further study the average ready queue length in the processor, when there is at least one outstanding cache miss. The queue length increase of CPP over the HAC was studied (Figure 15). The results indicate that the average queue

length is improved by up to 78% for these benchmarks. This parameter tells us when there is a cache miss in the new cache design, the pipeline still has a lot of work to do.

To summarize, we conclude that CPP design reduces the importance of caches misses when compared to BC and HAC configurations. That is the reason why CPP sometimes has higher cache misses but still gives better overall performance.

5 Related work

Different prefetching techniques have been proposed to hide cache miss penalty and improve cache performance. Hardware prefetching [2, 7, 1] does not require compiler support and the modification of existing executable code. Simple schemes [3] prefetch the data of next cache line while more sophisticated schemes use dynamic information to find data items with fixed stride [2] or arbitrary distance [1]. However, prefetching techniques significantly increase the memory traffic and memory bandwidth requirements. Our new proposed scheme, on the other hand, employs data compression and effectively transmits more words with same memory bandwidth. It does not explicitly increase the memory traffic and improve the overall performance.

Currently, data compression has been adapted into cache design mainly for reducing power consumption. Existing designs [5, 6] improve data density inside the cache with compression schemes of different dynamic cost and performance gain. In [5] a relatively complex compression algorithm is implemented in hardware to compress two consecutive lines. Due to its complexity, it is employed at L2 cache and data items are decompressed to L1 cache before its access. In [6] data could be compressed at both levels by exploiting frequent values found from programs. Two conflicting cache lines can be stored in the same line if both are compressible; otherwise, only one of them is stored. Both of the above schemes operate at the cache line level and do not distinguish the importance of different words within a cache line. As a result, they could not exploit the saved memory bandwidth for partial cache line prefetching.

The pseudo associative cache [12] also has a primary and a secondary cache line. Our new design has similar access sequence. However, the cache line is updated very differently. For pseudo associative cache, if a cache line enters its secondary place, it has to kick out the original line. Thus it has the danger to degrade the cache performance by converting a fast hit to a slow hit or even a cache miss. On the contrary, the new cache design only stores a cache line to its secondary place if there are free spots. It will neither pollute the cache line nor degrade the original cache performance.

6 Conclusion

A novel cache design is developed in this paper to remove the memory traffic obstacle of hardware prefetching. It partially prefetches compressible words from the next cache line from lower level memory hierarchies. It does not explicitly increase memory traffic and removes prefetch buffers. On an average, the new design improves the overall performance 7% over the base and 2% over the higher associativity cache configurations.

References

[1] A. Roth and A. Moshovos and G.S. Sohi, "Dependence based prefetching for linked data structures," *ACM 8th ASPLOS*, pages 116-126, 1998.

[2] J.L. Baer and T. Chen, "An effective on-chip preloading scheme to reduce data access penalty," *Supercomputing '91*, pages 178-186, 1991.

[3] N.P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fullyassociative Cache and Prefetch Buffers," *17th ISCA*, pages 364-373, 1990.

[4] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *CS-TR-97-1342*, Univ. of Wisconsin-Madison, Jun 1997.

[5] J-S. Lee, W-K. Hong and S-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," *IEEE ICCD*, pages 184-191, 1999.

[6] J. Yang, Y. Zhang and R. Gupta, "Frequent Value Compression in Data Caches," *IEEE/ACM MICRO*, pages 258-265, 2000.

[7] A. Smith, "Cache memories," *ACM Computing Survey*, 14:473-530, Sep 1982.

[8] Y. Zhang and R. Gupta, "Data Compression Transformations for Dynamically Allocated Data Structures," *International Conference on Compiler Construction*, pages 14-28, 2002.

[9] Y. Zhang, J. Yang and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *ACM 9th ASPLOS*, pages 150-159, 2000.

[10] T.M. Chilimbi, M.D. Hill, and J.R. Larus, "Cache-Conscious Structure Layout," *ACM PLDI*, pages 1–12, 1999.

[11] T.M. Chilimbi, M.D. Hill, and J.R. Larus, "Cache-Conscious Structure Definition," *ACM PLDI*, pages 13-24, 1999.

[12] D. Patterson and J. Hennessy, "Computer Architecture:A Quantitative Approach," 2nd Edition, Morgan Kaufmann Publishers, Inc. 1995.

[13] M. Carlisle, "Olden: Parallelizing Progrms with Dynamic Data Structures on Distributed-Memory Machines," PhD Thesis, Princeton Univ., Dept. of Comp. Science, June 1996.

[14] http://www.spec.org/.

[15] J. Yang and R. Gupta, "Energy Efficient Frequent Value Data Cache Design," *IEEE/ACM MICRO*, pages 197-207, 2002.

[16] Y. Zhang, "The Design and Implementation of Compression Techniques for Profile Guided Compilation," PhD Thesis, Univ. of Arizona, Dept. of Computer Science, Tucson, AZ, August 2002.

[17] M.D. Hill, "Multiprocessors should support simple memory consistency models," IEEE Computer, Vol 31:8, pages 28-34, 1998.