Michigan Technological University
Digital Commons @ Michigan Tech

Michigan Tech Publications

6-2021

Decreasing the Miss Rate and Eliminating the Performance Penalty of a Data Filter Cache

Michael Stokes
Florida State University

David Whalley
Florida State University

Soner Onder
Michigan Technological University, soner@mtu.edu

Follow this and additional works at: https://digitalcommons.mtu.edu/michigantech-p

Part of the Computer Sciences Commons

Recommended Citation
Stokes, M., Whalley, D., & Onder, S. (2021). Decreasing the Miss Rate and Eliminating the Performance Penalty of a Data Filter Cache. *ACM Transactions on Architecture and Code Optimization, 18*(3). http://doi.org/10.1145/3449043
Retrieved from: https://digitalcommons.mtu.edu/michigantech-p/15122

Follow this and additional works at: https://digitalcommons.mtu.edu/michigantech-p
Part of the Computer Sciences Commons

Decreasing the Miss Rate and Eliminating the Performance Penalty of a Data Filter Cache

MICHAEL STOKES and DAVID WHALLEY, Florida State University
SONER ONDER, Michigan Technological University

While data filter caches (DFCs) have been shown to be effective at reducing data access energy, they have not been adopted in processors due to the associated performance penalty caused by high DFC miss rates. In this article, we present a design that both decreases the DFC miss rate and completely eliminates the DFC performance penalty even for a level-one data cache (L1 DC) with a single cycle access time. First, we show that a DFC that lazily fills each word in a DFC line from an L1 DC only when the word is referenced is more energy-efficient than eagerly filling the entire DFC line. For a 512B DFC, we are able to eliminate loads of words into the DFC that are never referenced before being evicted, which occurred for about 75% of the words in 32B lines. Second, we demonstrate that a lazily word filled DFC line can effectively share and pack data words from multiple L1 DC lines to lower the DFC miss rate. For a 512B DFC, we completely avoid accessing the L1 DC for loads about 23% of the time and avoid a fully associative L1 DC access for loads 50% of the time, where the DFC only requires about 2.5% of the size of the L1 DC. Finally, we present a method that completely eliminates the DFC performance penalty by speculatively performing DFC tag checks early and only accessing DFC data when a hit is guaranteed. For a 512B DFC, we improve data access energy usage for the DTLB and L1 DC by 33% with no performance degradation.

CCS Concepts: • **Computer systems organization → Embedded hardware**;

Additional Key Words and Phrases: Data filter cache, line fills, data cache compression, guaranteeing cache hits

ACM Reference format:
Michael Stokes, David Whalley, and Soner Onder. 2021. Decreasing the Miss Rate and Eliminating the Performance Penalty of a Data Filter Cache. *ACM Trans. Archit. Code Optim.* 18, 3, Article 28 (April 2021), 22 pages.
https://doi.org/10.1145/3449043

28

This work was supported in part by the US National Science Foundation (NSF) under grants DUE-1259462, IIA-1358147, CCF-1533828, CCF-1533846, DGE-1565215, DRL-1640039, CRI-1822737, CCF-1823398, CCF-1823417, CCF-1900788, and CCF-1901005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of the NSF.
Authors' addresses: M. Stokes and D. Whalley, 1004 Academic Way, Computer Science Department, Florida State University, Tallahassee, Florida 32306-4530; emails: {mstokes, whalley}@cs.fsu.edu; S. Onder, 1400 Townsend Drive, Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931; email: soner@mtu.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/author(s).
1544-3566/2021/04-ART28
https://doi.org/10.1145/3449043

1 INTRODUCTION

It has been estimated that 28% of embedded processor energy is due to data supply [3]. Thus, reducing data access energy on such processors is a reasonable goal. A **data filter cache (DFC)**, sometimes also known as a **level-zero data cache (L0 DC)**, has been shown to be effective at reducing energy usage, since it requires much less energy to access than the much larger **level-one data cache (L1 DC)** and can still service a reasonable fraction of the memory references [10, 11]. However, a conventional DFC has disadvantages that have prohibited its use in contemporary embedded or high performance processors. First, a DFC has a relatively high miss rate due to its small size. A conventional DFC is accessed before the L1 DC causing the L1 DC to be accessed later than it would traditionally be accessed within the instruction pipeline, resulting in degradation of performance on DFC misses. Second, a single cycle **filter cache (FC)** line fill as proposed in many prior studies [4–6, 10, 11, 20] has been shown to be unrealistic, as it can adversely affect L1 DC area and significantly increase the energy usage for each L1 DC access [1]. A multicycle DFC line fill can also be problematic, as it can interfere with subsequent accesses to the DFC or L1 DC. These issues must be resolved for a DFC to be a practical alternative in a processor design.

We propose a new design that utilizes a DFC without the aforementioned problems. Our design for effectively using a DFC makes the following contributions: (1) We show that it is more energy-efficient on a DFC miss to lazily fill only a single word into a DFC line when the word is referenced and not resident than to eagerly fill every word of an entire DFC line. (2) As far as we are aware, we provide the first data compression technique for a DFC or for any first-level cache that shares and packs data words in a single cache line at the granularity of individual words from different lines or sublines in the next level of the memory hierarchy without increasing the cache access time. (3) We present a method that completely eliminates the DFC miss performance penalty by speculatively performing the DFC tag check early and only accessing DFC data when a hit is guaranteed.

2 BACKGROUND

Figure 1 shows the address fields for accessing the DTLB, L1 DC, and DFC, which are frequently referenced in the remainder of the article. The nonbold portion shows the virtual and physical addresses used to access the DTLB and L1 DC, respectively. The *virtual page number* is used to access the DTLB to produce the corresponding *physical page number*. The *page offset* remains the same. The *L1 block number* uniquely identifies the L1 DC line. The *L1 offset* indicates the first byte of the data in the L1 DC line. The *L1 index* is used to access the L1 DC set. The *L1 tag* contains the remaining bits that are used to verify if the line resides in the L1 DC. The bold portion shows the fields for accessing the DFC. The DFC fields are similar to the L1 fields. But due to the DFC containing smaller lines and less total data than an L1 DC, the DFC offset and DFC index fields are smaller and the DFC tag field is larger.

Each data word filled in a DFC line will require a read from the L1 DC and a write to the DFC. Placing a data word in a DFC line will only be beneficial for reducing energy usage when the data word is subsequently referenced due to temporal locality or when multiple individual portions (e.g., bytes) of the word are referenced due to spatial locality within the word. Energy usage will be reduced when n L1 DC accesses are replaced by a single L1 DC access and n DFC accesses requiring less energy, where n is greater than one.

3 TECHNIQUES TO IMPROVE DFC EFFICIENCY

We describe three general techniques in this article to improve the efficiency of a DFC. First, we describe how energy usage can be reduced by filling data words into a DFC line only when they are first referenced. Second, we illustrate how the DFC miss rate can be decreased by sharing a

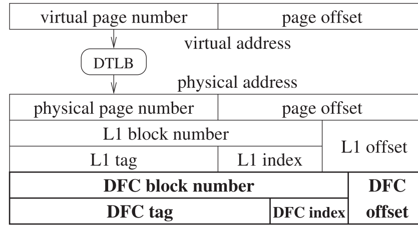

Fig. 1. Address fields.

Fig. 2. Components of a DFC lazy word filled line.

DFC line between multiple L1 DC (sub)lines. Finally, we outline how the DFC miss penalty can be eliminated by performing a DFC tag check early and only accessing the DFC data when it is known to reside in the DFC. Thus, we can reduce energy usage by eliminating unnecessary loads of words into a DFC and improving the DFC miss rate and at the same time not have any adverse effect on performance.

3.1 Lazily Filling Data Words into a DFC Line

In most caches, an *eager line fill strategy* is used where an entire cache line is filled with data from the next level of the memory hierarchy when there is a cache miss. Eagerly filling cache lines on a cache miss can improve performance as it increases the cache's hit rate, thus avoiding accessing the next level of the memory hierarchy to retrieve the data. However, a single cycle is required to load a word from an L1 DC in many embedded processors, meaning that introducing a DFC that is always accessed before the L1 DC can only degrade performance due to DFC misses. We will show in Section 3.3 that the performance degradation associated with DFC misses can be completely eliminated by only loading a value from the DFC when a hit is guaranteed. In this context, eagerly filling words into a DFC line on a DFC miss should only be performed if it can improve energy efficiency.

Many prior FC studies have proposed to fill an FC line in a single cycle to minimize the FC miss penalty [4–6, 10, 11, 20]. Fetching an entire DFC line of data in a single cycle has been asserted to be unrealistic, as a large bitwidth to transfer data between the CPU and the L1 DC can adversely affect L1 DC area and significantly increase the energy usage for each L1 DC access [1]. Given that a sizeable fraction of the data memory references will access the L1 DC due to a typically high DFC miss rate, it is best to utilize an L1 DC configuration that is efficient for L1 DC accesses.

We use a *DFC lazy word fill strategy* where each word is only filled when the word is referenced and is not resident in the DFC. We assume a uniform L1 DC bus width of 4 bytes. So when a load byte or load halfword instruction is performed and the word is not resident, the entire word from the L1 DC is copied into the DFC and the appropriate portion of the word is extended and sent to the CPU. Alignment requirements are enforced by the compiler we used so a load or a store never spans more than one word or a DFC line boundary. Figure 2 shows the information in the DFC line that is used for our *DFC lazy word fill strategy*. An *f* (filled) bit is associated with each word in the DFC line indicating if the word is resident, which can be viewed as the valid bit vector of a sectored cache [18]. A DFC *word hit* requires both a DFC tag match and the *f* bit to be set. This cache organization can be viewed as an extreme instance of subblocking, where each subblock is

the size of one word. We assume a DFC write through and write allocate policy for a *DFC line miss* (DFC line is not resident) or for a *DFC line hit+word miss* (DFC line is resident, but the referenced word within the line is not resident) as we are only writing a single word to the line at a time. However, we do not allocate a word on a *line hit+word miss* when there is a byte or halfword store, as we would have to read the word from the L1 DC first to ensure that the entire word is resident in the DFC.

One advantage of eagerly filling an entire DFC line is that only a single L1 DC tag check is required for the entire DFC line fill. With our lazy fill approach, *DFC line hits+word misses* are common where the DFC line corresponding to the L1 DC (sub)line that holds the data is resident, but the desired word is not. To still provide a benefit for these cases, we store the L1 DC way corresponding to the L1 DC (sub)line that holds the data along with the DFC line, as shown in Figure 2. When there is a *DFC line hit+word miss*, the *L1 DC way* field is used to access the L1 DC without an L1 DC tag check. Other approaches for memoizing the L1 DC way have been used in the past [12, 19, 21]. In addition, only a single L1 DC data array is accessed to load the word from the L1 DC into the DFC line. Only a DFC *line miss* will require an L1 DC tag check and a set-associative L1 DC data access. Note the DFC in our design is inclusive, where the data in each DFC line is guaranteed to be resident in the L1 DC. Thus, both DFC eagerly and lazily filled line approaches can avoid redundant L1 DC tag checks and set associative L1 DC data accesses.

L1 DC line evictions can occur due to conventional L1 DC misses or multiprocessor invalidation requests. We perform a DFC tag check to invalidate DFC lines whenever an L1 DC line is evicted. Thus, no additional changes are needed to the DFC in each core to provide multiprocessor support.

Having a write-through policy means that for each store a value is written to both the DFC and L1 DC, which can result in more energy usage for stores. However, utilizing the L1 DC way in the DFC to avoid an L1 DC tag check on DFC hits can offset some of the extra energy for writing the data value in both the DFC and L1 DC.

DFC lines are also more frequently evicted than L1 DC lines due to more capacity misses, which can result in words never being referenced that were eagerly filled in a DFC line. In contrast, each word in a lazily filled DFC line is referenced at least once.

Another DFC eager line fill strategy is to fill the entire DFC line one word at a time over multiple cycles, which will be more energy-efficient than filling a DFC line in one cycle. A multicycle DFC line fill will be problematic if it will delay subsequent accesses to the DFC or L1 DC. Also, dealing with multiple outstanding DFC line fills will require more complex logic. After a DFC line is allocated, each word that is filled in that line using either an eager or lazy approach will require an L1 DC read and a DFC write. However, a DFC lazy word fill approach is much simpler and will not fill words that are never referenced before the line is evicted. Furthermore, we will later show that many of the words that will be filled over multiple cycles will not be referenced and thus needlessly loaded into the DFC, wasting energy.

3.2 Decreasing the DFC Miss Rate by Line Sharing and Data Packing

We proposed in Section 3.1 a DFC lazy word filled approach that can be more energy-efficient than an aggressive DFC eager line fill approach when a DFC causes no performance penalty. We now present optimizations to the lazy word filled DFC that improve its hit rate. Much of the space available in a lazy word filled DFC line goes unused, as some words in the line will not be filled, because they are not referenced before the DFC line is evicted. To make better use of this available space in a lazy word filled DFC line, we allow multiple L1 DC (sub)lines to share the same DFC line. As long as the L1 DC (sub)lines refer to words in different parts of the DFC line, the DFC can simultaneously hold values from different L1 DC (sub)lines. This approach can decrease the DFC miss rate, as there will be fewer DFC line evictions and can increase the amount of data the DFC

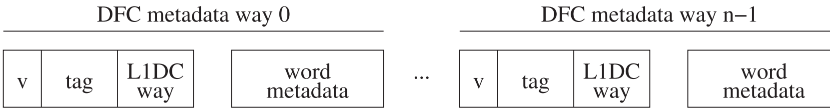

Fig. 3. DFC line metadata.

Table 1. DFC Metadata for Sharing Data Words

Code	Interpretation
0	4-byte value
1	not resident

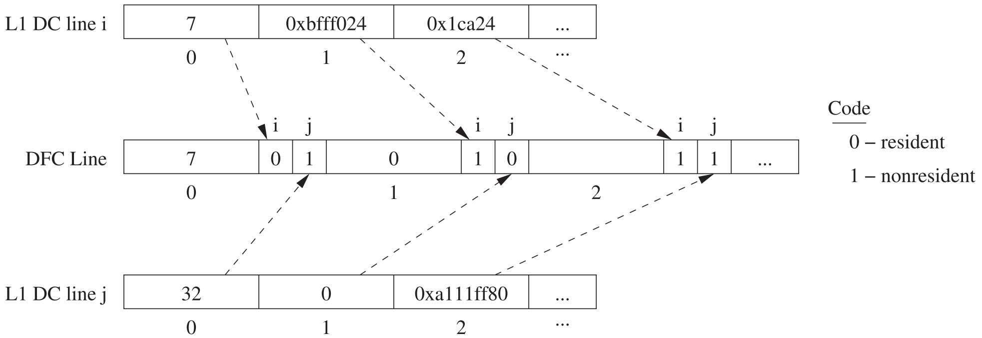

Fig. 4. Example of line sharing.

will likely hold. However, if multiple L1 DC (sub)lines refer to words corresponding to the same position in the DFC line, then only the most recently referenced word is retained.

In this section, we assume a direct-mapped DFC where a single DFC line is associated with each DFC set. We allow one, two, or four L1 DC (sub)lines to share each DFC line, depending on the hardwired DFC configuration. There must be a tag and other metadata for each L1 DC (sub)line that shares a single DFC line. Thus, there are multiple DFC tag arrays, but only a single DFC data array. There will be DFC metadata associated with each L1 DC (sub)line that currently shares a DFC line denoting which words of the L1 DC (sub)line are resident. Figure 3 shows that this DFC metadata for each L1 DC (sub)line will include a valid bit, a tag, an L1 DC way, and metadata about each data word. LRU information for the L1 DC (sub)lines that share a DFC line is used to determine which L1 DC (sub)line to evict on a DFC *line miss*, but is not shown in the figure.

Table 1 shows one option where different words within distinct L1 DC (sub)lines can reside in the same DFC line at the same time. We refer to this option as *DFC* **line sharing (LS)**. Only a single bit of metadata is required for each word within each L1 DC (sub)line to indicate if the word is resident or not, replacing the *f* (filled) bit in Figure 2. At most a single value from two (four) corresponding words in the L1 DC (sub)lines can reside in the DFC line at one time as we allow two (four) L1 DC (sub)lines to share a single DFC line. This requires DFC word evictions if two or more words are referenced from multiple L1 DC (sub)lines that correspond to the same word in the DFC line.

Figure 4 shows an example of two L1 DC lines sharing a DFC line (2xLS). These two L1 DC lines *i* and *j* are depicted with the first three values shown in each line. The DFC line shows both the

Table 2. DFC Metadata for Sharing+Packing Data Words

Code	Interpretation	Can Pack with
00	zero value	00, 01, 10, 11
01	1-byte (4xLS) or 2-byte (2xLS) value	00, 01, 11
10	4-byte value	00, 11
11	not resident	00, 01, 10, 11

data words and the metadata codes to the right of each word, using the metadata codes shown in Table 1. Word 0 in line *i* is resident and word 1 in line *j* is resident. Word 2 is not resident for either L1 DC line. This type of DFC line sharing may be beneficial when different portions of different L1 DC (sub)lines that map to the same DFC line are being accessed close in time. We will later show how some of the other values in L1 DC lines *i* and *j* may also be placed in the same DFC line.

Table 2 shows an extension to line sharing where values associated with the same corresponding words from distinct L1 DC (sub)lines can sometimes reside in the same word within a DFC line at the same time. We refer to this option as *DFC* **line sharing and data packing (LS+DP)**. Rather than evicting words from a DFC line if multiple L1 DC (sub)lines attempt to share the same DFC word, we allow multiple L1 DC (sub)lines to share the same word if the multiple values taken together can fit in four bytes. This approach will decrease the miss rate, as there will be fewer DFC word evictions and the amount of data stored inside the DFC will increase.

We require two bits of metadata for each word within each L1 DC (sub)line sharing the same DFC line to support data packing, as shown in Table 2. When a word is loaded from the L1 DC, the processor checks if the value is *narrow width*, meaning that the value can be represented in fewer bytes than a full data word. If only two distinct L1 DC (sub)lines are allowed to share the same DFC line (2xLS+DP), then *narrow width* means that the value can be represented in two bytes. If four distinct L1 DC (sub)lines are allowed to share the same DFC line (4xLS+DP), then *narrow width* means that the value can be represented in one byte. Otherwise, the value is considered *full width* (code 10). Zero can be viewed as a special narrow-width value, where the data value is not actually stored in the DFC line. Thus, a zero value (code 00) can be packed in the same word with any other value. A nonzero narrow width value (code 01) can be packed into the same word with any value that is not full width.

The placement within the word of nonzero narrow-width values for a given L1 DC (sub)line will be based on the *DFC metadata way* of the L1 DC (sub)line, as shown in Figure 3. When only two L1 DC (sub)lines can share the DFC line, a nonzero narrow width value will be placed in the lower halfword if the *DFC metadata way* was zero and the upper halfword if the *DFC metadata way* was one. Likewise, a nonzero narrow width value would be placed in the corresponding byte based on the *DFC metadata way* of the L1 DC (sub)line that is sharing that DFC line when four L1 DC (sub)lines can share the DFC line.

Figure 5 shows an example of line sharing and data packing with two L1 DC lines sharing a DFC line (2xLS+DP). These two L1 DC lines have the same values as in Figure 4. The DFC metadata for these two lines use the metadata codes shown in Table 2. Word 0 in each of the two L1 DC lines can be packed into word 0 of the DFC line, as both values are narrow width (can be represented in 2 bytes). Word 1 in each of the two L1 DC lines can be packed into word 1 of the DFC line, as word 1 of L1 DC line *j* is zero and is not stored in the DFC line. At most one value for word 2 can be stored as the values in word 2 for both L1 DC lines *i* and *j* are full width. It may be the case that the value from word 2 in L1 DC line *i* has not yet been referenced from the time the L1 DC line *i* was allocated in the DFC and thus has not yet been filled.

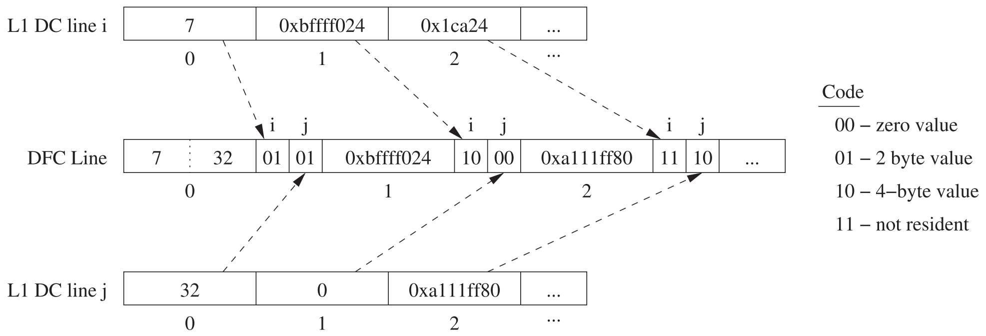

Fig. 5. Example of line sharing and data packing.

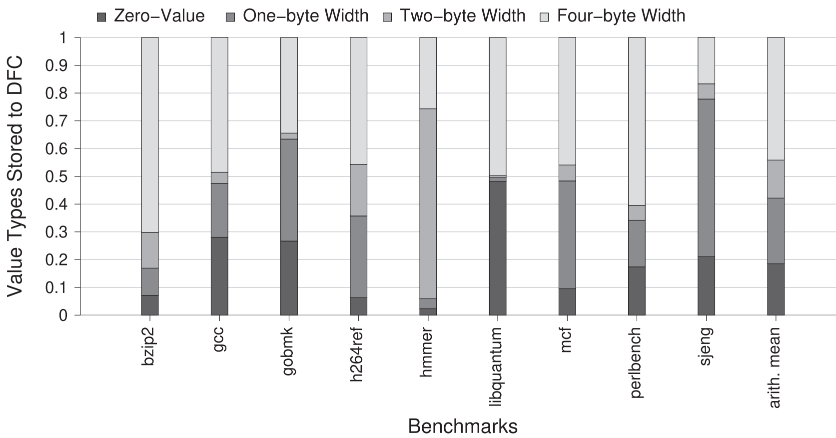

Fig. 6. DFC data value taxonomy.

Figure 6 shows the taxonomy of the data values stored in a DFC when applied to the SPECint 2006 benchmark suite. On average, 18.5% were zero values, which do not require an access to the DFC data. 23.6% could be represented in a single byte and 13.7% required two bytes. 44.1% of the values required three or four bytes to be represented. Thus, 55.9% of the values can be potentially packed into a word with a value from a different L1 DC line.

The DFC word metadata is organized into arrays, where there is a separate array for each set of corresponding words of DFC metadata from the L1 DC (sub)lines. The specific array to be accessed is determined by the *DFC word offset* (see Figure 1) value. The specific DFC word metadata within the set of corresponding DFC words is selected by which DFC tag matches and this metadata will indicate if the word is resident.

LS requires that multiple tags be checked in parallel, just like a set-associative cache. However, we are only accessing a single data word where a set-associative cache would access multiple data words. The LS check to determine if the word is resident may take slightly longer than a direct-mapped cache without LS, as we have to not only perform a tag check for the tags associated with that DFC line, but also check if the word is resident for the tag that matched. The access time should still be less than a set-associative DFC, as there is no need to access multiple data words from the lines in the set through a multiplexor, as only a single data word is accessed. In addition, we are assuming the 4-way set associative L1 DC can be accessed in a single cycle. The DFC associative tag memory is much smaller than the L1 DC tag memory and will require less time to access.

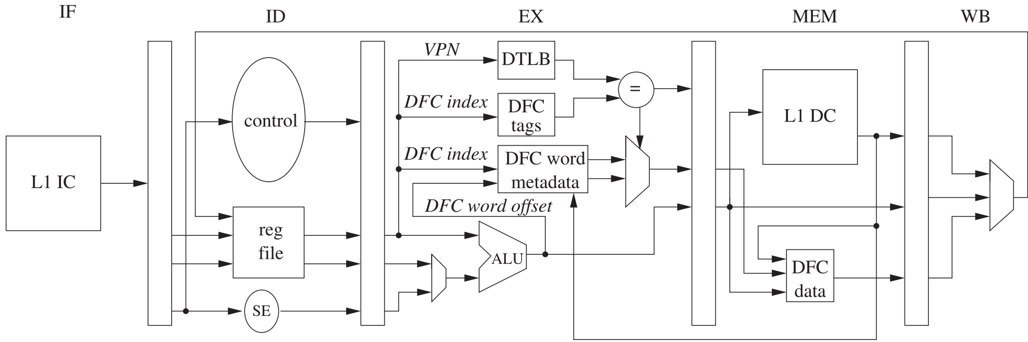

Fig. 7. Modified datapath to support guaranteed DFC hits.

3.3 Eliminating the DFC Miss Penalty by Only Accessing DFC Data on DFC Hits

It is desirable to not degrade performance when a value is not resident in the DFC, as DFC miss rates can be fairly high. We revised a traditional instruction pipeline that only loads data from a DFC when a DFC *word hit* is guaranteed. We guarantee DFC *word hits* by speculatively accessing the DFC metadata using the upper bits of the base register when the offset is small. If the addition of the base register and the offset does not cause these upper bits to change, then the speculation is successful and we can access the DFC data if the metadata shows the word is resident or we can use the L1 DC way if the metadata shows that the line is resident but the word is not. Otherwise, if we (1) are not able to speculate because the offset was too large, (2) speculatively access the metadata, but the calculation of the effective address modifies the bits we used to access the metadata, or (3) have a line miss in the metadata, then we must access the L1 DC, but now with no performance penalty. Figure 7 shows a classical five-stage pipeline modified to support access to a DFC with no performance penalty. Only the datapath for ALU and load instructions is shown with no forwarding to simplify the figure.

Assume a load instruction is being processed in the pipeline. The *virtual page number* **(VPN)** field (see Figure 1) of the base register value is used to speculatively access the DTLB during the EX stage. The DTLB is small enough to be accessed so the physical page number can be compared to a physical tag stored in the DFC in a single cycle. Unless the VPN is modified when calculating the effective address by adding the displacement, the physical page number that is output from the DTLB will be valid. The *DFC index* field (see Figure 1) of the base register value is used in the EX stage to speculatively access the DFC tag arrays so a DFC tag check can be performed. The processor will check that the *DFC index* field of the address is unaffected by the effective address addition by ensuring that the displacement is less than the DFC block size and inspecting the carry out of the *DFC offset* field. The *DFC index* field is also used to access the DFC word metadata. The processor will access the L1 DC during the MEM stage when the *DFC block number* field (see Figure 1) is affected by the effective address calcuation, a DFC *line miss* occurs, or the specified word is not resident in the DFC line. If there is a DFC *line hit+word miss*, then only a single L1 DC data array is accessed with no L1 DC tag check, as the *L1 DC way* field in the DFC line indicates which L1 DC way to access. If the DFC word metadata indicates that the value is zero, then the DFC data is not accessed. Otherwise, a DFC *word hit* for a nonzero value occurred, the DFC data will be accessed in the MEM stage, and the word metadata obtained during the EX stage will be used to determine how to extract the value from the word in the DFC line.

For DFC *word hits* that load a value of zero, the value is obtained after the EX stage instead of the MEM stage. Although this feature could be used to provide a small performance improvement, it was not evaluated in this study.

The DFC in our design will not be able to service every load due to DFC line misses, DFC *line hits+word misses*, and when the *DFC block number* field (see Figure 1) is affected by the effective address calculation. Thus, it would be desirable to load the data from the L1 DC in the MEM stage when the data cannot be supplied by the DFC in the EX stage. We use an inclusive cache policy and a DFC write-through policy to ensure that the L1 DC always has the most recent data. A write-through policy is much simpler to implement than a write-back policy in our DFC, as we do not have to deal with writing back dirty DFC lines over multiple cycles when a DFC line is replaced. Eviction of DFC lines due to L1 DC line evictions is also simpler, as the evicted DFC lines simply need to be invalidated.

Although not shown in Figure 7, the DFC tag comparison and DFC word metadata access is performed again in the MEM stage when the effective address calculation affects the *DFC block number*. If there is not a DFC tag match (DFC *line miss*), then the appropriate L1 DC (sub)line is allocated in the DFC line. The word loaded from the L1 DC is placed in the DFC line if there is either a DFC *line miss* or DFC *line hit+word miss*. If there is a DFC tag match and the word is resident on a store instruction, then both the DFC and L1 DC are updated in the MEM stage. In the few cases when the effective address calculation affects the virtual page number, we access the DTLB again in the MEM stage and do not allow a DTLB access in the EX stage of the next instruction during the same cycle.

The logic to speculatively perform a DFC tag check is similar to past work where a speculative L1 DC tag check was performed in the EX stage [2]. In that work the authors provided an RTL implementation synthesized using the Synopsis Design Compiler. The logic that we propose in this article is much simpler, as the tag arrays of a DFC are much smaller than the tag arrays of an L1 DC due to fewer lines in the DFC. Hence, we do not anticipate any increase in the processor cycle time.

The DFC tag memory and DFC data memory only need to have a single port for a single issue processor. We do not do any DFC operations in parallel except for accessing the DFC tag memory in the EX stage and the DFC data in the MEM stage for two different memory access instructions. If there is ever a structural hazard, then we give priority to the instruction that entered the pipeline first and access the L1 DC for the data for the instruction that entered the pipeline later. On an invalidation of an L1 DC line due to a line being replaced in a lower level of the memory hierarchy or due to a multiprocessor invalidation request, we invalidate the appropriate DFC lines while the entire processor is stalled.

3.4 Summary of the Techniques to Improve DFC Efficiency

We summarize in this subsection how the techniques described in the preceding three subsections interact. Figure 8 shows the DFC actions that are taken in the EX and MEM pipeline stages.

In the EX stage, we check if the DFC line is resident as a metaline. If the metaline is not resident, then we always allocate a DFC metaline since we are using a write allocate policy, will mark each word within that metaline as invalid, and indicate that there was a word miss in the EX/MEM pipeline register. IF the metaline is resident, then we access the existing word metadata for the accessed word and write that information to the EX/MEM pipeline register. Note that the word metadata can either come from the DFC word metadata store or can be forwarded from the MEM stage if that same word was updated by a load word miss or a store in the previous instruction. We also indicate which metaline will be accessed in the DFC data during the MEM stage.

In the MEM stage, we take different actions, depending on whether the access was a load or a store. If the access was a load and we detected a word miss in the EX stage, then we load the word from the L1 DC and pack the word with the corresponding words in the other metalines. If the access was a load and we detected a word hit in the EX stage, then we unpack the value

```
DFC_access EX stage:
    IF DFC line is not resident as a metaline THEN
        Allocate DFC tag as a metaline replacing the LRU metaline for
            that DFC line and invalidate word metadata for that metaline.
        Write indication of word miss to EX/MEM pipeline register.
    ELSE
        Access existing word metadata for accessed word within the DFC line
            and write to EX/MEM pipeline register.
        Write indication of word hit or word miss to EX/MEM pipeline register.
    Write indication of metaline to be accessed in DFC data.

Pack value into DFC word (data value, DFC data word location):
    Determine if data value can be packed into 0, (1 or 2), or 4 bytes.
    IF packed size is 1 or 2 bytes THEN
        Place value within DFC data word based on the metaline way.
    ELSE IF packed size is 4 THEN
        Update entire DFC word.
    Update DFC word metadata.

DFC_access MEM stage:
    IF load access THEN
        IF word is not resident in the DFC metaline THEN
            Load word from the L1 DC.
            Pack value into DFC word (data value, DFC data word location).
        ELSE
            Extract word from DFC data.
        Send requested data to the CPU.
    ELSE // store access
        IF word is resident in the DFC metaline OR access is word size THEN
            Pack value into DFC word (data value, DFC data word location).
        Write new data value through to the L1 DC.
```

Fig. 8. Pseudocode showing DFC actions taken in EX and MEM pipeline stages.

from the DFC data. We always send the loaded data to the CPU. If the access was a store and the word is resident or an entire word is being stored, then we pack the new word value with the corresponding data words in the other metalines for that DFC line and update the DFC word metadata. We always write the store data value through to the L1 DC to keep the data values in the DFC and the L1 DC consistent.

4 EVALUATION ENVIRONMENT

In this section, we describe the experimental environment used to evaluate the previously described DFC techniques. We use the 9 C benchmarks from the SPECint 2006 benchmark suite, which are compiled using *gcc* with the *-03* option. We use the ADL simulator [15] to simulate both a conventional MIPS processor as the baseline and a modified processor, as described in this article. Note that in this article, we only address the data memory hierarchy within a processor for a simple low-end embedded processor. This simulator performs a more realistic simulation than many commonly used simulators (in ADL data values are actually loaded from the caches,

Table 3. Benchmark Information

Name	Description	Insts Simulated	Data Pages Accessed	L1 DC Miss Rate
bzip2	text compression	10,143,879,979	2,858	0.024
gcc	C compiler	5,519,277,424	1,507	0.021
gobmk	plays game of Go	842,235,543	2,047	0.003
h264ref	video compression	107,222,101,484	3,382	0.006
hmmer	protein sequence analysis	2,165,534,715	65	0.002
libquantum	quantum comp simulation	304,226,199	29	0.009
mcf	vehicle scheduling	4,242,148,053	24,783	0.288
perlbench	Perl interpreter	1,513,785	44	0.011
sjeng	chess program	19,251,684,223	22,358	0.004

Table 4. Processor Configuration

page size	4KB, assume page coloring to support VIPT organization
L1 DC	32KB, 64B line size, 4-way associative, 1 cycle access time
DTLB	32 entries, fully associative
DFC	direct mapped, 32B line size, 128B to 1KB cache size, 1 cycle access time

Table 5. Energy for L1 DC and DTLB Components

Component	**Energy**
Read L1 DC Tags - All Ways	0.782 pJ
Read L1 DC Data 4 Bytes - All Ways	8.192 pJ
Read L1 DC Data 32 Bytes - All Ways	256.300 pJ
Write L1 DC Data 4 Bytes - One Way	3.564 pJ
Read L1 DC Data 4 Bytes - One Way	1.616 pJ
Read DTLB - Fully Associative	0.880 pJ

values are actually forwarded through the pipeline, etc.), which helps to ensure that the described techniques are correctly implemented. The description of the benchmarks along with some statistics are shown in Table 3. Table 4 shows other details regarding the processor configuration that we utilize in our simulations. We use the ADL simulator combined with CACTI [13, 14] for the energy evaluation to model processor energy. CACTI was used assuming a 32-nm CMOS process technology with **low standby power (LSTP)** cells and power gating. Table 5 shows the energy for accessing various components in the L1 DC and DTLB. Table 6 shows the dynamic energy for accessing various components of the DFC. CACTI does not provide energy values for very small caches. Thus, we estimated the energy for accessing the smaller DFCs by using the same rate of decrease in energy usage going from a 2KB L1 DC to a 1KB L1 DC with the same associativity and line size. Likewise, we made similar estimations of the energy usage for accessing DFC word metadata. The DFC metadata in the table includes the tag comparison along with accessing the word metadata. We do perform a DFC tag check whenever an L1 DC line is invalidated to determine which DFC lines need to be invalidated, and we account for this in the energy results. Leakage energy was gathered assuming a 1 GHZ clock rate. Our evaluation is for a simple low-end embedded processor. We did not simulate any memory hierarchy level after the L1 DC, as our approach will not affect the number of references to these other levels of the memory hiearchy.

Table 6. Energy for DFC Components

Component	LS+DP Config	Energy for Different DFC Sizes			
		128B	**256B**	**512B**	**1024B**
Read DFC Metadata	1	0.036 pJ	0.060 pJ	0.098 pj	0.162 pJ
	2xLS	0.039 pJ	0.065 pJ	0.109 pj	0.183 pJ
	2xLS+DP	0.040 pJ	0.068 pJ	0.116 pj	0.199 pJ
	4xLS	0.062 pJ	0.109 pJ	0.190 pj	0.332 pJ
	4xLS+DP	0.069 pJ	0.120 pJ	0.210 pj	0.367 pJ
Write DFC Metadata	1	0.143 pJ	0.169 pJ	0.199 pj	0.236 pJ
	2xLS	0.234 pJ	0.271 pJ	0.314 pj	0.363 pJ
	2xLS+DP	0.276 pJ	0.311 pJ	0.352 pj	0.397 pJ
	4xLS	0.395 pJ	0.471 pJ	0.561 pj	0.669 pJ
	4xLS+DP	0.405 pJ	0.485 pJ	0.582 pj	0.697 pJ
Read DFC Data		0.046 pJ	0.097 pJ	0.205 pj	0.434 pJ
Write DFC Data		0.126 pJ	0.240 pJ	0.455 pj	0.866 pJ

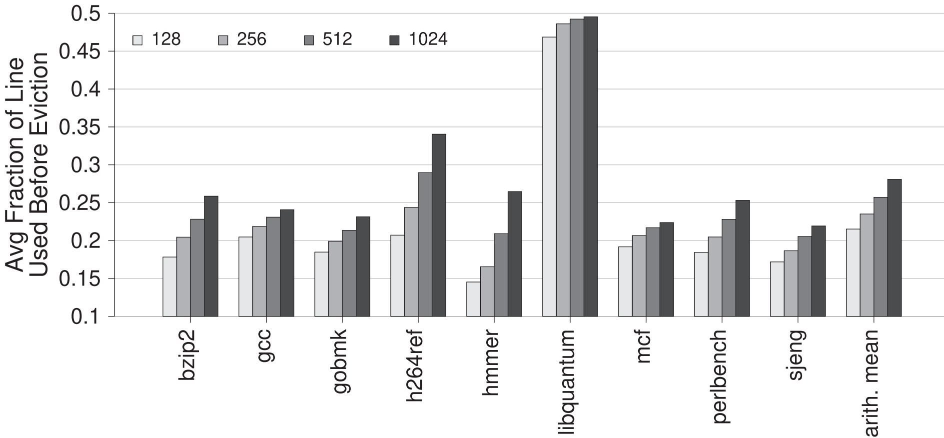

Fig. 9. Fraction of eagerly filled DFC lines referenced before eviction by benchmark.

5 RESULTS

This section describes the results of applying the proposed techniques in this article. We first show results of applying the lazily filling DFC data words, as described in Section 3.1. We next show results of sharing lines and packing data in a DFC, as described in Section 3.2. Finally, we show results after modifying the pipeline to speculatively perform a DFC tag check early and only access DFC data when a DFC hit has been guaranteed as described in Section 3.3. Since our combined techniques have no effect on performance, we do not provide any performance results.

Figure 9 shows the fraction of eagerly filled DFC lines with a 32B line size that are referenced before eviction by benchmark. The *libquantum* benchmark had the highest fraction of a line referenced before eviction, as many of its references are made when sequentially striding through a large structure. On average, the fractions are quite small due to frequent conflicts between DFC lines even though we counted the initial referenced word that caused the line to be filled as being referenced. Thus, even a 1,024B DFC with eager line filling has about 72% of the words within a line not referenced before the line is evicted.

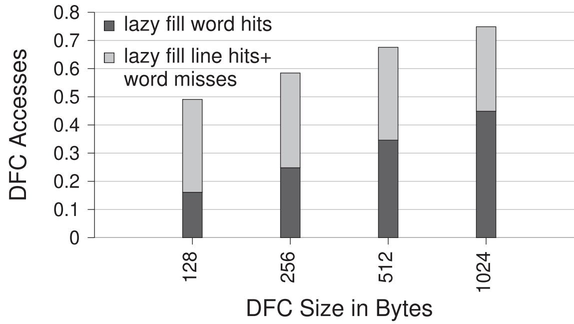

Fig. 10. Taxonomy of DFC accesses.

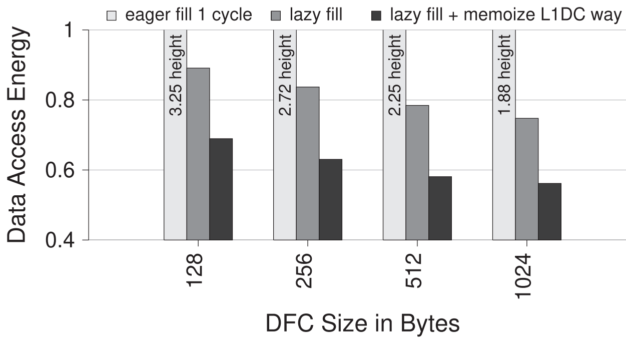

Fig. 11. Data access energy with eager vs. lazy filling.

Figure 10 shows the ratio of DFC lazy fill *word hits* and the ratio of DFC lazy fill *line hits+word misses*. Note that the height of each stacked bar is equal to the ratio of DFC *word hits* if a one-cycle eager line fill strategy was used. The space above the bar is the ratio of DFC lazy *line misses*. As the DFC size increases, the ratio of DFC lazy fill *word hits* increases, since there are fewer capacity miss evictions of DFC lines. The results show that a large fraction of the memory references do not need to access the L1 DC at all (*word hits*) and do not need to perform an L1 DC tag check and can access a single L1 DC data array when accessing the L1 DC (*line hits+word misses*).

Figure 11 shows the data access energy when eagerly filling an entire DFC line in a single cycle versus lazily filling the demanded word when the word is not resident in the DFC. Data access energy in this article is the energy used for accessing the DTLB, L1 DC, and DFC for both loads and stores. The baseline at 1.0 is for a processor without a DFC. We simulated L1 DC bitwidth configurations using the energy associated with a *DFC eager line fill strategy* (32-byte access that fills an entire DFC line in a single cycle) and a *DFC lazy word fill strategy* (4-byte access that fills a word in the DFC line only when the word is first referenced after the line is allocated). When eagerly filling an entire DFC line in a single cycle, the entire line from each L1 DC way must be read, resulting in more energy usage, as shown in Table 5. We label the height of the eager fill bars rather than showing the entire bar, since a DFC eager one-cycle filled line strategy uses significantly more data access energy than a processor without a DFC. These results confirm that a one cycle eager fill DFC approach is just not practical. For the *lazy fill* bar in the figure, an L1 DC tag check and associative L1 DC data array is accessed for each DFC *line hit+word miss*. For the *lazy fill + memoize L1 DC way* bar in the figure, no L1 DC tag check is performed and a single L1 DC array is accessed for each DFC *line hit+word miss*. For a 512B DFC, using lazy fill and

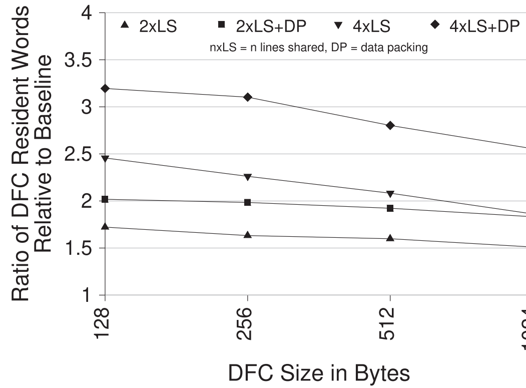

Fig. 12. DFC data utilization.

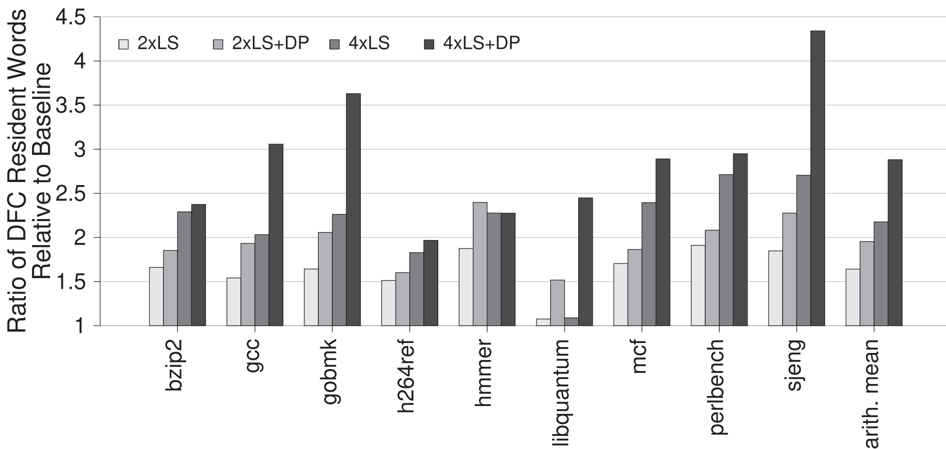

Fig. 13. DFC data utilization by benchmark for a 512B DFC.

memoizing the L1 DC way reduces data access energy for the DTLB, DFC, and L1 DC by about 42%, as compared to a baseline with no DFC.

Figure 12 shows the average utilization of words in the DFC for the different *line sharing* (LS) and *data packing* (DP) techniques. Utilization in this graph indicates the average number of resident words that are in the DFC after each 1,000 cycles. The results are compared to a baseline at 1.0 in the graph that represents a lazy word filled DFC with no line sharing or data packing. Line sharing and data packing are both shown to be quite effective for increasing the DFC utilization. However, as the DFC size increases, the relative utilization improvement decreases as compared to the baseline, since more of each application's working set fits into the DFC.

Figure 13 shows the utilization results by benchmark for a 512B DFC. In general, both increasing LS and DP improved the utilization. Note that improved utilization often does not result in comparable improvements in hit rates.

Figure 14 shows the fraction of memory references that are *word hits* and *line hits+word misses* in the DFC with different sizes and configurations. The space above the bars represent DFC *line misses*. Data packing increases the number of *word hits* within the line but does not typically change the number of *line misses*. Thus, the height of the bars with and without data packing are approximately the same when sharing the same number of L1 DC (sub)lines. As the DFC size

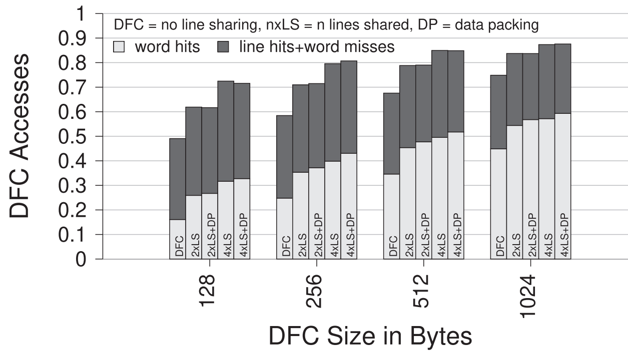

Fig. 14. DFC data hit rates.

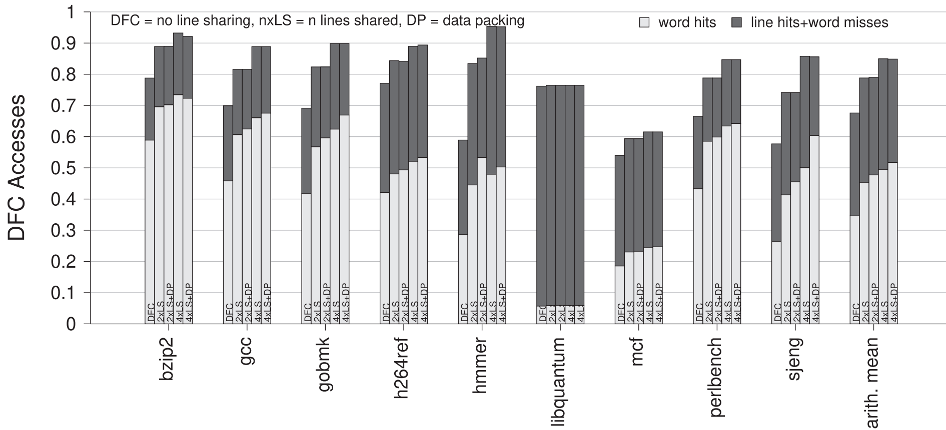

Fig. 15. DFC data hit rates by benchmark for a 512B DFC.

increases, the sum of the DFC *word hits* and *line hits+word misses* also increases. The figure also shows how valuable line sharing and data packing are for improving the DFC *word hit* rate. For instance, a 256 byte DFC that shares 4 L1 DC sublines with data packing (4xLS+DP) provides almost the same DFC *word hit* rate (42%) as a 1,024 byte DFC with no line sharing (43%). These results show that line sharing and data packing provide more flexibility so the DFC can often better adapt to the data reference patterns in an application. Compressing cache data has a greater opportunity to decrease the DFC miss rate compared to decreasing the miss rates of larger caches, since an application's working set is less likely to fit in a smaller uncompressed DFC.

Figure 15 shows the DFC data hit rates by benchmark for a 512B DFC. The *libquantum* benchmark had very little temporal locality, as most of its references were striding through a large structure. Thus, the various techniques did not increase the hit rate much, since there were few conflict misses and most hits were due to spatial locality.

Figure 16 shows the average data access energy usage for the different combinations of line sharing and data packing techniques varied for different DFC sizes. Data access energy includes the energy for all accesses to the DTLB, DFC, and L1 DC. The baseline at 1.0 is for a processor without a DFC. The figure shows that with smaller DFC sizes, energy usage is reduced when sharing more L1 DC lines and packing more data words as the DFC miss rate is more effectively lowered due to reducing contention for DFC lines. With larger DFC sizes, there is less contention for DFC lines

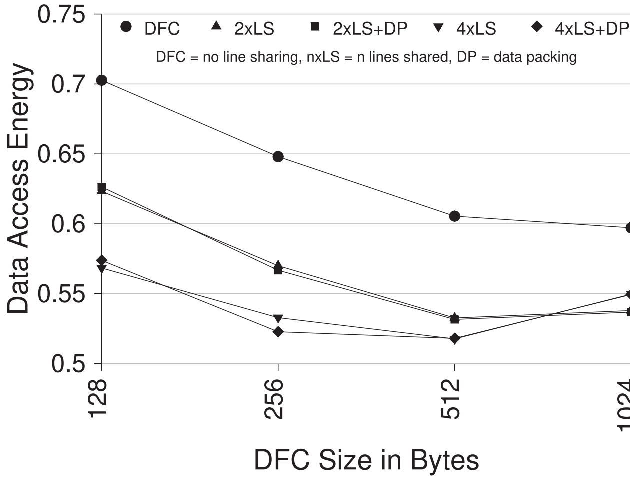

Fig. 16. Data access energy with line sharing and data packing.

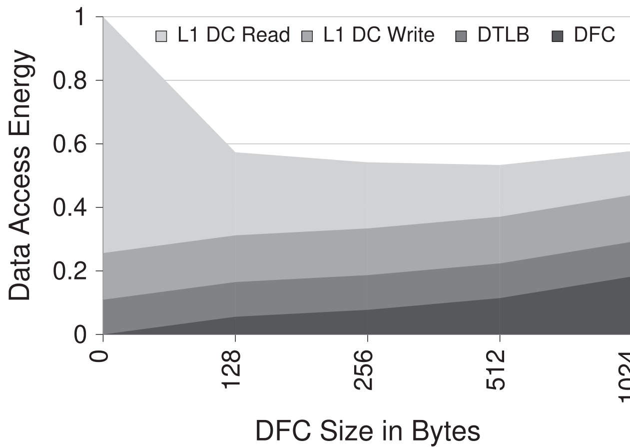

Fig. 17. Component data access energy.

and the extra overhead of additional DFC tag comparisons and word metadata accesses outweighs a smaller improvement in the DFC miss rate. The best data access energy usage is with sharing four lines (4xLS or 4xLS+DP) for a DFC size of 512 bytes. While Figures 12 and 14 showed that data packing improved both DFC utilization and hit rates, the energy used to access the word metadata mitigates most of those benefits. So the total energy usage with data packing was about the same as without data packing. Note that we started the y-axis at 0.5 and ended it at 0.75 so the reader can more easily distinguish between the 4xLS and 4xLS+DP lines in the figure.

Figure 17 shows the average data access energy usage for each component for the best configuration of line sharing and data packing with each DFC size, as shown in Figure 16. A DFC size of zero indicates no DFC was used. Using a DFC significantly reduces L1 DC read energy. Note that L1 DC write energy stays constant due to using a DFC write-through policy. Likewise, the DTLB energy stays constant as the DTLB is accessed for each memory reference. The DFC energy grows as the DFC size increases. One can see that the increase in DFC energy surpasses the decrease in L1 DC read energy when moving from a 512 byte to a 1,024 byte DFC. The 512 byte DFC has slightly lower total data access energy than a 256 byte DFC.

Table 7. DFC Metadata

Data Size	Organi-zation	Metadata						Metadata / Data Size
		tags	valid	word	L1DC way	LRU	Total	
128B	1xLS	100 bits	4 bits	32 bits	8 bits	0 bits	18 bytes	0.14
	2xLS	200 bits	8 bits	64 bits	16 bits	4 bits	36.5 bytes	0.29
	2xLS+DP	200 bits	8 bits	128 bits	16 bits	4 bits	44.5 bytes	0.35
	4xLS	400 bits	16 bits	128 bits	32 bits	20 bits	74.5 bytes	0.58
	4xLS+DP	400 bits	16 bits	256 bits	32 bits	20 bits	90.5 bytes	0.71
256B	1xLS	192 bits	8 bits	64 bits	16 bits	0 bits	27 bytes	0.11
	2xLS	384 bits	16 bits	128 bits	32 bits	8 bits	71 bytes	0.28
	2xLS+DP	384 bits	16 bits	256 bits	32 bits	8 bits	87 bytes	0.34
	4xLS	768 bits	32 bits	256 bits	64 bits	40 bits	145 bytes	0.57
	4xLS+DP	768 bits	32 bits	512 bits	64 bits	40 bits	177 bytes	0.69
512B	1xLS	368 bits	16 bits	128 bits	32 bits	0 bits	52 bytes	0.10
	2xLS	736 bits	32 bits	256 bits	64 bits	16 bits	138 bytes	0.27
	2xLS+DP	736 bits	32 bits	512 bits	64 bits	16 bits	170 bytes	0.33
	4xLS	1,536 bits	64 bits	512 bits	128 bits	80 bits	290 bytes	0.57
	4xLS+DP	1,536 bits	64 bits	1,024 bits	128 bits	80 bits	354 bytes	0.69
1KB	1xLS	704 bits	32 bits	256 bits	64 bits	0 bits	100 bytes	0.10
	2xLS	1,408 bits	64 bits	512 bits	128 bits	32 bits	268 bytes	0.26
	2xLS+DP	1,408 bits	64 bits	1,024 bits	128 bits	32 bits	332 bytes	0.32
	4xLS	2,816 bits	128 bits	1,024 bits	256 bits	160 bits	548 bytes	0.54
	4xLS+DP	2,816 bits	128 bits	2,048 bits	256 bits	160 bits	676 bytes	0.66

Table 7 shows the size of each DFC's metadata based on the size of the data and the organization. The largest contributing factor is the size of the tags. We are assuming a 32-bit address size and a 32 byte DFC line size. Thus, each tag in the 128B DFC is 25 bits (32 bits for the address size—2 bits for the set index—5 bits for the line offset). The number of tags and the number of valid bits are the number of lines of DFC data times the line sharing factor. The word metadata size for the 1xLS, 2xLS, 4xLS organizations is 1 bit for each data word in the DFC times the line sharing factor. Note the *f* (filled) bit serves this purpose for the 1xLS organization. The word metadata size for the 2xLS+DP and 4xLS+DP organizations is 2 bits for each data word in the DFC times the line sharing factor. Each L1 DC way is 2 bits, since the L1 DC in these experiments is 4-way set associative. Thus, the number of bits for the L1 DC ways will be twice the number of lines of DFC data times the line sharing factor. The size of the LRU information is the number of DFC sets * ceiling(log2(n!)), where n is the number of L1 DC (sub)lines that can map to each DFC line. The ratio of the metadata to the data size of the DFC shows that it increases with larger line sharing factors and data packing. Note the larger metadata sizes associated with using the techniques we evaluated did have an impact on the data access energy shown in Figure 16, but we were still able to obtain significant reductions in the energy usage of the L1 DC reads, as shown in Figure 17. Note also that even the largest DFC configuration is only 5% of the size of the L1 DC (34,128 bytes of data and metadata) that we simulated. The 512B DFC 4xLS+DP configuration provided the best energy results and was only 2.5% of the L1 DC size.

Figure 18 shows the taxonomy of loads accessing the DFC for different sizes. The *word hits* indicate how often the value was obtained from the DFC, which ranged from 14.3% for a 128B DFC to 27.0% for a 1,024B DFC. The *line hits+word misses* indicate how often the line was resident in the DFC, but the word was not resident. Note that a *line hit+word miss* means that the value can

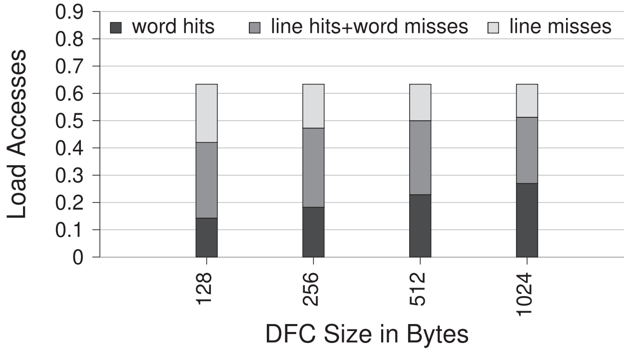

Fig. 18. Taxonomy of load accesses with guaranteed DFC hits.

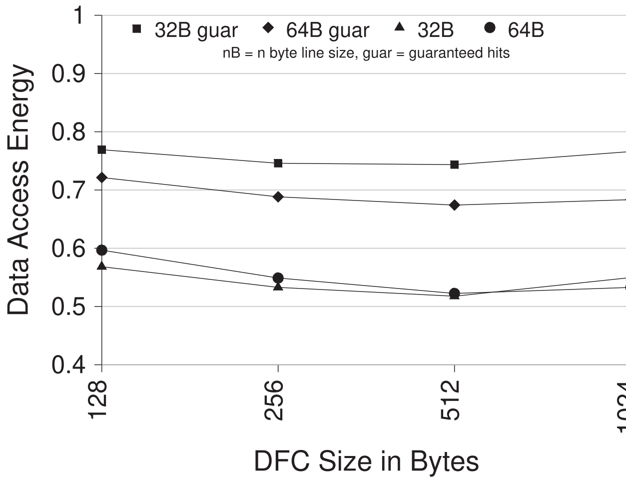

Fig. 19. Data access energy with guaranteed DFC hits.

be obtained from the L1 DC during the MEM stage without an L1 DC tag check and with accessing only a single L1 DC data array. A lazy fill *line hit+word miss* is equivalent to a first reference to a word that is a hit in an eagerly filled DFC line. The *line misses* indicate that the DFC tag and word metadata were accessed and that either the entire line was not resident or the speculative address generation caused the DFC block number to change, so the wrong DFC set was accessed. The sum of the three portions of the bar are the same regardless of the DFC size. This sum represents the fraction of loads when a speculative DFC tag access occurred. The space above each bar indicates that the DFC was not accessed due to the displacement of the load instruction being larger than the DFC line size.

Figure 19 shows the data access energy when accessing DFC data (1) only on guaranteed hits and (2) without speculation (accessing DFC metadata after address generation) for each DFC size with its best configuration. Data access energy includes the energy for all accesses to the DTLB, DFC, and L1 DC. All configurations shown share 4 L1 DC (sub)lines per DFC line and include data packing (4xLS+DP). Results are shown for both a 32 byte and 64 byte line size. The energy usage is higher with guaranteed hits (with speculation), which is due to two reasons. First, useless DFC and DTLB accesses occur due to speculation failures when the DFC block number was affected by the effective address calculation. Second, loading data that resides in the DFC cannot be exploited

when the displacement is larger than the DFC block size or when speculation failures occur. The data access energy using guaranteed hits is much less for a 64 byte line size, as there are fewer speculation failures and fewer displacements larger than 64 as opposed to 32. The data access energy reduction using guaranteed DFC hits is still significant. For a 64 byte line size, the energy reduction ranges from 27.9% for a 128B DFC to 32.6% for a 512B DFC. Note that a DFC without speculation will cause a performance degradation due to a one cycle miss penalty on DFC misses. A longer execution time will require more energy, which is not shown in Figure 19.

We have shown in this section that three complementary techniques can be utilized to make a DFC more effective. The following benefits were obtained for a 512B DFC, which was our most energy-efficient DFC size. By using a lazy DFC fill approach, we eliminate loads of words into the DFC that are never referenced before being evicted, which occurred for about 75% of the words in each line. By sharing and packing data words from multiple L1 DC lines to lower the DFC miss rate and storing the L1 DC way when a DFC word is not resident, we completely avoid accessing the L1 DC for loads about 23% of the time and avoid a fully associative L1 DC access for loads over 50% of the time. Our 512B DFC with metadata is only about 2.5% of the L1 DC size. By completely eliminating the DFC performance penalty by only accessing DFC data when a hit is guaranteed, we improve data access energy usage for the DTLB and L1 DC by 33% with no performance degradation. Assuming that 28% of the embedded processor energy is due to data supply [3], this would result in an overall reduction of about 9% for embedded processor energy.

6 RELATED WORK

There has been a number of prior studies on compressing data in first-level data caches or using common values to avoid data accesses in a first-level data cache. A small **frequent value cache (FVC)** has been proposed that is accessed in parallel to the L1 DC and is used to improve the miss rate as the FVC effectively functions as a victim cache [26]. The FVC is very compact, as it uses 3 bits to represent codes for seven frequent word values and an eighth code is used to represent that the value is nonfrequent and not resident in the FVC. The FVC was later adapted so a separate frequent value L1 DC data array was accessed first to avoid the regular L1 DC data array if the value was frequent [23]. This approach reduced L1 DC energy usage at the expense of delaying access to a nonfrequent value by a cycle. A **compression cache (CC)** that serves as a replacement for an L1 DC has also been designed to hold frequent values [25]. Each CC line holds either a single uncompressed line or two compressed lines when half or more of the words in the line are frequent values. A separate bit mask is used to indicate if a word value is frequent, and if not, then indicates where the word is located within the line. This cache design will result in a more complicated access and likely a longer access time. It appears in both the FVC and CC designs that a separate profiling run was performed to acquire the frequent values for each benchmark that was simulated, which will limit the automatic utilization of these designs. An approach similar to the CC was developed that allows two lines to be compressed into the size of one line, but does not increase the L1 DC access latency [17]. One proposed technique was to require that all words be representable in two bytes (narrow width) and another technique allows a couple of words in the two lines to not be narrow width where additional halfwords in the line are used to store the upper portions. Dynamic zero compression adds a **zero indicator bit (ZIB)** to every byte in a cache [22]. This ZIB is checked and the access to the byte is disabled when the ZIB is set. This approach requires more space overhead and may possibly increase the cache access time, but was shown to reduce energy from data cache accesses. The **zero-value cache (ZVC)** contains a bit vector of the entire block, where each bit indicates if the corresponding byte or word contains the value zero[8]. Blocks are placed in the ZVC (exclusive of the L1 DC) when a threshold of zero values for bytes/words within a block is met. The ZVC allows the access for loads and stores to

the L1 DC to be completely avoided. ZVC misses may result in execution time penalties, though the authors claim the ZVC is small enough to be accessed before the L1 DC with no execution time penalty. The same authors later developed the **narrow-width cache (NWC)** containing 8-bit values to reduce the miss rate [9]. The NWC is accessed in parallel to the L1 DC cache and blocks are placed in the NWC if a threshold of narrow values within the block is met. The NWC is used to reduce the miss rate. While all of these cache techniques either decrease the cache miss rate or reduce energy usage, we are unaware of any prior DFC or first-level cache technique that shares and packs data from different lines in the next level of the memory hierarchy into the same line at the current level of the memory hierarchy at the granularity of individual words without increasing the access time.

The conventional approach to support a fast L1 DC load access is to use a virtually indexed, physically tagged organization so in parallel the processor can access the DTLB to obtain the physical page number, access the L1 DC tag arrays to compare all the L1 DC tags in the indexed L1 DC set to the tag bits associated with the physical page number obtained from the DTLB, and access all the L1 DC data arrays to load a data word from each line within the L1 DC set. The data is selected based on which tag matches. This conventional approach is energy-inefficient, as the data can reside in at most one L1 DC way.

The **data cache access memoization (DCAM)** approach has been used to memoize the L1 DC way and DTLB way with the base register used to access memory [19]. This approach avoids an L1 DC tag and DTLB accesses when the calculated address shows the same cache line that was memoized with the base register. In addition, only a single data way within the L1 DC need be accessed when this information can be memoized. This approach does not work when the base register value is overwritten or when the displacement in the load or store instruction is too large. Our approach could be used in addition to the DCAM approach to avoid more L1 DC tag checks, DTLB accesses, and set-associative L1 DC data array accesses. Other approaches for memoizing the L1 DC way have also been used [12, 21].

Instead of using memoization, way prediction can be used to predict the L1 DC way to be accessed to reduce energy usage. If the way is correctly predicted, then only a single way of tag and data need to be accessed. Conventional way prediction accesses the tag and the data of the **most recently used (MRU)** way first and then accesses the remaining ways in parallel if there is a miss [7]. A technique that combines selective direct-mapping and way prediction where a predictor predicts if the line is in a direct-mapped position or set associative position [16]. Another approach used multiple MRU predictors by using the least significant bits of the address to disambiguate among the tags [24]. A final approach used another level of prediction for misses [27]. If a miss is predicted, then a phased cache access is performed where the tags are accessed before accessing the data. If a hit is predicted, then way prediction is used. Unlike our proposed approach, way prediction can increase the execution time when the prediction is incorrect. Way prediction is also complementary to our approach, as way prediction can still be used in an L1 DC when the value cannot be accessed from the DFC in the preceding cycle.

The speculative tag access approach is used to reduce the energy for accessing the L1 DC [2]. It speculatively performs an L1 DC tag check and DTLB access during the address generation stage using the *index* and *tag* fields from the base register value. If adding the displacement does not affect the *index* and *tag* fields, then only a single L1 DC data way is accessed during the MEM stage. We use a similar speculative tag access approach, but instead speculatively access the DFC tag and word metadata during the address generation stage to only access the DFC data during the MEM stage when a hit is guaranteed.

There have been a few DFC designs that have been proposed to eliminate the DFC miss penalty. Small alterations to the original FC design have been explored, where these new designs assume

that DFC tag comparison is performed within the execute stage after the effective address has been computed [4]. This approach requires a very small DFC and/or a slow clock rate to be feasible. The **practical DFC (PDFC)** speculatively performs a DFC tag check in parallel with the effective address generation [1]. The speculative DFC access is only attempted when the load or store displacement is small so the *DFC index* field is unlikely to be updated. The PDFC also assumed the DFC data could be accessed in the address generation stage, but after the computation of the *DFC offset* field. Another approach speculatively accesses a **filter data cache (FDC)** in the MEM stage when an FDC reference is predicted to hit by performing a partial tag comparison [12]. This approach results in a performance delay when the prediction is incorrect. A small overall execution time benefit was provided by assuming a two-cycle L1 DC access time so one-cycle FDC hits would improve performance. How a DFC or FDC line can be efficiently filled in a single cycle is not addressed in References [4] or [12]. In contrast to these approaches, our design can support a much larger DFC due to timing issues, since our DFC data access occurs in the MEM pipeline stage only after a DFC hit has been guaranteed in the EX stage.

Our line sharing approach is conceptually similar to a decoupled sector cache. A sector cache has sectors that consist of several contiguous cache blocks that are associated with a single address tag. A decoupled sector cache has multiple address tags for each sector and allows blocks to be associated with any one of these address tags [18]. A block in a sector cache is the size of the unit that is fetched from the next level of the memory hierarchy when the referenced block is not resident. We believe that decoupled sector cache organizations have not been successfully applied in DFCs or L1 DCs, as spatial locality needs to be exploited in small cache line sizes. In our line sharing approach, each DFC line can be viewed as a sector and each word in the DFC line as a block.

7 CONCLUSIONS

We have described a design that allows a DFC to reduce energy usage and not degrade performance. The following benefits were obtained for a 512B DFC, which was our most energy-efficient DFC size. We showed that a DFC lazy word fill approach is more energy-efficient than a DFC eager line fill approach. We are able to eliminate loads of words into the DFC that are never referenced before being evicted, which occurred for about 75% of the words in each line. We also demonstrated that it is possible to share and pack multiple L1 DC lines into a single DFC line at the granularity of individual words to improve the DFC hit rate, which results in completely avoiding L1 DC accesses for loads about 23% of the time and avoid a fully associative L1 DC access for loads about 50% of the time. This was accomplished with a DFC that is only about 2.5% of the size of the L1 DC. Finally, we improve data access energy usage for the DTLB and L1 DC by 33% with no performance degradation by speculatively performing a DFC tag check early and only accessing DFC data when a DFC hit is guaranteed.

ACKNOWLEDGMENT

We appreciate the suggestions by the reviewers that helped to improve this article.

REFERENCES

[1] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. 2013. Designing a practical data filter cache to improve both energy efficiency and performance. *ACM Trans. Archit. Compiler Optim.* 10, 4 (Dec. 2013), 54:1–54:25.

[2] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. 2013. Speculative tag access for reduced energy dissipation in set-associative L1 data caches. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'13)*. 302–308.

[3] W. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. 2008. Efficient embedded computing. *IEEE Comput.* 41, 7 (July 2008), 27–32.

[4] N. Duong, T. Kim, D. Zhao, and A. Veidenbaum. 2012. Revisiting level-0 caches in embedded processors. In *Proceedings of the IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, New York, NY, 171–180.

[5] S. Hines, P. Gavin, Y. Peress, D. Whalley, and G. Tyson. 2009. Guaranteeing instruction fetch behavior with a lookahead instruction fetch engine (LIFE). In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*. 119–128.

[6] S. Hines, D. Whalley, and G. Tyson. 2007. Guaranteeing hits to improve the efficiency of a small instruction cache. In *Proceedings of the ACM SIGMICRO International Symposium on Microarchitecture*. 433–444.

[7] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. 1999. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the IEEE International Symposium on Low Power Design (ISLPED'99)*. 273–275.

[8] Mafijul Islam and Per Stenstrom. 2009. Zero-value caches: Cancelling loads that return zero. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*. 237–245.

[9] Mafijul Islam and Per Stenstrom. 2010. Characterization and exploitation of narrow-width loads: The narrow-width cache approach. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. 227–236.

[10] J. Kin, M. Gupta, and W. H. Mangione-Smith. 1997. The filter cache: An energy efficient memory structure. In *Proceedings of the IEEE International Symposium on Microarchitecture*. 184–193.

[11] J. Kin, M. Gupta, and W. H. Mangione-Smith. 2000. Filtering memory references to increase energy efficiency. *IEEE Trans. Comput.* 49, 1 (Jan. 2000), 1–15.

[12] J. Lee and S. Kim. 2015. Filter data cache: An energy-efficient small L0 data cache architecture driven by miss cost reduction. *IEEE Trans. Comput.* 64, 7 (July 2015), 1927–1939.

[13] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. 469–480. DOI: https://doi.org/10.1145/1669112.1669172

[14] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. 694–701. DOI: https://doi.org/10.1109/ICCAD.2011.6105405

[15] Soner Önder and Rajiv Gupta. 1998. Automatic generation of microarchitecture simulators. In *Proceedings of the IEEE International Conference on Computer Languages*. Chicago, 80–89.

[16] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the ACM/IEEE IEEE International Symposium on Microarchitecture (MICRO'01)*. 54–65.

[17] Prateek Pujara and Aneesh. 2005. Restrictive compression techniques to increase level 1 cache capacity. In *Proceedings of the International Conference on Computer Design*. 327–333.

[18] Andre Seznec. 1994. Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio. In *Proceedings of the International Symposium on Computer Architecture*. 384–393.

[19] Michael Stokes, Ryan Baird, Zhaoxin Jin, David Whalley, and Soner Onder. 2019. Improving energy efficiency by memoizing data access information. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 6.

[20] W. Tang, R. Gupta, and A. Nicolau. 2001. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proceedings of the International Conference on Computer Design*. IEEE Computer Society, Washington, DC, 68–73.

[21] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis. 2019. Decoupled fused cache: Fusing a decoupled LLC with a DRAM cache. *ACM Trans. Archit. Compiler Optim.* 15, 4 (Jan. 2019), 23.

[22] L. Villa, M. Zhang, and K. Asanovic. 2000. Dynamic zero compression for cache energy reduction. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 214–220.

[23] Jun Yang and Rajiv Gupta. 2002. Energy efficient frequent value data cache design. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 197–207.

[24] C. Zhang, X. Zhang, and Y. Yan. 1997. Two fast and high-associativity cache schemes. *IEEE Micro* 17, 5 (1997), 40–49.

[25] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent value compression in data caches. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 258–265.

[26] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent value locality and value-centric data cache design. In *Proceedings of the International Symposium on Architecture Support for Programming Languages and Operating Systems*. 150–159.

[27] Z. Zhu and X. Zhang. 2002. Access-mode predictions for low-power cache design. *IEEE Micro* 22, 2 (2002), 58–71.

Received July 2020; revised January 2021; accepted February 2021