FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCE

TECHNIQUES TO REDUCE DATA CACHE ACCESS ENERGY USAGE AND LOAD DELAY HAZARDS

By

MICHAEL STOKES

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2019

Michael Stokes defended this dissertation on August 1, 2019.
The members of the supervisory committee were:

David B. Whalley
Professor Directing Thesis

Linda DeBrunner
University Representative

Xin Yuan
Committee Member

Gary Tyson
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with university requirements.

TABLE OF CONTENTS

ABSTRACT

Level-one data cache (L1 DC) accesses impact energy usage as they frequently occur and use significantly more energy than register file accesses. Modern processors use virtually-indexed, physically-tagged caches to reduce the L1 DC access time at the expense of increasing the energy to access it. It has been estimated that 28% of embedded processor energy is due to data supply [6]. In addition, level-one data caches have a significant impact on performance as a hit in the level-one data cache avoids accessing higher levels of the memory hierarchy, which typically have longer access times. Modern processors employ strategies such as *critical-word first* as well as *lockup-free caches* to limit the penalty of an L1 DC miss. However, as the issue-width of a processor is increased, the number of cycles that can be overlapped with a L1 DC line fill is decreased. This dissertation provides techniques that reduce both the energy usage of level-one data caches as well as improves the performance of processors by reducing the number of stalls due to loads and stores.

CHAPTER 1

INTRODUCTION

Level-one data cache (L1 DC) accesses impact energy usage as they frequently occur and use significantly more energy than register file accesses. Level-one data caches are typically set-associative and virtually-indexed, physically-tagged (VIPT). Making a cache set-associative reduces its miss rate while making it VIPT reduces its access time at the expense of increasing the energy required to access it. As the number of transistors placed onto a chip increases according to Moore's Law, the energy expended by simultaneously switching these transistors also increases. As a result, the overall temperature of the chip increases. The processor's clock rate must be constrained to avoid damaging the chip, reducing the chip's power density at the expense of performance, a phenomenon known as the *power wall.* It has been estimated that 28% of embedded processor energy is due to data supply [6]. Thus, reducing data access energy on such processors is a reasonable goal.

In addition, level-one data caches have a significant impact on performance as a hit in the level-one data cache avoids accessing higher levels of the memory hierarchy, which typically have longer access times. Even so, hits in the level-one data cache can still cause *load-delay hazards* when a subsequent instruction demands a value before it can be loaded from the data cache. This problem is exacerbated as modern processors increase the number of cycles required to access the level-one data cache. Modern processors employ strategies such as *critical-word first* as well as *lockup-free caches* to limit the penalty of an L1 DC miss. However, as the issue-width of a processor is increased, the number of cycles that can be overlapped with a L1 DC line fill is decreased.

This dissertation provides techniques that reduce both the energy usage of level-one data caches as well as improves the performance of processors by reducing the number of stalls due to loads and stores. Chapter 3 introduces a non-speculative technique that allows a processor to directly access a set-associative data cache, allowing the processor to 1) avoid accessing the ways of a set that don't hold the data, 2) avoid the DTLB, and 3) avoid the L1 DC tag array. Chapter 4 proposes a technique that not only removes the performance penalty associated with level-zero data caches, but uses level-zero data caches to improve performance by reducing the number of load-delay hazards. In Chapter 5, detailed findings are introduced showing that *word-filled* level-zero data caches are able to save more energy than *line-filled* level-zero data caches. Chapter 5 then provides two methods, *line sharing* and *data packing*, that increase the hit rate of level-zero data caches by significantly increasing the amount of data that can be stored. Chapter 5 goes on to provide a method of utilizing a word-filled, line sharing and data packing level-zero data cache without the performance penalty typically associated with level-zero data caches. Chapter 6 extends the methods shown in Chapter 5 to the level-one data cache. A thorough analysis of line sharing and data packing's effect on various level-one data cache designs is conducted, varying the size, associativity, and the sub-block size, data packing methods, as well as the amount of L2 DC (sub)lines that share an L1 DC line.

CHAPTER 2

BACKGROUND

To understand the following dissertation, it's important to know the purpose and functionality of the level-one data cache (L1 DC).

2.1 The Memory Hierarchy

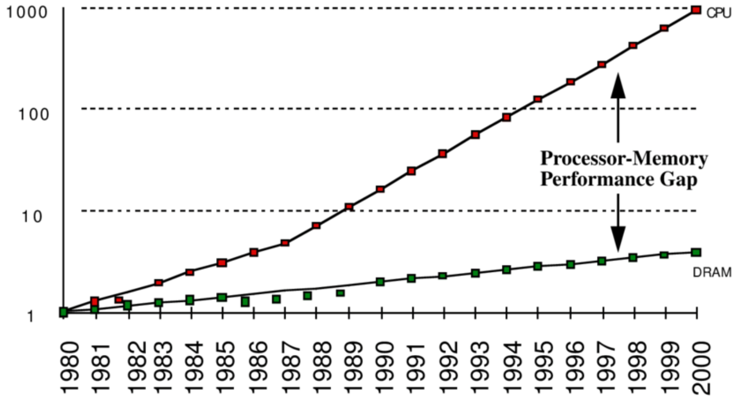


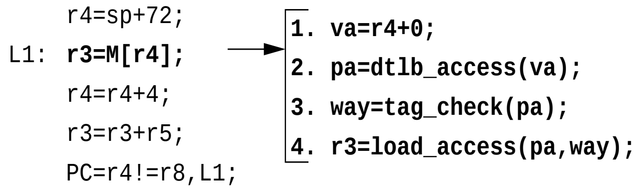
Figure 2.1: The Processor-Memory Performance Gap

Since 1980, processor speeds have increased year-over-year by roughly 60% while DRAM speeds have only increased at a rate of only 7%, a problem known as the processor-memory gap. Processors take advantage of both *temporal locality*, where data that is referenced is likely to be referenced again in the near future, as well as *spatial locality*, where data in proximity to referenced data will likely be referenced as well. To address the processor-memory performance gap, multiple levels of cache are now used. Level-one (L1) caches in the memory hierarchy are large enough to provide reasonable hit rates, but also small enough to provide fast access times. Level-two (L2) and level-three (L3) caches provide slower access times, but reduce the number of references that need to access main memory. These multiple levels of cache now all reside on the same chip as the processor and are built using SRAM technology. In addition to this, CMOS scaling trends result in faster transistors with with relatively longer wire delays. Because of these reasons, level-one data cache sizes have largely remained the same and are now pipelined to keep up with reduced clock cycle times.

Upon a cache miss, a (sub)line is fetched from higher levels of the memory hierarchy and placed into the cache. Where a cache line can be placed is determined by its *associativity.* In a *fully associative* cache, a (sub)line can be placed in any cache line of a cache. In an *n-way set associative* cache, a cache is split into sets with n cache lines per set and a (sub)line can be placed inside any one of the n ways of a set. A *direct mapped* cache is a special case of a set-associative cache where n is equal to one. By restricting the cache lines where a (sub)line can be placed, both the time and the energy to access the cache decrease at the expense of also decreasing the hit rate.

2.2 Anatomy of Memory Access Operations

Contemporary architectures designed using RISC principles attempt to implement each instruction using a single μop. However, memory operations involve many hidden hardware μops. These μops not only form dependence chains, but also use a significant amount of energy.



```
    r4=sp+72;
L1: r3=M[r4];     --> 1. va=r4+0;
    r4=r4+4;          2. pa=dtlb_access(va);
    r3=r3+r5;         3. way=tag_check(pa);
    PC=r4!=r8,L1;     4. r3=load_access(pa,way);
```

Conventional Micro Operations

Figure 2.2: Micro-Ops Associated with Load Instructions

Figure 2.2(a) shows code containing a load and a store along with the μops that implement these instructions. The load μops are: #1 Add the base register value and the offset to obtain the virtual address (*va*); #2 Access the data translation lookaside buffer (DTLB) using the *va* to get the physical address (*pa*); #3 Perform the tag check to identify the *way* where the data resides in a set-associative cache; and #4 use the *pa* index and the *way* to access the cache data and update the register. Unfortunately, these μops are not visible to the compiler with conventional ISAs and it would be expensive to implement each μop as an ISA instruction in terms of code size, fetch bandwidth, and energy.

2.3 Virtually Indexed, Physically Tagged Caches

Several hardware techniques shorten this dependence chain and mitigate the delay, but do so at the expense of significantly more energy usage and/or imposed constraints, such as a limited page size. For example, virtually-indexed, physically-tagged (VIPT) caches exploit the fact that the cache index remains invariant during translation with appropriately sized pages, allowing μops (2), (3), and (4) shown in Figure 2.2(a) to proceed in parallel by simultaneously accessing all ways of data in the L1 DC set at the expense of significant energy usage. This approach leaves the dependence between the first and the remaining three μops into successive pipeline stages such that the execution unit performs the virtual address computation (i.e., μop #1) and the memory access stage performs all the remaining operations, leading to the infamous

Figure 2.3 depicts how a classical in-order pipeline performs a load from an n-way set-associative L1 DC. The virtual memory address is generated by adding a displacement to a base address obtained from the register file in an address generation (ADDR-GEN) stage. The displacement is a sign-extended immediate and the base address is obtained from the register file. In the SRAM-ACCESS stage the DTLB, the L1 DC tags, and the L1 DC data are all accessed in parallel to minimize load hazard stalls and the tag value of the physical address is compared to the tag value of the physical page number from the DTLB.[1] This organization is energy inefficient as all data arrays are accessed, but the value can reside in at most one way within a cache set.

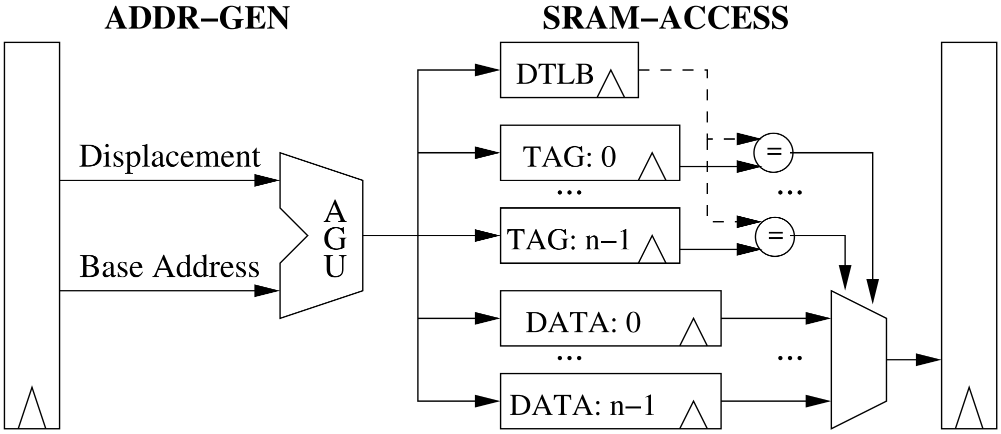


Figure 2.3: Conventional L1 DC Pipeline Load Access

[1] The register level after the ADDR-GEN stage is embedded in the DTLB, TAG, and DATA blocks.

Figure 6.1 shows the address fields used to access the DTLB and the L1 DC. The virtual page number is used to access the DTLB, which produces the corresponding physical page number. The virtual and physical page offsets remain the same. The L1 DC block number uniquely identifies the L1 DC line being accessed. The *L1 DC offset* indicates the first byte of the data to be accessed in the L1 DC line. The *set index* is used to access the L1 DC set. The *tag* contains the remaining bits that are used to verify if the line resides in the L1 DC.[2]

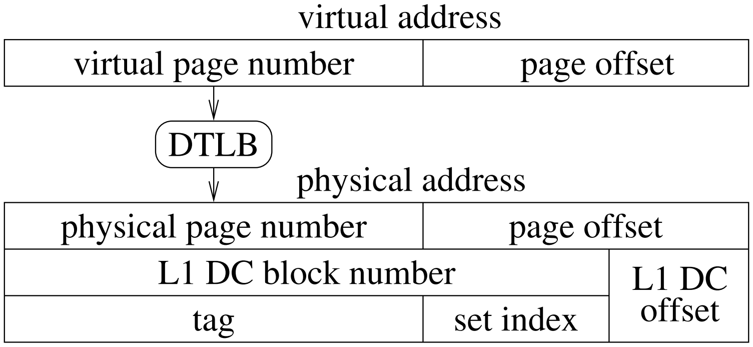


Figure 2.4: Address Fields

2.4 Level-Zero Data Caches

A level-zero data cache (L0 DC), also known as a data filter cache (DFC), has been shown to be effective at reducing data access energy [14,15]. An L0 DC is a smaller, typically direct-mapped cache that is accessed before the L1 DC. A reference that hits in the L0 DC does not need to access the L1 DC while a reference that misses in the L0 DC accesses the L1 DC in the following cycle. An L0 DC is energy efficient since a large fraction of the memory references can be serviced from the L0 DC that is much smaller than a level-one data cache (L1 DC), resulting in less energy usage for each L0 DC reference as compared to an L1 DC reference. However, a conventional L0 DC has disadvantages that has discouraged its adoption in contemporary processors. First, an L0 DC can cause a performance penalty as it has to be accessed before the L1 DC in order to reduce energy usage; upon an L0 DC miss, the L1 DC is accessed a cycle later than it normally would, potentially causing load-delay hazards that would not occur had an L0 DC not been used. This increase in

[2] We depict the *physical page number* and the *tag* fields being the same size, but the *physical page number* could be smaller for a virtually-indexed, physically-tagged (VIPT) cache. To simplify the description, we assume these two fields are the same size.

execution time will mitigate some of the energy benefit of using an L0 DC. Second, a single cycle L0 line fill as proposed in many prior studies [7,9,10,14,15,27] has been shown to be unrealistic as it can adversely affect L1 DC area and energy efficiency [3].

2.5 Impact of L1 DC Misses on Processor Performance

L1 DC misses decrease performance as the data must be fetched from higher levels of the memory hierarchy. Higher levels of the memory hierarchy typically take many more cycles to access than the L1 DC. In addition, caches in embedded processors are typically blocking, meaning that memory operations are stalled while an L1 DC line is being filled. One technique mitigates this delay by servicing a load that missed in the L1 DC as quickly as possible by fetching the requested data first. This allows the load to complete faster than if it waited for the entire L1 DC line to be filled. While the remainder of the L1 DC line is being filled, the processor can continue issuing new instructions. However, if a memory-accessing instruction is issued before the L1 DC line is filled, then the pipeline must stall before the line fill is completed. This decreases the latency of an L1 DC miss penalty as well as overlaps the L1 DC line fill with non-memory access operations.

2.5.1 Sub-blocking

One technique embedded processors use to decrease the miss penalty of L1 DCs as well as the energy they consume is to decrease the number of words fetched during an L1 DC line fill. L1 DC lines are split into sub-blocks and line fills occur at the granularity of sub-blocks: when an L1 DC miss occurs, only the sub-block containing the requested word is filled. This means that some sub-blocks of an L1 DC will remain empty as they will not be referenced before the line is evicted. Sub-blocking increases the miss rate as filling an entire L1 DC line captures more spatial locality but decreases the number of words needlessly fetched from higher levels of the memory hierarchy. In addition, it takes fewer cycles to fill a sub-block as fewer words need to be fetched from higher levels of the memory hierarchy.

CHAPTER 3

IMPROVING ENERGY EFFICIENCY BY MEMOIZING DATA ACCESS INFORMATION

Level-one data cache (L1 DC) and data translation lookaside buffer (DTLB) accesses impact energy usage as they frequently occur and each L1 DC and DTLB access uses significantly more energy than a register file access. Often, multiple memory operations will reference the same cache line using the same register, such as when iterating through an array. A technique is proposed in this chapter to memoize L1 DC access information, such as the L1 DC data array way and the DTLB way, by associating this information with the register used to access it. When a load or store calculates the effective address by adding the base register with the displacement value, the processor detects whether the effective address shares the cache line memoized with the base register. If so, the L1 DC tag array access and the DTLB access to determine the L1 DC way are avoided and instead the memoized information is used. In addition, only a single data array way in a set-associative L1 DC needs to be accessed during a load instruction when the L1 DC way has been memoized. This nonspeculative memoization approach provides existing executables a significant reduction in data access energy usage compared to a conventional cache and provides even greater energy usage reduction after way prediction is applied when memoized information is unavailable.

3.1 Introduction

Level-one data cache and data translation lookaside buffer accesses frequently occur and each of these accesses use significantly more power than a register file access. It has been estimated that 28% of embedded processor energy is due to data supply [6]. Thus, reducing data access energy on such processors is a reasonable goal.

The tag arrays and data arrays of an L1 DC can be accessed in parallel for load instructions to improve the latency of obtaining data from the L1 DC, which is sometimes referred to as a *conventional* cache [19]. The tag arrays are often accessed before the data arrays of level-two (L2) and level-three (L3) caches to reduce energy usage, which is sometimes referred to as a *phased*

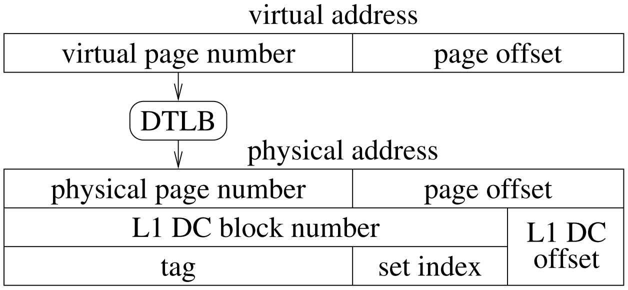


Figure 3.1: Address Fields

cache [19]. The advantage of a phased cache is that at most a single data array need be accessed as the result of the tag check will be known when the data in the cache is accessed. However, using a phased L1 DC is often impractical since the reduced energy usage for the phased L1 DC data accesses would be largely offset by the increased energy required for longer execution times.

This dissertation proposes the *Data Cache Access Memoization* (DCAM) technique to retain data access information so that subsequent memory accesses dereferencing the same register can often more efficiently access the L1 DC. These efficient L1 DC accesses are achieved by associating the L1 DC way and DTLB way with the base register of a memory reference. When the processor detects that a subsequent memory reference will reference the same L1 DC line, the processor can use the memoized information to avoid the L1 DC tag check, avoid the DTLB access, and access only a single data array in a set-associative L1 DC organization. When memoization information cannot be utilized for the base register, the default L1 DC access mechanism (e.g. conventional [19] or way prediction [11, 22]) can be used.

The contributions of this dissertation are as follows. (1) We show that simple and efficient memoization techniques that associate data access information with the base register being dereferenced can often be utilized without ISA changes to significantly reduce data access energy usage. (2) We provide a simple method that allows the data access information to be restored even after other instructions update the base register value. (3) We show that energy usage can be further reduced when data access information is unavailable for the nonspeculative DCAM approach by applying a speculative approach, such as way prediction.

3.2 Memoizing L1 DC and DTLB Information

The L1 DC way and DTLB way must be stored in a structure to allow reuse of data access information. In fact, a DTLB access and L1 DC tag check will often be redundant since the same line may be accessed again. Figure 3.2(a) shows code for loading from and storing to the same variable. The store can use the same L1 DC way as the load instruction since the value of `r6` has not been changed. Figure 3.2(b) shows an example of accessing sequential array locations, where an L1 DC line is likely to be repeatedly accessed.

```
r6=...;[pam]
...
...=M[r6];
...
M[r6]=...;
```

(a) Redundant Accesses

```
   r20=...;[pam]
L3:r2=M[r20];
   ...
   r20=r20+4;[pam]
   PC=r20!=r21,L3;
```

(b) Strided Accesses

Figure 3.2: Memoization Examples

One issue that must be resolved is when the displacement in the load or store instruction is a nonzero value. Figure 3.3 shows the average frequency of the number of bits needed to represent displacement values (the most significant 16 bits are sign extended to be all 0's or 1's) for load and store operations in the MiBench benchmark suite, where the range for n bits is $-2^{n-1}..2^{n-1}-1$ and does not comprise the values in the previous range. A zero displacement occurs 46% of the time and large offsets comprise a small fraction of the displacements. Note that negative displacements occur less than 2% of the time.

One problem is that the address associated with the base register value may not be associated with the same L1 DC line as the effective address that is computed by adding the base register and the displacement value. For a load or store instruction to be able to use or memoize cache access information, the magnitude of the displacement must be smaller than the L1 DC line size. However, the effective address of a load or store instruction with such a displacement may still fall outside of the cache line associated with the base register. If the displacement is positive and is smaller than the cache line size, then the effective address must point to either the current or next sequential cache line. Thus, the processor tracks both the current and the next sequential L1 DC line associated with the address in the base register, which allows dealing with small positive displacements that cross to the next sequential line in memory.

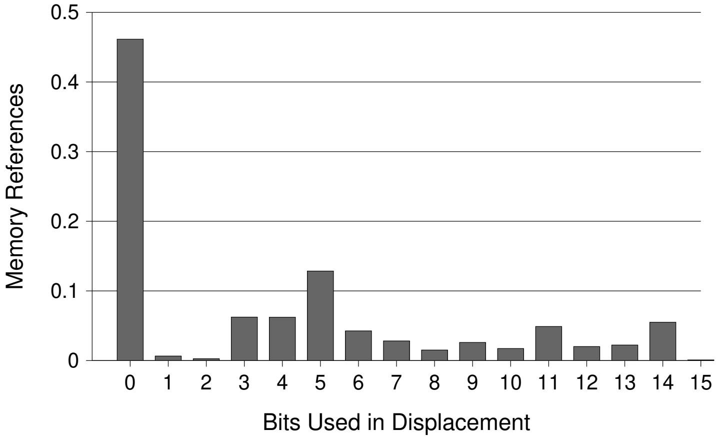
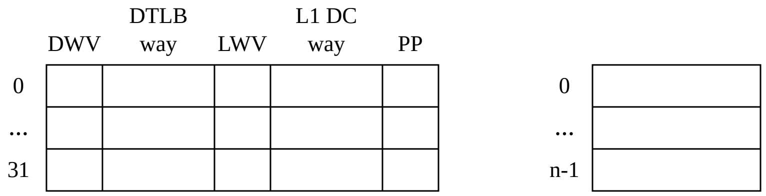


Figure 3.3: Displacement Size Frequency

The DCAM approach associates L1 DC access information with the base register number of a load or store instruction and detects when updates to this register does not invalidate this information. Consider the data cache access structure (DCAS) in Figure 3.4(a) that contains fields associated with each base register in load and store instructions. The *DWV* bit indicates if the *DTLB way* field is valid. If the *DWV* bit is not set, then the rest of the DCAS entry is considered invalid. The *DTLB way* field holds the DTLB way in which the associated physical page number resides. The *LWV* bit indicates if the *L1 DC way* field associated with the address in the base register is valid. The *L1 DC way* field holds the L1 DC way in which the cache line resides that is associated with the address in the register. The *LWVN* bit indicates if the next sequential line has a valid way. The *L1 DC N way* holds the way for the next sequential line. The L1 DC *set index* field (see Figure 3.1) of the effective address indicates the L1 DC set and need not be stored in the DCAS since the set index is available from the effective address calculation. The *PP* field contains page protection bits from the DTLB entry since the DCAS structure allows DTLB references to be avoided and these bits need to be checked to ensure pages are properly accessed. The DCAS entry needs to be accessed during the EX stage to allow a single L1 DC data array access for a load in the following cycle.

Figure 3.4(b) depicts the DCAV structure used to keep DCAS entries coherent when an L1 DC line is evicted or invalidated. Each DCAV entry contains a bit vector, where each bit represents an integer register. An entry is indexed by the *L1 DC way*, where n is the L1 DC associativity level.



(a) Data Cache Access Structure (DCAS)

(b) Data Cache Access Valid Information (DCAV)

Figure 3.4: Data Cache Access Information

Each time a DCAS entry shown in Figure 3.4(a) is associated with a line, the bit corresponding to the register number of that way in the DCAV structure is set. Each time a register's *LWV* bit (see Figure 3.4(a)) is cleared, the bit corresponding to that register number is also cleared in every DCAV entry. When an L1 DC line is replaced or invalidated, the corresponding bits set in the entry accessed by the *L1 DC way* of that line are used to determine which DCAS entries will have their *LWV* bit cleared. Thus, this structure contains an inverse mapping between each L1 DC way and the DCAS entries. All the DCAS *DWV* bits and the values in the DCAV structure are cleared upon a DTLB eviction, which infrequently occurs.

3.3 Detecting DCAS Re-Use

There are many cases where the address in a register is updated, but still is within the same line in the cache and more frequently within the same page. Figure 3.5 shows that it is simple for the processor to detect if the cache line to be accessed will change during an effective address computation of a load or store instruction (*M[rs+immed]*) or during an integer immediate addition (*rd = rs + immed*). First, the magnitude of the immediate has to be less than the size of the *line offset* field.

Second, the carry out values can be inspected during the addition to check whether or not the *L1 DC block number* as shown in Figure 3.1 has changed. If the *set index* field is updated during a load or store address computation with a positive displacement that is smaller than the L1 DC line size, then either the *L1 DC N way* field can be used or the tag check has to be performed if the *LWVN* bit is clear. In the latter case, a single way in the DTLB can be accessed using the *DTLB way* field to obtain the physical tag value when the *virtual page number* (*VPN*) field is not updated. If the VPN field is updated, then all the ways in the DTLB have to be accessed. If the *set* *index* field is updated during an integer addition instruction by a small positive value, then the *L1 DC way N* field is copied to the *L1 DC way* field and the *LWVN* bit is cleared. By inspecting the carry out values for integer add or subtract operations using either two register values or register and an immediate, the processor can continue to memoize all or portions of a register's data access information after updates to the base register if the update does not change the cache line or page associated with the address contained in the register.

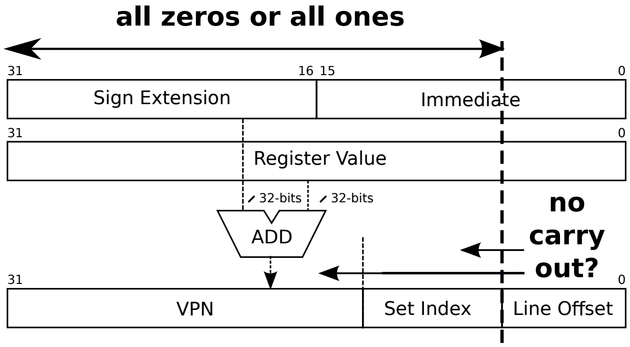


Figure 3.5: Detecting Address Changes

If an integer add instruction references a source register with its *DWV* bit set, then its corresponding DCAS information is copied to the destination register DCAS entry if the destination register differs from the source register. Other integer register updates cause the DWV field in the DCAS entry indexed by the destination register number to be invalidated.

Figure 3.6 shows the percentage reduction of L1 DC tag array and DTLB accesses as a result of using the LWVN field. Benchmarks such as *blowfish* with larger offset sizes than average tend to have a significant improvement. Benchmarks such as *adpcm* saw almost no improvement as virtually all loads and stores used a displacement value of zero.

3.4 The DCAS Refresh Buffer

Frequently, a DCAS entry is invalidated but its contents continue to point to the correct cache line. In Figure 3.7 DCAS entry 20 is set during the load instruction and is overwritten during the function call to foo, shown in Figure 3.9(a). During foo's epilogue code, `r20`'s value is restored, again pointing to the same cache line in its DCAS entry. If the processor can detect during a load or a store that the base register's DCAS entry points to the same cache line as the value held inside the base register, then the processor can restore the DCAS entry contents.

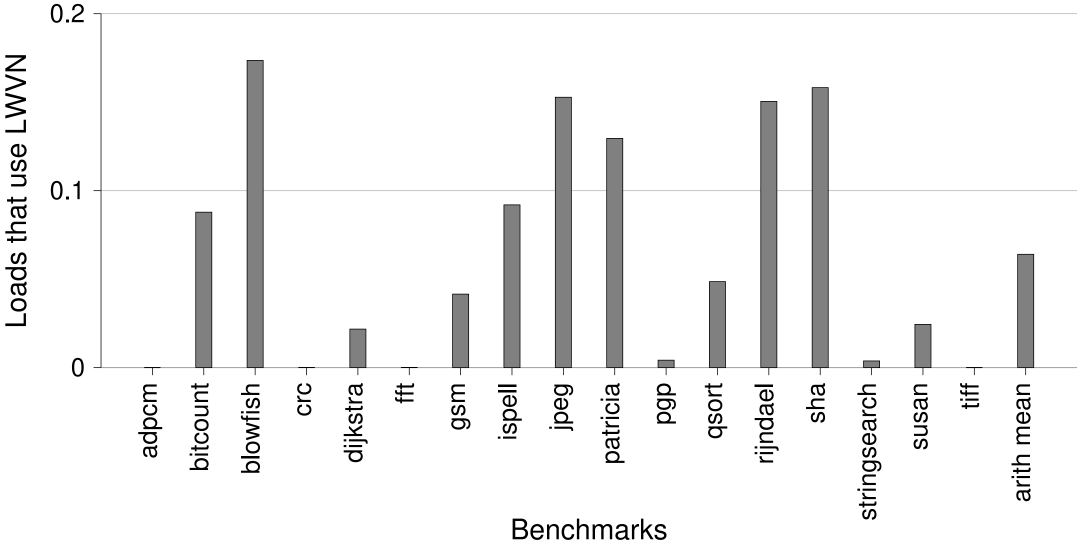


Figure 3.6: Reduction of L1 DC Tag and DTLB Accesses Using LWVN

```
L3: r2=M[r20];            foo: sp=sp-12;
       ...                     M[sp+4]=r20;
    jal foo                       ...
       ...                     r20=0;
    r20=r20+4;                    ...
    PC=r20!=r21,L3;            r20=M[sp+4];
                               sp=sp+12;
                               jr ra
```

Figure 3.7: DCAS Refresh Example

The processor stores the tag and set index portions of the virtual address of the L1 DC line with a DCAS entry in addition to its L1 DC access information. If a load or store detects that its DCAS entry is invalid but its contents still refer to the cache line associated with the tag and set index stored alongside it, then the processor compares the virtual tag and set index portions of the base register with the virtual tag and set index portions stored alongside the DCAS entry during the EX stage. If they match, then the processor can restore the *DWV*, *DTLB way*, *LWV*, *L1 DC way*, *LWVN*, and *L1 DC way next* fields if they were previously valid. Furthermore, if the DCAS entry and base register don't point to the same cache line but do point to the same page, then the processor can restore the DCAS entry's *DWV* and *DTLB way* fields to avoid a fully associative DTLB access.

DCAS entries can now be in one of three states: 1) *valid*, meaning the DCAS entry and base register value point to the same cache line and/or page and that the way is known, 2) *false invalid*, meaning the DCAS entry and base register value may not point to the same line or page but the DCAS information is still valid for the line and page stored in the virtual tag and set index fields of the DCAS entry, and 3) *true invalid*, meaning the DCAS entry has no valid cache access information.

A DCAS entry becomes *valid* after a load or a store instruction determines the L1 DC way (DTLB way) and the effective address points to the same line (page) in the base register value. A DCAS entry becomes *true invalid* after an L1 DC line eviction or a DTLB page eviction. A DCAS entry becomes *false invalid* if the base register is overwritten by an instruction that doesn't change its DCAS information. For example, after instruction `r20=0;` executes in Figure 3.7, the DCAS contents still refers to the same DTLB way and L1 DC way shown in Figure 3.9(b). The DWV field is marked as *false invalid*, indicating that the DCAS cannot guarantee that the base register contents and DCAS entry refer to the same cache line, but it can guarantee that the DCAS entry is still *valid* for the stored tag and set index. The next time a load or store refers to a DCAS entry marked as *false invalid*, the virtual tag and set index fields of the base register are compared with those fields stored in the DCAS Refresh Buffer to see if the DCAS contents can be restored by setting the DWV field to *true valid* as shown in Figure 3.9(c). As the DCAS and the DCAS Refresh Buffer are both indexed by the base register number, the cost of accessing this buffer is relatively inexpensive.

Figure 3.8 shows the percentage reduction in the number of L1 DC tag array and DTLB accesses made when the DCAS refresh buffer is used. On average, this approach reduces the number of L1 DC tag array and DTLB accesses by 10.6%. Benchmarks like bitcount which update a global variable inside of a loop with a function call, and therefore must continually re-load the address, has a significant improvement of over 48%. As the refresh buffer is directly accessed and only used when the DCAM entry is marked as *false invalid*, it will not expend a significant amount of energy, mitigating the energy savings.

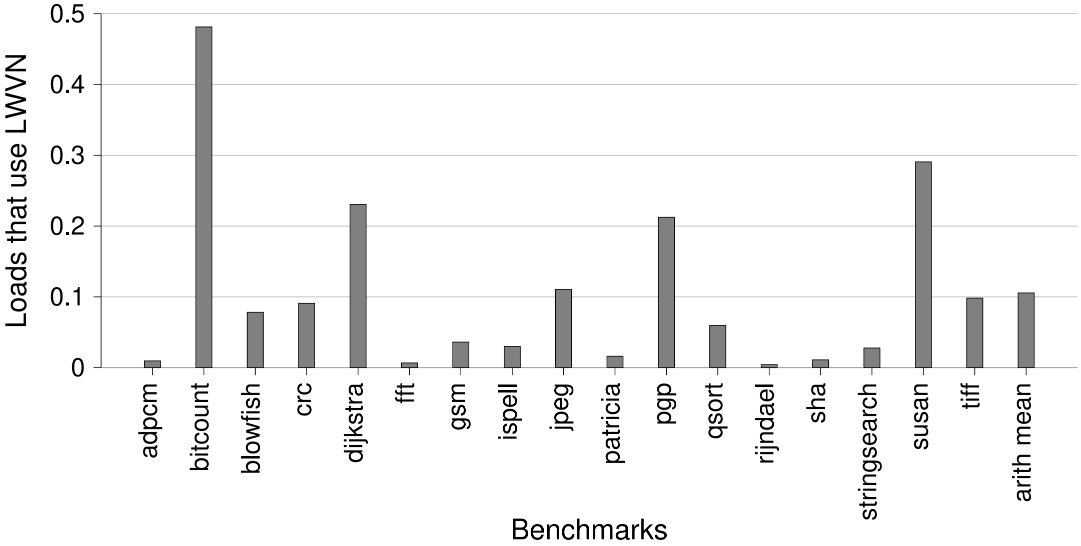
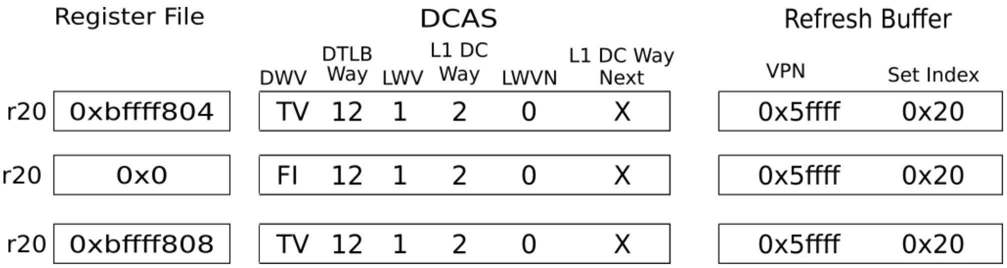


Figure 3.8: DCAS Refresh Buffer Effectiveness



	Register File	DCAS						Refresh Buffer	
		DWV	DTLB Way	LWV	L1 DC Way	LWVN	L1 DC Way Next	VPN	Set Index
r20	0xbffff804	TV	12	1	2	0	X	0x5ffff	0x20
r20	0x0	FI	12	1	2	0	X	0x5ffff	0x20
r20	0xbffff808	TV	12	1	2	0	X	0x5ffff	0x20

Figure 3.9: DCAS Refresh Buffer Example

3.5 Evaluation Framework

In this section the experimental environment is described. The seventeen benchmarks from the MiBench benchmark suite [8], which is a representative set of embedded applications, are used to evaluate the DCAM approach. All benchmarks are simulated using the large dataset option and compiled using gcc with the *-03* option.

The ADL simulator [21] was used to simulate both a conventional MIPS processor as the baseline and the modified processor as described in this dissertation. Both configurations are single-issue, in-order processors with six-stage pipelines as shown in Table 3.1. Table 3.2 shows other details regarding the processor configuration utilized in these simulations. Note that the DWV bit is separated from the rest of the DCAS structure so this bit can be accessed during the RF (register fetch) pipeline stage, which allows the processor to avoid accessing the rest of the DCAS structure when the DWV bit is not set.

Table 3.1: DCAM Pipeline Stages

Stage	Name	DCAM Pipeline
IF	Inst. Fetch	
ID	Inst. Decode	
RF	Reg. Fetch	Read DWV Bit
EX	Execute	Read/Refresh DCAS
MEM	Mem. Access	Update DWV/DCAS
WB	Write Back	

Table 3.2: Processor Configuration

page size	8KB
L1 DC	32KB, 64B line size 4-way associative, 1 cycle hit, 10 cycle miss penalty
DTLB	32 entries, fully associative
DCAS	64 total bytes
DCAS Refresh Buffer	96 total bytes
DCAV	4 total bytes

Table 3.3: Energy for L1 DC and DTLB Components

Component	**Energy**
Read L1 DC Tags - All Ways	0.494 pJ
Read L1 DC Data - All Ways	6.358 pJ
Write L1 DC Data - One Way	2.723 pJ
Read L1 DC Data - One Way	1.590 pJ
Read DTLB - Fully Associative	1.675 pJ
Read DTLB - One Way	0.057 pJ
Read DCAS - 1 Entry	0.025 pJ
Write DCAS - 1 Entry	0.030 pJ
Read DCAV - 32 Bits in All 4 Entries	0.072 pJ
Write DCAV - 1 Bit in All 4 Entries	0.036 pJ
Refresh Buffer Read - 1 Entry	0.074 pJ
Refresh Buffer Write - 1 Entry	0.142 pJ

CACTI was used to estimate L1 DC and DTLB energy usage assuming 22-nm CMOS process technology with low standby power (LSTP) cells. Table 3.3 shows the energy required for accessing

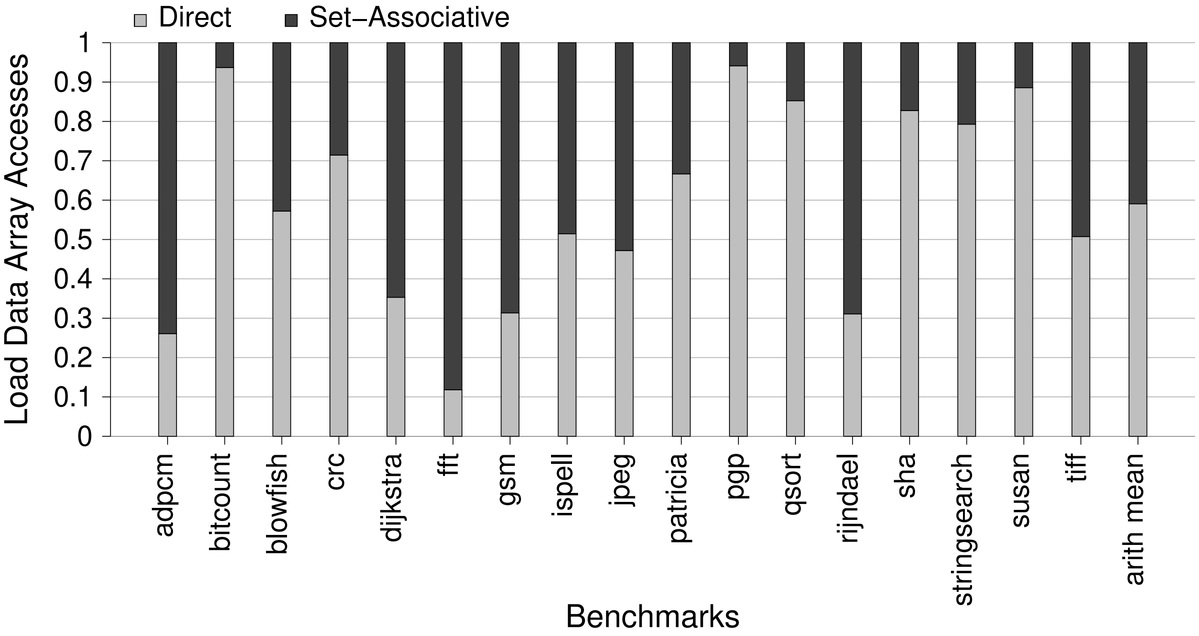


Figure 3.10: L1 DC Data Array Load Accesses

the various components. Leakage energy was gathered assuming a 1 GHZ clock rate.

3.6 Results

Figure 3.10 shows the ratio of L1 DC data array load accesses that are direct (single L1 DC way) or set associative (all L1 DC ways). Over 59% of the load accesses on average are now direct. In the baseline all loads access all L1 DC data arrays and all stores access a single L1 DC data array as the tag check must occur before the L1 DC is updated.

Figure 3.11 shows the ratio of tag checks and DTLB accesses that remain after applying the DCAM technique. On average about 63% of the L1 DC tag checks are eliminated and about 82% of the fully associative DTLB accesses are eliminated. About 18% of the original DTLB accesses are now just accessing a single way of the DTLB, which occurs when the *set index* field is updated and causes an L1 DC tag check, but the *virtual page number* field is unaffected. A single way DTLB access requires much less energy than a fully associative DTLB access, as shown in Table 3.3. On average over 7% of these avoided L1 DC tag checks are due to memoizing the next sequential line.

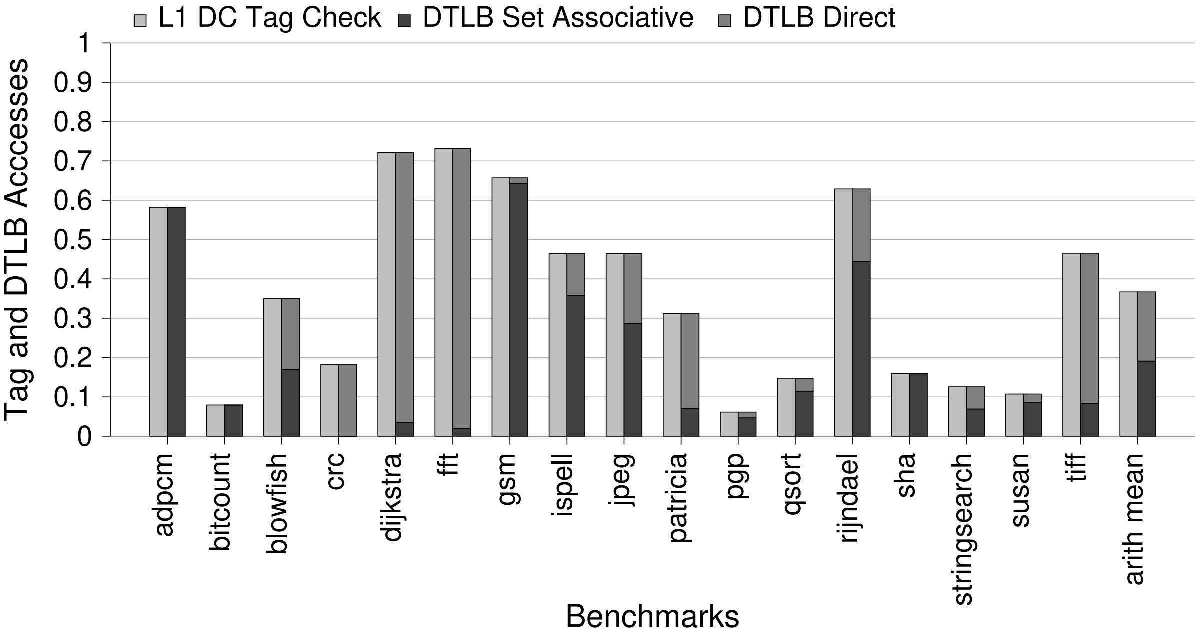


Figure 3.11: Remaining DTLB and Tag Checks

Figure 3.12 shows the breakdown of energy used by the components involved in a data access operation. For each benchmark the left bar shows results for the baseline and the right bar shows results for the DCAM technique. On average about 1.1% of the total energy is due to leakage. For the average baseline data access energy, 57.9% is due to data array reads from load instructions, 12.5% is due to data array writes from store instructions, 6.7% is due to L1 DC tag checks, and 22.7% is due to DTLB accesses. DCAM reduces the energy on average for data array reads to 31.6%, L1 DC tag checks to 2.4%, and DTLB accesses to 4.2%. Note that the energy for data array writes remains the same as writes are direct accesses in both the baseline and DCAM. There is an average overhead of 1.5% for accessing the DCAS and DCAV structures when using the DCAM technique. Overall, the data access energy is reduced to roughly 54% of the baseline on average. The overall energy savings ranges from 71.4% for the *susan* benchmark to 12% for the *fft* benchmark. These energy reductions are significant given that these benefits are obtained on existing binaries with no ISA changes.

Figure 3.13 shows the same breakdown of energy used by the components involved in a data access operation for other various techniques. Using DCAM alone (52.4%) fails to do better than

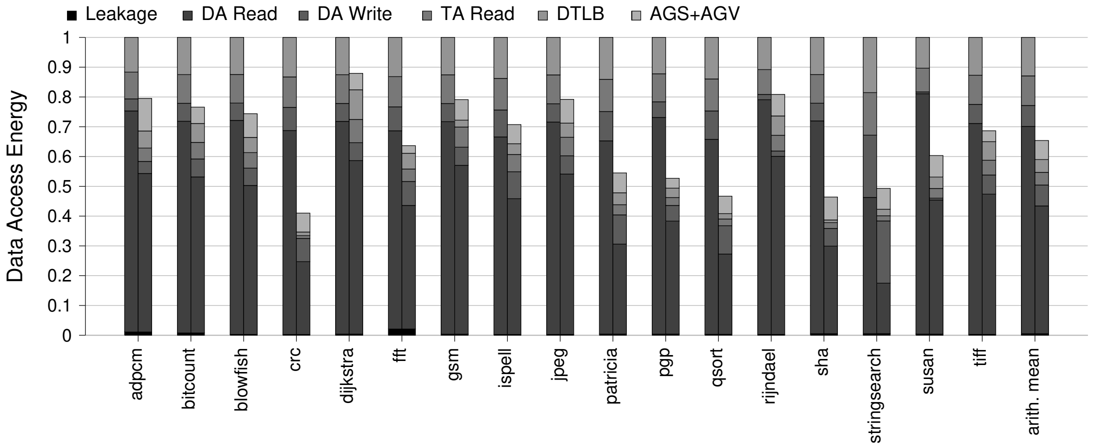


Figure 3.12: DCAM Energy Relative to Baseline

way caching (46.1%). Way prediction [11, 22] is more commonly used than way caching due to other way caching disadvantages. DCAM (a nonspeculative technique) in combination with way prediction, which only predicts the way when it is not memoized by DCAM, achieves the best results (36.3%). This is because way prediction and other speculative techniques, cannot avoid accessing the DTLB and the tag array. Other techniques that can avoid these accesses do so at a significantly higher cost in overhead energy relative to DCAM, such as way caching. All of the other evaluated techniques have some disadvantages that are described in Section 3.7.

3.7 Related Work

Many techniques have been investigated to reduce data access energy. Most of these techniques require trade-offs that may affect how they can be implemented or used. Not all of these techniques conflict with the DCAM approach, as combining some approaches with DCAM could result in lower data access energy than using either approach alone. Taken together, these various characteristics provide a taxonomy of data access efficiency techniques that can be used to compare against the DCAM approach that is shown in Table 3.4.

Unlike the DCAM approach, way-prediction techniques (WP) can have a performance penalty of several percent [11, 22] (OM). These techniques predict which way of the data array is being accessed and this prediction is then verified by performing a L1 DC tag comparison and DTLB access (TD). Newer versions of way-prediction are more accurate, but require a custom SRAM implementation to mitigate the latency of accessing way prediction information before the regular

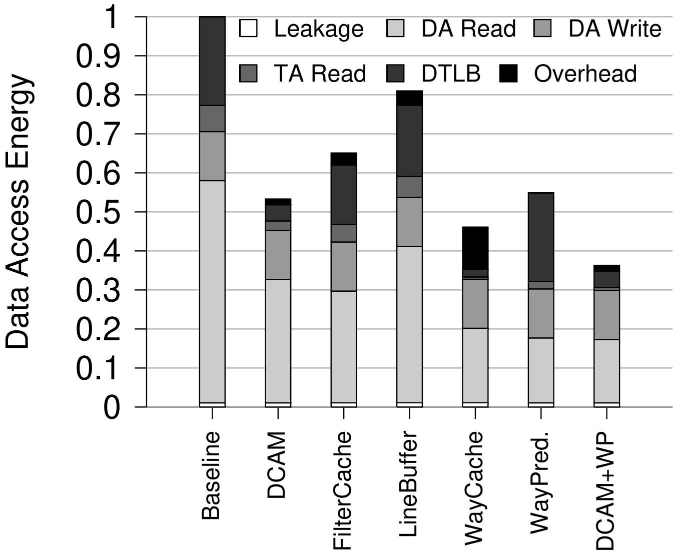


Figure 3.13: Comparison of Energy Techniques

L1 DC tag and data access (CS) using a hash of the virtual address. Nicolaescu et al. propose to save the L1 DC way of the last 16 cache accesses in a table (WC), and each memory access speculatively performs a fully associative tag search on this table (CP, CS). If there is a match, then only the corresponding way is activated [20]. In contrast, the structures used in the DCAM approach to avoid an associative L1 DC data array access are much less expensive to access. Way halting (WH) is another method for reducing the number of tag comparisons [31], where partial tags are stored in a fully associative memory (the halt tag array) with as many ways as there are sets in the cache. In parallel with decoding the word line address the partial tag is searched in the halt tag array. Only for the set where a partial tag match is detected can the word line be enabled by the word line decoder. This halts access to ways that cannot contain the data as determined by the partial tag comparison. Way halting requires a specialized SRAM implementation that might have a negative impact on the maximum operational frequency (CS). WP and WH could be combined with the DCAM approach to reduce energy usage even further (COM).

There have also been some techniques proposed to avoid DTLB accesses. For example, opportunistic virtual caching (OVC) is a technique to allow some blocks in the L1 caches to be cached with virtual addresses by changing the operating system to indicate which pages can use virtual

Table 3.4: Comparison of DCAM Approach to Various L1 DC Access Techniques

Data Access Techniques		Characteristics of Techniques	
WP	Way Prediction	MS	***m***ore ***s***pace required
WC	Way Caching	OM	***o***verhead on ***m***isses
WH	Way Halting	CP	may be on ***c***ritical ***p***ath
TLC	TagLess Cache	CI	***c***ompiler/ISA changes
LB	Line Buffer	CS	***c***ustom ***S***RAM required
FC	Filter Cache	TD	***T***ag/***D***TLB access
TCE	Tag Check Elision	COM	***com***plements DCAM
DAGDA	Decoupled AddrGen & Data Access	HC	***h***igher ***c***omplexity

	MS	OM	CP	HC	CI	CS	TD	COM
WP		X				X	X	X
WC	X		X			X	X	
WH				X		X	X	X
TLC	X		X					X
LB			X			X	X	
FC	X	X					X	
TCE	X		X	X				
DAGDA					X			

caching [5] (OS). In contrast, the DCAM technique can avoid many DTLB accesses by detecting that the physical page has not changed while requiring no OS changes.

L1 DC tag checks for memory references are eliminated when the cache line to be accessed can be identified by the compiler as being known by using direct address registers (DARs) [29]. The compiler annotates a memory reference that sets a DAR identifying the accessed L1 DC line and subsequent memory references that are guaranteed to access the same line reference the same DAR to avoid the tag check (CI). Unlike the DCAM technique, several compiler transformations are required, such as loop unrolling and alignment of variables on cache line boundaries, to make these guarantees, which can result in both code and data size increases. In addition, the DAR approach requires ISA modifications to support it.

A tagless cache (TLC) design has been proposed that uses an extended TLB (ETLB) to avoid tag checks [24]. While the TLC approach can significantly reduce energy usage, the authors assume the ETLB is accessed first to subsequently allow accessing a single L1 DC data array, which could either increase the cycle time or require an additional cycle to service an L1 DC access (CP). The DCAM approach could be used in conjunction with the TLC approach as the ETLB can be avoided when memoization detects that the L1 DC way is already known (COM). Unlike DCAM, the TLC approach does not avoid TLB accesses (TD). Finally, the use of a TLC requires dealing with synonyms, homonyms, and other problems associated with virtually addressed data accesses.

Other small structures have been suggested to reduce L1 DC energy usage. A line buffer (LB) can be used to hold the last line accessed in the L1 DC [26]. The buffer must however be checked before accessing the L1 DC, placing it on the critical path, which can degrade performance (CP). A line buffer also has a high miss rate, which may increase the L1 DC energy usage due to continuously fetching full lines from the L1 DC memory (OM). A small filter cache (FC) accessed before the L1 DC has been proposed to reduce the power dissipation of data accesses [15]. However, filter caches reduce energy usage at the expense of a significant performance penalty due to their high miss rate (OM), which mitigates some of the energy benefits and has likely discouraged its use.

There are some similarities between the Tag Check Elision (TCE) approach and the DCAM approach [34]. Like DCAM, the TCE approach stores an L1 DC way with each integer register. However, there are several significant differences between TCE and DCAM. The TCE approach is likely to memoize more cases with large displacements. However, this feature comes with several disadvantages as compared to the DCAM approach, as depicted in Table 3.4, including that the TCE complexity may increase the critical path that could affect the cycle time (CP). Unlike TCE, DCAM retains the DTLB way to avoid DTLB accesses when a different line is accessed within the same page. TCE stores a bound with every register to memoize L1 DC ways, which in their evaluation was a 29-bit value (MS). In contrast, DCAM requires no immediate value with DCAS entries, which should require much less power to access. TCE requires two comparisons and an addition to verify that the effective address of the memory reference is within the bounds of the cache line as well as an extra addition and a bound read and write each time an integer register is incremented by a value (CP, HC). DCAM's check for a carry out of an addition into the *set index* field and *VPN* fields is much simpler. Finally, TCE's invalidation scheme requires much more space than DCAM's invalidation method (MS).

The Decoupled Address Generation and Data Access (DAGDA) technique exploits memoization to improve data access energy efficiency. However, all loads and stores are required to utilize zero displacements, requiring both compiler and instruction set architecture (ISA) changes [25].

3.8 Conclusions

In this chapter, an approach was described to reduce energy usage by saving L1 DC access information with the register used to access memory. By associating the DTLB access and L1 DC tag check with the base register used in a memory operation the processor is often able to avoid L1 DC tag array accesses and DTLB accesses and access a single L1 DC data array for loads. Furthermore, a technique is shown to retain this information across pointer updates if the updated value falls within the same cache line or page of the source register. These energy saving benefits were able to be obtained on unmodified binaries.

CHAPTER 4

AN ENERGY EFFICIENT DESIGN FOR UTILIZING A LEVEL-ZERO DATA CACHE

Level-zero data caches (L0 DCs), also known as data filter caches (DFCs), have been shown to be effective at reducing data access energy usage. However, this energy reduction comes with a performance penalty when the data being accessed is not in the L0 DC as the initial access to the L1 DC is delayed by at least one cycle. In this chapter a design is described for utilizing an L0 DC that both reduces data access energy usage and provides a performance improvement. In contrast to a traditional L0 DC, the L0 DC design proposed in this chapter allows data to be accessed during the level-one data cache (L1 DC) address generation stage. Performance is improved as L0 DC load hits provide the data earlier than the L1 DC, reducing stalls due to load hazards. Data access energy usage is reduced as the proposed L0 DC design is smaller and requires no DTLB access, making it more efficient to access than an L1 DC. This chapter also provides additional techniques that reduce the power for many of the memory operations still accessing the L1 DC.

4.1 Introduction

A level-zero data cache (L0 DC), also known as a data filter cache (DFC), has been shown to be effective at reducing data access energy [14, 15]. An L0 DC is energy efficient since a large fraction of the memory references can be serviced from the L0 DC that is much smaller than a level-one data cache (L1 DC), resulting in less energy usage for each L0 DC reference as compared to an L1 DC reference. However, a conventional L0 DC has disadvantages that has discouraged its adoption in contemporary processors. First, an L0 DC can cause a performance penalty as it has to be accessed before the L1 DC in order to to reduce energy usage. Upon an L0 DC miss, the L1 DC is accessed a cycle later than it normally would, potentially causing load-delay hazards that would not occur had an L0 DC not been used. This increase in execution time will mitigate some of the energy benefit of using an L0 DC. Second, a single cycle L0 line fill as proposed in many prior studies [7, 9, 10, 14, 15, 27] has been shown to be unrealistic as it can adversely affect

L1 DC area and energy efficiency [3]. These issues must be resolved for an L0 DC to be a practical alternative for a high performance embedded processor.

This chapter proposes a new design that allows the use of an L0 DC that improves both energy efficiency and performance. The key insight for this design is to dynamically detect instructions that update a register whose value is dereferenced by a load or a store and to keep these register values in a small structure that can be accessed a cycle earlier in the pipeline. This feature allows the effective address to be calculated a cycle earlier so that the L0 DC can be accessed a pipeline stage before the L1 DC is conventionally accessed.

Our design for utilizing an L0 DC makes the following contributions. (1) In contrast to a conventional L0 DC that degrades performance, accessing the L0 DC data early provides a small performance improvement by avoiding many load hazard stalls and removing the potential performance penalty typically associated with L0 DCs. (2) Data access energy usage is reduced not only because the L0 DC is smaller than an L1 DC, but also because the proposed L0 DC design does not require a DTLB access. (3) L1 DC access information obtained during the L0 DC access allows the L1 DC to be more efficiently accessed when servicing L0 DC misses, writing through to the L1 DC, and filling words within an L0 DC line from the L1 DC.

4.2 Proposed L0 DC Design

In this section an approach is described for utilizing an L0 DC to both improve performance and reduce energy usage. This section provides a high-level overview of the design.

Figure 4.1 shows a high-level datapath for ALU and load instructions in a six stage instruction pipeline that is utilized in this chapter. Loads from the L0 DC occur during the fourth (L0DC) stage and loads from the L1 DC occur during the fifth (L1DC) stage. Loads from the L0 DC are possible one cycle before the L1 DC is accessed when the effective address (base register plus displacement) can be calculated during the RF stage. To accomplish this, the base register of the load is obtained during the ID stage from the *basereg* structure, a small subset of the register file used by loads and stores. Since the displacement value is available immediately from the instruction bits, the effective address can be computed during the RF stage. Alternatively, if the displacement of the load is zero, there is no need to calculate the effective address. In this case, the effective address is simply the base register value which can be obtained from the register file. Using this

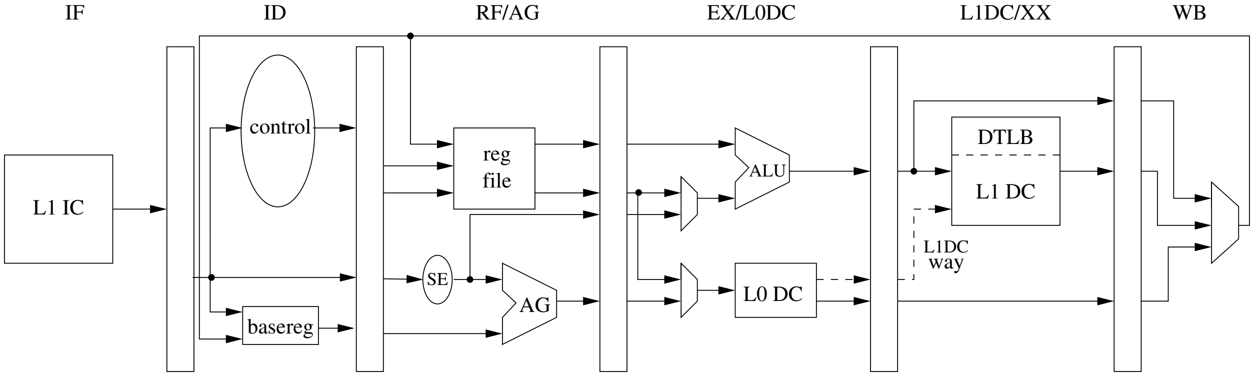


Figure 4.1: Datapath for ALU and Load Instructions

early effective address computation strategy, the data can be obtained from the L0 DC during the address generation (AG) stage, one cycle before the the L1 DC is accessed during the MEM stage.

Having the L0 DC and L1 DC always accessed in a given pipeline stage helps to avoid structural hazards and simplifies the pipeline. Each L0 DC line contains the corresponding L1 DC way in which it resides, which is used to make L1 DC accesses more efficient when they are needed to be performed. The DTLB is only accessed when accessing the L1 DC and the L1 DC way is unknown. Forwarding paths and the internal pipelining for the L0 DC and L1 DC to handle stores are not shown to simplify the figure.

The L0 DC in this design will not be accessed on every load instruction since sometimes the base register value is not available from the *basereg* structure and the displacement is not zero. One strategy would be to access the L0 DC after the effective address is calculated during the AG stage. However, an L0 DC miss could cause a performance penalty as the data would be retrieved from higher levels of cache a cycle later than it normally would. Instead, it would be desirable to load the data from the L1 DC instead of waiting until the address generation is complete to access the L0 DC, effectively removing the perfomance penalty typically associated with L0 DCs. Since the data can be retrieved from either the L0 DC or the L1 DC the processor needs to ensure that data in the L1 DC has the same values as any data that are resident in the L0 DC. An inclusive cache policy and an L0 DC write-through policy are used 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 as the processor does not have to deal with writing back dirty L0 DC lines over multiple cycles, which would allocate the L0 DC read port, making the L0 DC inaccessible during this period. Instead, evicted L0 DC lines simply need to be invalidated.

Figure 4.2 shows the information that will be contained in each L0 DC line. The page protection bits are copied from the DTLB when the L0 DC line is allocated. This is necessary to ensure that data is accessed properly as the DTLB is avoided during L0 DC hits, as described is Section 4.3.5. An L0 DC line also identifies the L1 DC way where the L0 DC line resides. Each data byte within the line has a filled bit to indicate if that byte within the line is resident, described in Section 4.3.4.

v	tag	PP	L1 DC way	f	data	...	f	data

v = valid bit PP = page protection bits f = filled bit

Figure 4.2: L0 DC Line Contents

The remainder of this section is used to describe the design in more detail.

4.3 Utilizing an L0 DC to Improve Performance

The following subsections describe how this design makes base register values available earlier in the pipeline, is integrated into a pipeline, fills data words in an L0 DC line, and is virtually addressed.

4.3.1 Making Base Register Values Available Earlier in the Pipeline

A simple approach to ensure that base register values are available in the *basereg* structure is to have all integer instructions that update a register to write their register value to the *basereg* structure. There are two problems with this approach: 1) the energy-saving benefits of using an L0 DC are mitigated due to unnecessary *basereg* writes and 2) additional pressure is placed on the basereg structure, meaning it will be harder to retain *basereg* values long enough so that they can be used by loads and stores because they will be evicted by unnecessary *basereg* writes.

To solve these problems, an approach is described to dynamically detect instructions that update an integer register whose value will be dereferenced by a load or a store. Such registers are referred to as base registers since they contains the base value of the effective address for a load or a store. Such instructions are referred to as base address generation (BAG) instructions as they update the base register of a load or store. Table 4.1 shows the different MIPS instructions that are likely to be a BAG instruction and account for over 99.9% of the BAG instructions. An instruction with one of these opcodes is referred to as a potential BAG (PBAG) instruction.

Table 4.1: Last Instruction to Compute a Data Address

Size	Type	Operation	MIPS Inst Effect	Source Operands
Scalar	Local	(1) int immed add	rd = rs + immed	stack pointer and offset
	Global	(2) bitwise immed OR	rd = rs | immed	high | low global address
	Pointer	(3) int load	rt = M[rs]	pointer variable address
Composite	Array Elem	(4) int reg add	rd = rs + rt	array address + elem offset
	Struct Field	(1) int immed add	rd = rs + immed	struct address + field offset
	Ptr Arith	(5) int reg sub	rd = rs - rt	pointer - var offset

When a PBAG instruction is executed, a small *basereg* structure as depicted in Figure 4.3 is updated. Each *basereg* element contains the value of an integer register that was dereferenced in a load or store instruction. The structure shown in the figure contains at most four base register values. The *baseregindex* structure is indexed by the base register number of a load or store instruction and is used to select the base register value from a multiplexor. The BV (Base register Valid) bit indicates if the integer architectural register currently points to a *basereg* element. The DR (DeReferenced) bit indicates if the base register has been dereferenced by a load or a store with a nonzero displacement. The processor will update the *basereg* element during the WB stage of a BAG instruction. A separate *baseregnum* structure contains for each *basereg* element the integer register number associated with that value. The LRU element of the *basereg* structure is replaced if there is not a valid value already associated with the base register number and the *baseregnum* is used to clear the BV bit of the replaced *basereg* element.

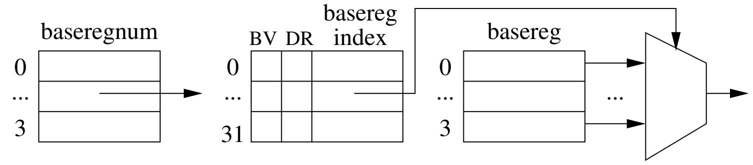


Figure 4.3: Base Register Structure

4.3.2 Using Base Register Values in Loads and Stores

Similar to how to the number of unnecessary *basereg* writes have been reduced, a technique is proposed to reduce the number of unnecessary *basereg* reads. Ideally, a *basereg* element is accessed only during load and store instructions that use a non-zero displacement value so there are no unnecessary basereg reads, which mitigate the energy savings of the approach. However, since the instruction type isn't known until the end of the ID stage and possibly the register number depending on the instruction set architecture, this section describes an approach which approximates this behavior.

A bit is associated with each instruction in the L1 IC, called the *LS* (Load/Store) bit, to classify instructions that are a load or store with a nonzero displacement (when any bits in the immediate field are not zero) as shown in Table 4.2. The reason the processor checks for a nonzero displacement is that a load or store with a zero displacement does not need to access the *basereg* structure as described in Section 4.3.3. This *LS* bit associated with an instruction is read each time during the instruction fetch (IF) pipeline stage.

Table 4.2: Value Associated with the *LS* Bit

Value	Meaning
0	otherwise
1	load or store with a nonzero displacement

Instructions with a PBAG opcode as shown in Table 4.1 comprise a significant fraction of the instructions executed. It is desirable to only update the *basereg* when an actual address is being generated as unnecessary *basereg* updates expend additional energy and may replace useful *basereg* values. The following technique is used to avoid unnecessary updates to the *basereg* structure. The *DR* (dereferenced) bit in Figure 4.3 is set when a register is dereferenced by a load or a store with a nonzero displacement where the PBAG instruction was executed in time for the *basereg* structure to be dereferenced. When a BAG instruction sets a register that has its *DR* bit set, then the BAG instruction will update the *basereg* structure and set the BV bit in Figure 4.3. This approach works well as a register that is used to access memory in loops is often not set by other non-PBAG instructions. Note that non-PBAG instructions will clear both the BV and *DR* bits associated with the register number being updated.

Figure 4.4 shows a comparison of several different approaches that can be used to determine which instructions will update the *basereg* structure. One approach that can be used is to allow all instructions that update to an integer register to write their result to the *basereg* structure, referred to as the "Side Effect" approach as all instructions with an integer side effect write to it. However,

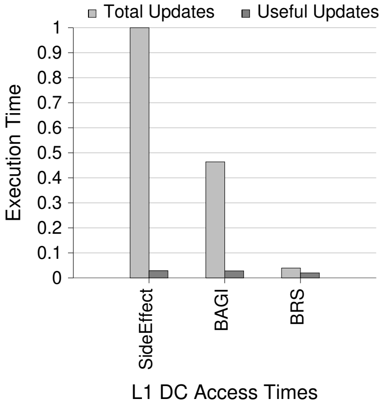


Figure 4.4: Comparison of BRS Update Approaches

as shown in the figure, only about 2.8% of these instructions will have their *basereg* element used by a load or store instruction. This is because many instructions 1) don't update a register used by a load or store instruction, 2) update a register that is used by a load or a store whose displacement value is zero and thus doesn't need to access the *basereg* structure, or 3) updates a register that is used by a load or a store but whose value will not be ready in time to be read from the *basereg* structure. An additional approach used is to only update *basereg* structure for BAG instructions (BAGI), which constitute the vast majority of instructions that are the last to update a register before the register is used in a load or store. This filters out approximately 61% of unnecessary basereg writes. However, this still updates the *basereg* structure 15 times unnecessarily for every one update eventually used by a load or store. Utilizing the DR bit to detect registers that are used by loads and stores only updates the *basereg* structure for about 4% of all instructions, of which, about half of them are used by subsequent loads and stores. This approach drastically reduces the number of updates to the *basereg* structure, enabling the *basereg* structure to use fewer elements as there is less pressure due to unnecessary updates evicting useful entries.

4.3.3 Integrating L0 DC Accesses into the Instruction Pipeline

Table 4.3 describes the different integer instruction pipeline stages used in this study. Note many of the names for these stages occur at the same time in the pipeline, but distinct names are provided to facilitate understanding what pipeline actions are performed by different instructions. This pipeline separates the ID (instruction decode) and RF (register fetch) stages to reduce energy usage caused by unnecessary register fetches. The *basereg* structure is accessed during the ID stage of an instruction marked as a load or store with a nonzero displacement (see Table 4.2) to obtain the base register value. The AG (address generation) stage performs an addition with this base register value and the displacement to calculate the effective memory address.

Table 4.3: Instruction Pipeline Stages

Stage	Meaning	Stage	Meaning
IF	instruction fetch	L0TC	L0 DC tag check
ID	instruction decode	L1TC	L1 DC tag check
RF	register fetch	WB	write back
AG	address generation	TC	L0 DC/L1 DC tag check
EX	execute	DCW	L0/L1 DC write, no tag check
L0DC	L0 DC access	L1W	L1 DC write, no tag check
L1DC	L1 DC access	XX	stage not used

Table 4.4 shows how different instructions proceed through a six stage pipeline utilizing the pipeline stages shown in Table 4.3. Note this design can easily be adapted to work with additional pipeline stages supporting a multicycle L1 DC load access.

Table 4.4: Stages Used by Instructions

Instruction	Pipeline Stages					
	1	2	3	4	5	6
(a) ALU inst	IF	ID	RF	EX	XX	WB
(b) L1 load hit	IF	ID	RF	EX	L1DC	WB
(c) L0 load hit	IF	ID	AG	L0DC	XX	WB
(d) L0 load hit no disp	IF	ID	RF	L0DC	XX	WB
(e) L0 load miss	IF	ID	AG	L0DC	L1DC	WB
(f) L0 load miss no disp	IF	ID	RF	L0DC	L1DC	WB
(g) L1 store hit	IF	ID	RF	EX	TC	DCW
(h) L0 store hit	IF	ID	AG	L0TC	XX	DCW
(i) L0 store hit no disp	IF	ID	RF	L0TC	XX	DCW
(j) L0 store miss	IF	ID	AG	L0TC	L1TC	L1W
(k) L0 store miss no disp	IF	ID	RF	L0TC	L1TC	L1W

An *ALU inst* (case (a) in Table 4.4) proceeds through the pipeline and does not perform any action (XX) in the 5th pipeline stage. A BAG ALU instruction also writes to the *basereg* structure in the sixth stage.

An L1 load hit or L1 store hit (cases (b) and (g) in Table 4.4) operation means that the base register value was not available in the *basereg* structure and the load or store will access the L1 DC. An L1 store hit will require that both the L0 DC and the L1 DC to be updated when the data line is resident in the L0 DC since the L0 DC uses an inclusive cache policy. The L0 DC tag arrays are replicated to allow an L0 DC tag check in either the L0DC or L1DC stages without causing a structural hazard.

All operations beginning with L0 means the L0 DC is accessed. For cases (c) and (d) in Table 4.4, the load hits in the L0 DC and no L1 DC access is performed. In the cases of (d), (f), (i), and (k) in Table 4.4, the displacement of the load and store is zero, meaning no address calculation is necessary and the processor does not access the *basereg* structure. For cases (c), (e), (h), and (j) in Table 4.4, the load/store has a non-zero displacement and the processor obtains the base register value from the *basereg* structure in the ID stage in order to complete the address calculation in the AG stage, where a separate adder is utilized to avoid a structural hazard. Note an AG stage does not access the register file when the base register value had already been obtained during the ID stage from the *basereg* structure. For cases (e), (f), (j), and (k) in Table 4.4, the L0 DC is accessed but the word being accessed is not resident and the processor accesses the L1 DC in the following cycle.

The *basereg* structure is only accessed when an instruction has been marked in the *LS* bit vector as a load or store instruction with a nonzero displacement. The base register value is not available in the *basereg* structure when the base register value for a particular register in the *basereg* structure was replaced or the base register value was not calculated in time to perform an address calculation in the AG stage. The base register value can be forwarded to the AG stage of a load or store after the EX stage of a BAG ALU instruction, the L0DC stage of BAG load that hits in the L0 DC, or the L1 DC stage of a BAG load that obtains its value from the L1 DC. However, there must be at least one instruction between the ALU BAG instruction (or a BAG load that hits in the L0 DC) and the load or store that uses the BAG instruction destination register so that the value can either be read from the *basereg* structure or forwarded to the AG stage of the load or store instruction. A BAG load that obtains its value from the L1 DC must be separated from the load or store that uses the value by at least two instructions for forwarding to occur. Note that if forwarding cannot occur, then the L1 DC is accessed to avoid pipeline delays.

4.3.4 Filling L0 DC Lines

While the L0 DC line size can be smaller than an L1 DC line, it is still advantageous to utilize a multiword L0 DC line size to exploit spatial locality in data references. Many prior filter cache studies have assumed that an L0 line is the same size as an L1 line and can be filled in a single cycle to reduce the L0 miss penalty [7, 9, 10, 14, 15, 27]. Such an assumption is unrealistic as reading an entire line from an L1 DC in a single cycle requires a larger bitwidth, which could increase the area of the L1 DC and negatively affect both L1 DC access time and access energy. In addition, some applications referenced only a single word from an L0 DC line before the L0 DC line was evicted.

This section describes an L0 DC fill strategy that is realistic and does not require a performance delay due to a miss penalty. An *f* (filled) bit is associated with each data byte in an L0 DC line, as shown in Figure 4.2. Thus, there are two types of L0 DC misses. An *L0 DC line miss* means the entire L0 DC line is not resident and an *L0 DC reference miss* means that the data reference being accessed within an L0 DC line is not resident. An *f* bit is associated with each byte in an L0 DC line to allow stores of bytes or halfwords into an L0 DC line without having to load a word from the L1 DC on a store *L0 DC reference miss*. When a load *L0 DC line miss* occurs, the L0 DC line is allocated and the single demanded word is loaded into the L0 DC line. This approach is able to achieve a hit rate comparable to a line-filled L0 DC while not incurring the overhead associated with filling an entire L0 DC line in a single cycle or the complexity of filling an L0 DC line over multiple cycles.

Using this approach, it's possible that the L0 DC line is resident but the desired word of data is not resident. By storing additional information with each L0 DC line, L1 DC accesses can be made more efficient after accessing the L0 DC in the case of line hits but word misses. Figure 4.2 shows that the L1 DC way is stored with each L0 DC line. In this case the L1 DC way is used to access the L1 DC without an L1 DC tag check or DTLB access.

4.3.5 Utilizing a Virtually Tagged L0 DC

Our L0 DC is accessed using virtual addresses, which means that virtual tags are used to check if there is an L0 DC hit. The advantage of this approach is that there is no need to access the DTLB in parallel with the L0 DC access, which also avoids a structural hazard for accessing the DTLB during the L0DC stage for some memory instructions and the L1DC for other memory instructions. However, using a virtual cache causes a number of complications, which is simpler to handle in a smaller L0 DC.

(1) To handle the synonym problem, where different virtual addresses can map to the same physical address, the L0 DC lines corresponding to an evicted L1 DC line are also evicted. In Figure 4.2 the L1 DC way associated with each L0 DC line is included. When an L0 DC line is

replaced after a miss, the L1 DC way and index values are compared to the same values in other L0 DC lines within the same L0 DC set. If there is a match with another L0 DC line, then that line is invalidated. Note the portion of the L1 DC index value that can differ can be obtained from the least significant bits of the L0 DC tag as a virtually-indexed physically-tagged (VIPT) L1 DC is assumed.

(2) The homonym problem is that a single virtual address may map to different physical addresses when multiple virtual address spaces are used due to context switches. Our solution is to invalidate all the L0 DC lines on context switches. Few additional L0 DC misses will result from this invalidation since the L0 DC is much smaller than the L1 DC and it is unlikely L0 DC lines associated with one process will remain after switching back to the same process. Note the *basereg* structure automatically gets updated as a context switch restores all the register values associated with another process.

(3) The page protection problem is that pages must be safely accessed. The DTLB contains page protection (PP) bits that will be copied into each L0 DC line as shown in Figure 4.2. The overhead of storing and accessing these PP bits is small since there are only a few PP bits for each DTLB page and there are few L0 DC lines.

(4) The multiprocessor cache coherency problem occurs when a cache line needs to be invalidated due to a cache coherency invalidation request. Our L0 DC is strictly inclusive with respect to the L1 DC. When an L1 DC line is evicted (due to a line replacement or coherency invalidation request), any L0 DC line that has a matching L1 DC way and index is also invalidated. Note all the L1 DC ways fields can be checked in parallel.

4.4 Evaluation Environment

This section describes the evaluation environment. The seventeen benchmarks from the MiBench benchmark suite [8], which is a representative set of embedded applications, are used to evaluate the proposed design. All benchmarks are compiled using gcc with the *-03* option.

The ADL simulator [21] was used to simulate both a conventional MIPS processor as the baseline and the modified processor as described in this chapter. The ADL simulator performs a more realistic simulation than many commonly used simulators (in ADL data values are actually loaded from the caches, values are actually forwarded through the pipeline, branch target addresses from the branch target buffer are actually used, etc.). Both configurations are single-issue, in-order processors with six-stage pipelines. Branch instructions resolve in the EX stage, so there is a 3-cycle misprediction penalty. The simulator used a gshare branch predictor with a branch target buffer. Table 4.5 shows other details regarding the processor configuration that is utilized in the simulations, where the L0 DC, basereg, baseregindex, and LS vector structures are only used in the modified processor.

Table 4.5: Processor Configuration

page size	8KB
L1 DC	32KB, 64B line size, 4-way associative, 1 cycle hit, 10 cycle miss penalty
DTLB	32 entries, fully associative
L0 DC	512B, 16B line size, 4-way associative, 1 cycle hit, 1 cycle word miss penalty
basereg	4 entries, 16 total bytes
baseregindex	32 entries, 16 total bytes
LS vector	256 entries, 32 total bytes

The ADL simulator [21] was used in combination with CACTI [18] for energy evaluation to model both a conventional MIPS processor as the baseline and the modified processor as described in this chapter. CACTI was used to model the L1 DC, L0 DC, base register structure, and LS vector assuming a 32-nm CMOS process technology with low standby power (LSTP) cells and power gating. A way-predicted access is modeled as a direct-mapped L1 DC with one-quarter of the total L1 size. On a misprediction, the energy of a way-associative access is added. Table 4.6 shows the energy required for accessing the various components related to memory accesses. CACTI does not provide energy values for very small caches. Thus, the energy for accessing the smaller L0 DCs is estimated 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. Leakage energy was gathered assuming a 1 GHZ clock rate.

Table 4.6: Energy for L1 DC and DTLB Components

Component	Energy
L1 DC Tags - 3 Ways (WP miss)	0.473 pJ
L1 DC Data - 3 Ways (WP miss)	8.266 pJ
L1 DC Tag - 1 Way (WP hit)	0.177 pJ
L1 DC Data - 1 Way (WP hit)	1.930 pJ
DTLB - Fully Associative	1.030 pJ
L0 DC (512B) Tag - 1 Way	0.025 pJ
L0 DC (512B) Data - 1 Way	0.178 pJ
LS Bit Vector - 1 bit	0.005 pJ
Basereg+Baseregindex - 1 entry	0.071 pJ
Register File - 1 Entry	0.366 pJ

4.5 Results

Initially, an *LS* bit was associated with each L1 IC instruction. However, the energy overhead for accessing such a bit from the L1 IC on each instruction outweighed the benefit of only accessing the *basereg* structure for each load and store. A smaller bit vector was used to reduce the energy usage as opposed to associating a bit with every instruction in the L1 IC. Figure 4.5 shows how bits within an address are used to index into the mth bit in the *LS* vector. The least significant bits of the address (with the exception of the two least significant bits that are always zero due to instructions being aligned on a four-byte boundary) are used to index into the *LS* bit vector. Section 4.3.3 shows how instructions marked as 1 (load or store with a nonzero displacement) utilize the *basereg* structure.

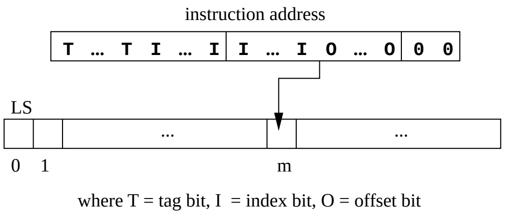

instruction address
T … T I … I I … I O … O 0 0
LS
...
...
0
1
m
where T = tag bit, I = index bit, O = offset bit

Figure 4.5: Accessing the LS Bit Vector

Because multiple instructions can map to the same bit in the *LS* vector, a misclassfication can occasionally occur, which is detected after decoding the instruction. In such a case, the bit in the *LS* vector will be reversed. A load/store with a nonzero displacement that is classified as 0 (see

Table 4.2) will not read from the *basereg* structure and will not access the L0 DC. Forwarding can occur to the AG stage even when the bit in the *LS* vector is appropriately classified since the *basereg* element could have the wrong value when the BV bit was valid as the *basereg* structure is updated during the WB (write back) stage. An instruction that is not a load/store with a nonzero displacement that is classified as 1 (see Table 4.2) will simply read the *basereg* structure. Note that an instruction that successfully reads a value from the *basereg* structure will not redundantly read the same register from the register file.

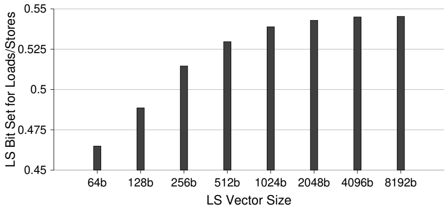


Figure 4.6: Service Rate for Varying Size of LSVector

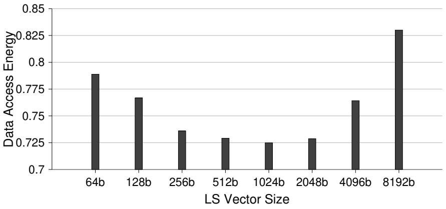


Figure 4.7: Data Access Energy for Varying Size of LSVector

Different *LS vector* sizes are evaluated with a 512B L0 DC and 32 *basereg* elements. The best data access energy was obtained when the *LS vector* size was 1024 bits.

Figure 4.8 shows the L0 DC service rate for a 512B L0 DC with a varying number of *basereg* elements with an *LS vector* size of 1024 bits. About 30.6% of the values were provided by the L0 DC with no *basereg* elements, which shows that many of the memory references had a zero displacement. The difference in the L0 DC load service rate between 2 and 4 *basereg* elements was only about 1.4%. Figure 4.9 shows the data access energy with a varying number of *basereg* elements with an *LS vector* size of 1024 bits. This data access energy includes the L0 DC, L1 DC, and DTLB and is relative to not using an L0 DC. The lowest data access energy was obtained when the number of *basereg* elements was two. This is due to not many registers being live simultaneously that are used to hold addresses to access memory when there is a nonzero displacement. Thus, two *basereg* elements are used in the remaining results that are presented in this chapter.

Figure 4.10 shows the breakdown of memory accesses. The left and right bars associated with each benchmark shows the types of loads and stores that occurred, respectively. An *L0 Hit* indicates that the data was present in the L0 DC. In contrast, an *L0 Line Hit* indicates that the line was resident, but that the data was not resident within the line. An *L0 Hit* for loads results in no access

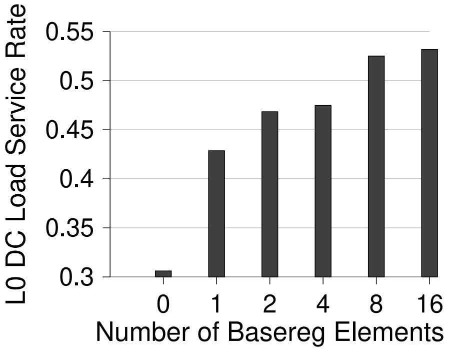


Figure 4.8: Service Rate for Varying Number of Basereg Elements

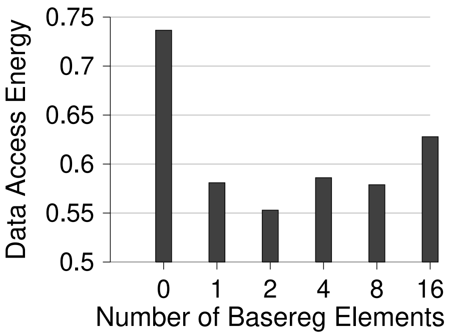


Figure 4.9: Data Access Energy for Varying Number of Basereg Elements

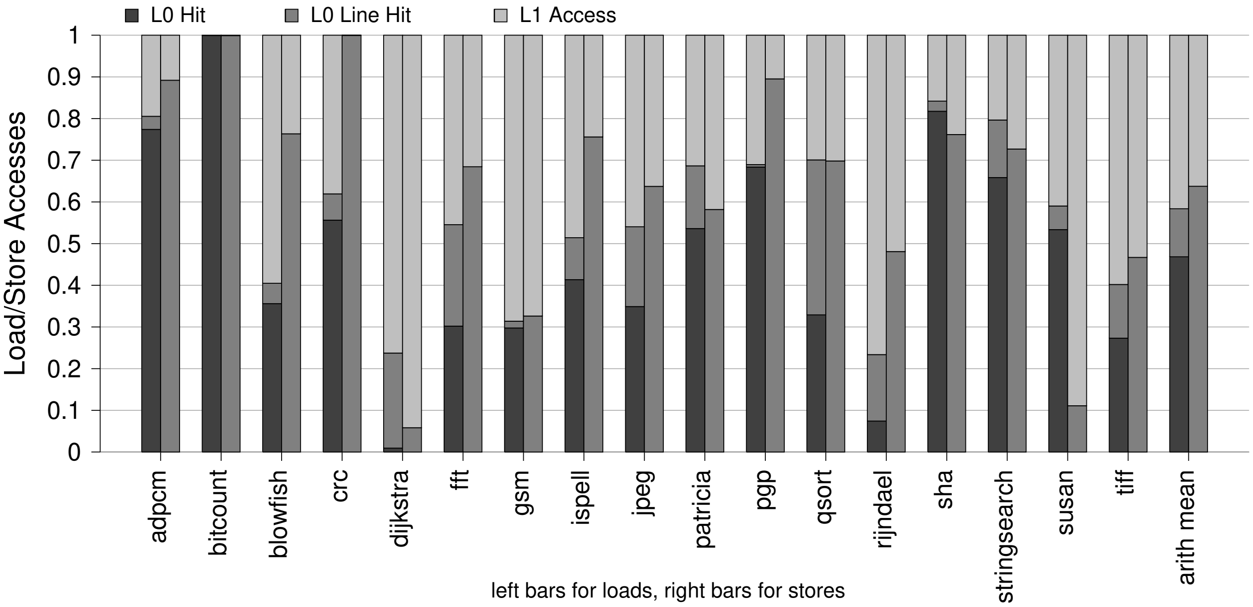
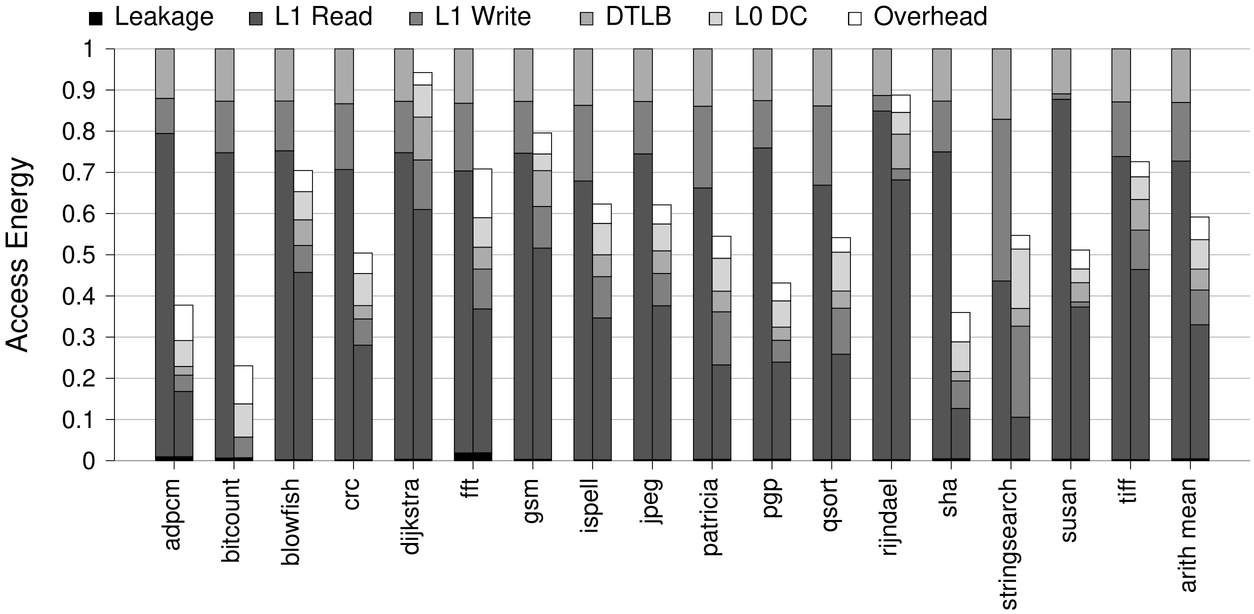


Figure 4.10: Load and Store Access Taxonomy

to the L1 DC. An *L0 Line Hit* for loads indicates that a word was loaded from the L1 DC into the L0 DC and that no DTLB access or L1 DC tag check was performed as the L1 DC way is known since it is stored in the L0 DC line. An *L0 Line Hit* for stores results in a write to the L1 DC as a write-through policy to the L1 DC is used. An *L0 Line Hit* for stores is similar to a *L0 Line Hit* for loads in that the data is written to the L1 DC without a DTLB access or an L1 DC tag check. Note there is no *L0 Hit* given for stores as it's necessary to write-through to the L1 DC so the L1 DC remains inclusive of the L0 DC. An *L1 Access* indicates that either the address could not



left bars indicate no L0 DC, right bars indicate using an L0 DC

Figure 4.11: Data Access Component Energy

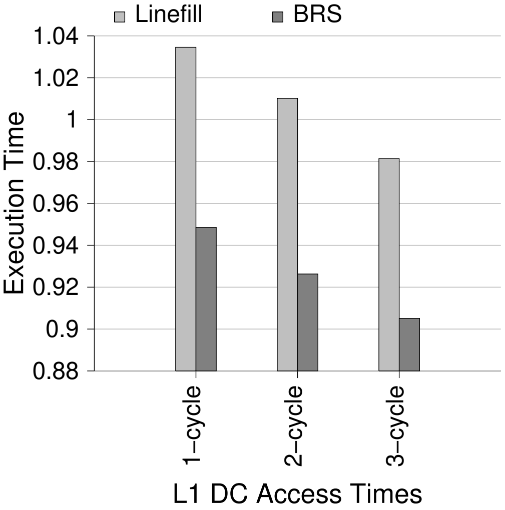


Figure 4.12: Varying L1 DC Hit Access Times

be calculated early as the base register value was not available in the *basereg* structure or pipeline or that the line was not resident in the L0 DC.

The classification of loads and stores within each benchmark shows the type of locality present within the benchmark. The fraction of load *L0 Hits* indicates temporal locality or spatial locality within a word (byte or halfword references) as an entire word is loaded into the L0 DC line on a load miss. Note the *bitcount* benchmark has very high temporal locality and the *dijkstra* benchmark has very low temporal locality. The fraction of load *L0 Line Hits* indicates spatial locality across words within an L0 DC line. The fraction of *L0 Line Hits* would increase if the L0 DC line size was larger.

Figure 4.11 shows the component data access energy of loads and stores. The left bar for each benchmark shows the baseline results without an L0 DC and the right bar shows the results with an L0 DC. The *leakage* energy for these structures is so small that it cannot be seen. The *L1 Read* and *L1 Write* indicates the energy required for accessing the L1 DC during loads and stores, respectively. The *Overhead* represents the energy required for accessing the *LS* bit vector to recognize loads and stores with nonzero displacements, and the accesses to the *basereg* and *baseregindex* structures by BAG instructions.

It is interesting to see the effect on the energy of the different components when utilizing an L0 DC. The *DTLB* energy usage had a significant reduction for two reasons. First, an L0 DC load hit does not access the DTLB since the L0 DC is virtually addressed. Second, many of the L1 DC accesses (L1 DC loads, L1 DC stores, and L0 DC next sequential word line fills from the L1 DC) do not need to perform a DTLB access as the L1 DC way is known after the L0 DC is accessed. The *L1 Read* component representing the L1 DC access energy for load instructions was significantly reduced due to a significant fraction of L0 DC hits that do not access the L1 DC and L0 DC reference misses and L1 DC sequential word line fills that do not perform an L1 DC tag check. The L1 Write component representing L1 DC access energy for store instructions was reduced due to avoiding L1 DC tag checkson L0 line hits.

Figure 4.12 shows the effect on performance for using an L0 DC as the number of cycles required to access the L1 DC increases. The ratios shown for each bar compares a traditional, line-filled L0 DC as well as the proposed approach relative to a processor without an L0 DC. In the first set of bars, it can be seen that a line-filled L0 DC incurs a performance penalty of 3.5% due to load-delay hazard stalls caused by L0 DC misses compared to a processor with no L0 DC. Our approach, on the other hand, improves performance by 5.1% by reducing load-delay hazard stalls by retrieving data a cycle earlier than usual for L0 DC hits. As the access time of the L1 DC is increased to 2 or 3 cycles, both the linefill approach and the proposed approach perform better relative to the baseline. The linefill approach performs better as L0 DC hits can retrieve data 1 or 2 cycles earlier than usual, reducing the number of load-delay hazard stalls on L0 DC hits. While L0 DC misses can still cause load-delay hazard stalls, this is largely offset by this reduction. Our approach continues to improve performance as retrieving data during the EX stage can eliminate 1-, 2-, and 3-cycle load-delay hazards. Our approach reduces the number of cycles executed relative to a conventional, line-filled L0 DC by 7.5% to 8.6% for each L1 DC access time. This result taken together with the fact that a single-cycle L0 DC linefill strategy is not feasible as it increases the time and energy to access the L1 DC means the approach outlined in this chaper outperforms a conventional L0 DC and provides a template for implementing L0 DCs in modern processors.

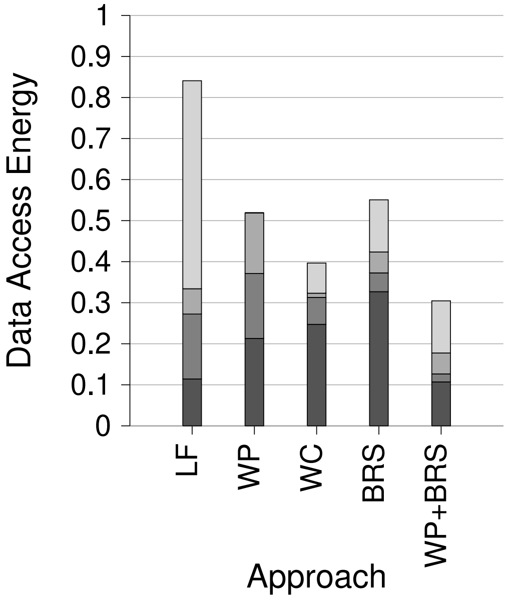


Figure 4.13: Comparison of Methods that Reduce
L1 DC Access Energy

Figure 4.13 compares several different approaches for reducing memory access energy. A conventional, line-filled L0 DC (LF) fails to provide significant energy savings (16%) even though it provides a better L0 DC service rate than the proposed design. This is because the energy required to fetch the data from the L1 DC and to fill the entire L0 DC line in a single cycle mitigates much of the potenital energy savings.

Way prediction (WP) techniques [11, 22] are now commonly used to predict which way of the L1 DC data array is being accessed and this prediction is verified by performing a DTLB access and an L1 DC tag comparison. Way prediciton can both reduce energy usage (a single L1 DC tag array and a single L1 DC data array are accessed) and improve L1 DC load hit time (the requested data from one L1 DC data array can be sent to the CPU without waiting for an L1 DC tag check to be performed). Way prediction provides significant energy savings (48.1%) as it's able to avoid an n-way set-associative L1 DC access. However, it still incurs the overhead of accessing the L1 DC tag array and DTLB and performing a set-associative L1 DC access on mispredictions.

Way caching (WC) stores the tag and way of the 16 most recently accessing L1 DC lines in a 16-way, fully associative structure. Way caching is not only able to avoid set-associative L1 DC accesses but also L1 DC tag array and DTLB accesses. However, way caching incurs a significant amount of overhead because it must retrieve the desired way from this 16-way, fully associative structure. In addition, way cache accesses lie on the critical path as this structure is accessed between the time the effective address is calculated and the L1 DC is accessed.

Using the proposed approach (BRS) alone fails to beat way caching (45% vs 60.4%) due to way caching's high hit rate. By requiring the L0 DC to be accessed only for loads and stores that are able to retrieve their base register value from the *basereg* structure or for loads and stores with displacements of zero so the effective address can be calculated a cycle earlier, the L0 DC service rate is lowered. However, this approach is complementary to way prediction. In the case that a load or store is either unable to calculate its address early and thus cannot access the L0 DC in time or because it misses in the L0 DC, the L1 DC is accessed using way prediction techniques. Using this approach in conjunction with way prediction (WP+BRS) is able to reduce energy usage of the L1 DC by 69.3%. Note that way caching and way prediction are not complementary as both structures are accessed between the calculation of the effective address. Further, way caching has a very high hit rate (96%), and thus adding way prediction will not reduce L1 DC read energy further. In addition, there is no method to avoid *both* way caching's high energy overhead for accessing the way cache and way prediction's overhead of accessing the DTLB and L1 DC tag array to verify predictions.

4.6 Related Work

There have been numerous techniques that have been explored to reduce data access energy within a processor. This section only considers techniques that do not require any changes to an executable or the operating system. Most of these techniques include various compromises that affect the benefits they can achieve and/or the feasibility of their implementation. Some of these techniques can be used in combination with the proposed L0 DC design. Table 4.7 provides acronyms for various data access techniques. Table 4.8 provides acronyms for the characteristics of these data access techniques. Table 4.9 provides an overview of the characteristics of these various techniques.

Table 4.7: Data Access Technique Acronyms

LB	Line Buffer
FC	Filter Cache
PDFC	Practical DFC
ZCL	Zero Cycle Loads
WC	Way Caching
WH	Way Halting
TLC	TagLess Cache
TCE	Tag Check Elision

Table 4.8: Data Access Characteristic Acronyms

MS	More Space
MD	Miss Delays
CP	Critical Path
HC	Higher Complexity
CS	Custom Sram
TD	Tag/DTLB access
ME	More Energy
FA	Fast Access
CM	CoMplementary

Table 4.9: Characteristics of Various Data Access Techniques

	MS	MD	CP	HC	CS	TD	ME	FA	CM
LB		X	X			X			
FC	X	X				X			
ZCL					X		X	X	
PDFC	X		X		X			X	
WC	X		X		X	X			
WH				X	X	X			X
TLC	X		X						X
TCE	X		X	X					

Other small structures have been suggested to reduce L1 DC energy usage. A line buffer (LB) can be used to hold the last line accessed in the L1 DC [26]. The buffer must however be checked before accessing the L1 DC, placing it on the critical path, which can degrade performance (CP). A line buffer also has a high miss rate, which may increase the L1 DC energy usage due to continuously fetching full lines from the L1 DC memory (MD).

The original proposed filter cache (FC) accessed before the L1 DC has been proposed to reduce the power dissipation of data accesses [15]. However, FCs reduce energy usage comes at the expense of a significant performance penalty due to their high miss rate (MD), which mitigates some of their energy benefits and has likely discouraged its use in industry. Small alterations to the FC design have been explored [7], where these designs assume that L0 DC tag comparison is performed within the execute stage after the effective address has been computed. This approach requires a very small L0 DC and/or a slow clock rate to be feasible. Probably the most similar technique to the proposed design is the practical DFC (PDFC) [3]. This approach speculatively performs an L0 DC tag check in parallel with the effective address generation. The speculative L0 DC access is only attempted when the load or store displacement is small so that the *L0 index* field is unlikely to be updated and sometimes the speculative access fails due to the index field getting updated. In contrast, the L0 DC in the proposed design is accessed after the effective address generation, so more accesses can be obtained from the L0 DC. This PDFC design also assumed that the L0 DC data could be accessed in the same cycle as the effective address generation, but after the computation of the *L0 offset* field. Thus, a very small L0 DC and/or an L0 DC implementation in flip-flops is required to make this design feasible. The PDFC approach also has a more complicated L0 DC line fill strategy. In contrast to these various FC approaches, the proposed design can support a much larger L0 DC due to timing issues since the address calculation is performed before the L0 DC access. A larger L0 DC can significantly improve the L0 DC hit rate.

A zero-cycle load (ZCL) approach has been used to improve performance by reducing the average latency of loads [2]. Associated with the instruction address is a cache that contains the predicted base register value and either an index register value or a displacement, which are all accessed during the IF stage. Fast address calculation (FA) is used during the ID stage to speculatively access the L1 DC [1]. This approach increases data access energy as the L1 DC is accessed twice when the speculative address calculation fails. Also, the pipeline becomes more complicated as the L1 DC is accessed both during the ID stage and the MEM stage.

Multiple techniques have been proposed to make L1 DC accesses more energy efficient. Nicolaescu et al. propose to save the L1 DC way of the last 16 cache accesses in a table (WC), and each memory access speculatively performs a fully associative tag search on this table (CP, CS). If there is a match, then only the corresponding way is activated [20]. Way halting (WH) is another method for reducing the number of tag comparisons [31], where partial tags are stored in a fully associative memory (the halt tag array) with as many ways as there are sets in the cache. In parallel with decoding the word line address the partial tag is searched in the halt tag array. Only for the set where a partial tag match is detected can the word line be enabled by the word line decoder. This halts access to ways that cannot contain the data as determined by the partial tag comparison. Way halting requires a specialized SRAM implementation that might have a negative impact on the maximum operational frequency (CS). WH for an L1 DC could be combined with the L0 DC in the proposed design to reduce energy usage even further (CM).

A tagless cache (TLC) design has been proposed that uses an extended TLB (ETLB) to avoid tag checks [24]. While the TLC approach can significantly reduce energy usage, the authors assume the ETLB is accessed first to subsequently allow accessing a single L1 DC data array, which could either increase the cycle time or require an additional cycle to service an L1 DC access (CP). Our L0 DC approach could be used in conjunction with the TLC approach as the ETLB can be avoided when there is an L0 DC hit or the L0 DC detects that the L1 DC way is already known (COM). The TLC approach does not avoid TLB accesses (TD). Finally, the use of a TLC requires dealing with synonyms, homonyms, and other problems associated with virtually addressed data accesses, which is more difficult in an L1 DC.

The tag check elision (TCE) approach stores an L1 DC way with each integer register [34]. TCE stores a bound with every register to memoize L1 DC ways, which in their evaluation was a 29-bit value (MS). The TCE approach requires two comparisons and an addition to verify that the effective address of the memory reference is within the bounds of the cache line as well as an extra addition and a bound read and write each time an integer register is incremented by a value (CP, HC). Our memoization of L1 DC ways in L0 DC lines requires much less space and is much simpler.

4.7 Conclusions

This chapter shows that an L0 DC can be effectively utilized. L0 DC accesses without a performance penalty on misses is possible by detecting instructions that update a register whose value will be dereferenced by a load or store and storing that base register in a small structure that is accessed earlier in the pipeline. Likewise, a large fraction of memory references use a zero displacement, which also allows L0 DC references to occur earlier in the pipeline. Utilizing an L0 DC is appropriate as the base register value for a memory reference is not always available and an L1 DC can always be directly accessed without a delay compared to a conventional processor in these cases. This chapter also showed that the data access energy savings are significant and that unlike the traditional use of an L0 DC, performance can be improved as opposed to being degraded. Furthermore, the proposed design for utilizing an L0 DC requires no ISA changes or compiler support.

CHAPTER 5

DECREASING THE MISS RATE AND ELIMINATING THE PERFORMANCE PENALTY OF A DATA FILTER CACHE

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 chapter, a new DFC design is presented that both decreases the DFC miss rate and completely eliminates the DFC performance penalty. First, this chapter shows that a DFC that lazily fills each word in a DFC line from a level-one data cache (L1 DC) only when the word is referenced is more energy efficient than eagerly filling the entire DFC line. Second, this chapter demonstrates 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. Finally, this chapter presents a method that completely eliminates the DFC performance penalty by only accessing the DFC when a hit is guaranteed. Using these DFC techniques this chapter then shows that data access energy usage can be significantly improved with no performance degradation.

5.1 Introduction

It has been estimated that 28% of embedded processor energy is due to data supply [6]. 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 a level-one data cache (L1 DC) and can still service a reasonable fraction of the memory references [14, 15]. 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 [7, 9, 10, 14, 15, 27] has been claimed to be unrealistic as it can adversely affect L1 DC area and significantly increase the energy usage for each L1 DC access [3]. A multicycle DFC line fill is also problematic when it interferes 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.

In this chapter a new design is proposed that utilizes a DFC without the aforementioned problems. The proposed design for effectively using a DFC makes the following contributions. (1) The design shows 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) This design provides 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) A method is presented that completely eliminates the DFC miss performance penalty by only accessing DFC data when a hit is guaranteed.

5.2 Evaluation Environment

This section describes the experimental environment used in the following three sections of this chapter. The design is evaluated using the 9 C benchmarks from the SPECint 2006 benchmark suite compiled using *gcc* with the *-O3* option. The ADL simulator [21] was used to simulate both a conventional MIPS processor as the baseline and a modified processor as described in this chapter. Table 5.1 shows other details regarding the processor configuration that are utilized in the following simulations. The ADL simulator was used in combination with CACTI [17, 18] 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 5.2 shows the energy for accessing various components in the L1 DC and DTLB. Table 5.3 shows the energy for accessing various components of the DFC. CACTI does not provide energy values for very small caches. Thus, the energy is estimated 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, similar estimations are made 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.

Leakage energy was gathered assuming a 1 GHZ clock rate. *DFC line sharing* (LS), *DFC data packing* (DP), and *DFC word metadata* will be described in Section 5.4.

Table 5.1: Processor Configuration

page size	8KB
L1 DC	32KB, 64B line size, 4-way associative
DTLB	32 entries, fully associative
DFC	direct mapped, 32B line size, 128B to 1KB cache size

Table 5.2: 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	70.355 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

Table 5.3: 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

5.3 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. Section 5.5

shows 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 [7,9,10,14,15,27]. 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 [3]. 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.

In this chapter *DFC lazy word fill strategy* is used where each word is only filled when the word is referenced and is not resident in the DFC. This design assumes 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. Figure 5.1 shows the information in the DFC line that is used for the *DFC lazy word fill strategy.* An *f* (filled) bit is associated with each word in the DFC line indicating if the word is resident. 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. The design uses 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 only a single word is written to the line at a time. However, a word is not allocated on a *line hit+word miss* when there is a byte or halfword store as the processor would have to read the word from the L1 DC first in order to ensure that the entire word is resident in the DFC.

V	tag	L1DC way	f	data word	...	f	data word

Figure 5.1: Components of a DFC Lazy Word Filled Line

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 the 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. In order to still provide a benefit for these cases, the L1 DC way corresponding to the L1 DC (sub)line that holds the data is stored along with the DFC line, as shown in Figure 5.1. 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. 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 this 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.

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-1 DFC accesses requiring less energy, where n is greater than one.

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. Figure 5.2 shows the fraction of words within eagerly filled lines in a direct-mapped DFC with a 32B line size that are referenced before the line was evicted. The fraction is quite small due to frequent conflicts between DFC lines even though the initial referenced word that caused the line to be filled is counted as being referenced. Thus, even a 1024B DFC with eager line filling has about 74% of the words within a line not referenced before the line is evicted.

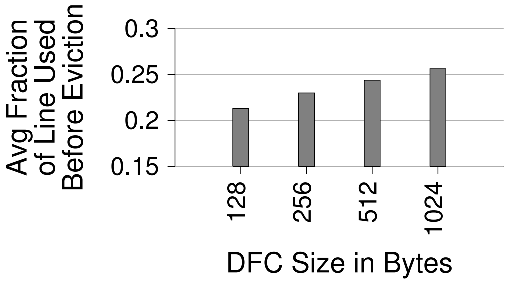


Figure 5.2: Fraction of Eagerly Filled DFC Lines Referenced before Eviction

Figure 5.3 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 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).

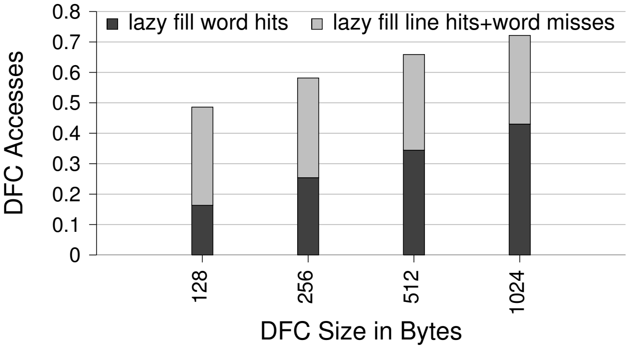


Figure 5.3: Taxonomy of DFC Accesses

Figure 5.4 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 is the energy used for accessing the DTLB, L1 DC, and DFC. The baseline at 1.0 is for a processor without a DFC. Different L1 DC bitwidths were simulated 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 5.2. The height of the eager fill bars are labeled 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. 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 1024B DFC, using lazy fill and memoizing the L1 DC way reduces data access energy by about 42%.

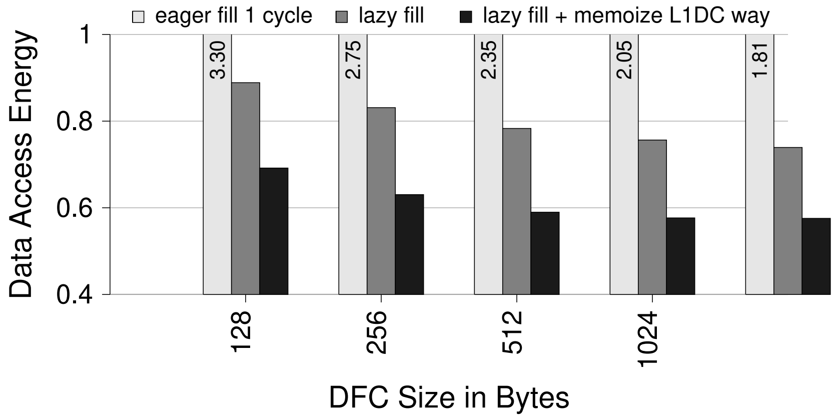


Figure 5.4: Data Access Energy with Eager vs Lazy Filling

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 as it will delay subsequent accesses to the DFC or L1 DC.

One solution is to provide separate read and write ports to the DFC so that DFC word fills (DFC writes) can occur in parallel with servicing DFC loads (DFC reads). However, there are still disadvantages with this solution. First, a DFC implemented with a two-port SRAM will require more circuitry, resulting in more area and power. Second, there will still be read accesses to the L1 DC on DFC load misses, which results in a structural hazard for the L1 DC read port during a DFC multicycle line fill, requiring a delay of the DFC line fill for at least one cycle to service the DFC miss. Third, DFC store hits will result in a structural hazard for the DFC write port during DFC line fills, which will also delay a DFC line fill. Finally, dealing with multiple outstanding DFC line fills will require even 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, Figure 5.2 shows 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.

Eagerly filling words into a DFC line will only improve energy efficiency if the remaining words will be referenced and the energy for performing the entire DFC (sub)line fill, which will not require subsequent DTLB accesses and L1 DC tag checks, will be less than referencing these words from the L1 DC. However, utilizing way prediction to access the L1 DC will also reduce energy usage as only a single L1 DC tag array and data array need to be accessed.

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

Since it was determined in Section 5.3 that a DFC lazy word filled approach is more energy efficient than an aggressive DFC eager line fill approach when a DFC causes no performance penalty, this section now presents 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. In order to make better use of this available space in a lazy word filled DFC line, the proposed design allows 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 decreases the DFC miss rate as there will be fewer DFC line evictions and increases the amount of data the DFC will likely hold. However, if multiple L1 DC (sub)lines refer to words corresponding to the same position in the DFC line, only the most recently referenced word is retained.

This section assumes a direct-mapped DFC where a single DFC line is associated with each DFC set. One, two, or four L1 DC (sub)lines are allowed 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 5.5 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. Not shown in the figure is LRU information for the L1 DC (sub)lines that share a DFC line, which is used to determine which L1 DC (sub)line to evict on a DFC line miss.

Table 5.4 shows one option where different words within distinct L1 DC (sub)lines can reside in the same DFC line at the same time. This option is referred to 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

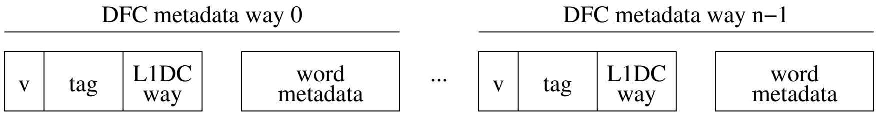


Figure 5.5: DFC Line Metadata

is resident or not, replacing the f (filled) bit in Figure 5.1. 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 this design allows 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.

Table 5.4: DFC Metadata for Sharing Data Words

Code	Interpretation
0	4-byte value
1	not resident

Figure 5.6 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 metadata for these two lines are also shown to the right of each L1 DC line, using the metadata codes shown in Table 5.4. 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.

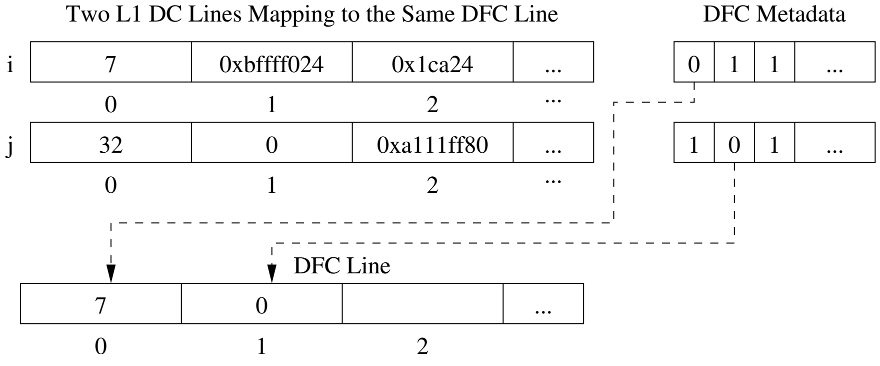


Figure 5.6: Example of Line Sharing

Table 5.5 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. This option is referred to 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, the design allows 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.

Table 5.5: 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

Two bits of metadata are required for each word within each L1 DC (sub)line sharing the same DFC line to support data packing, as shown in Table 5.5. 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 5.5. 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 5.7 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 5.6. The DFC metadata for these two lines use the metadata codes shown in Table 5.5. 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.

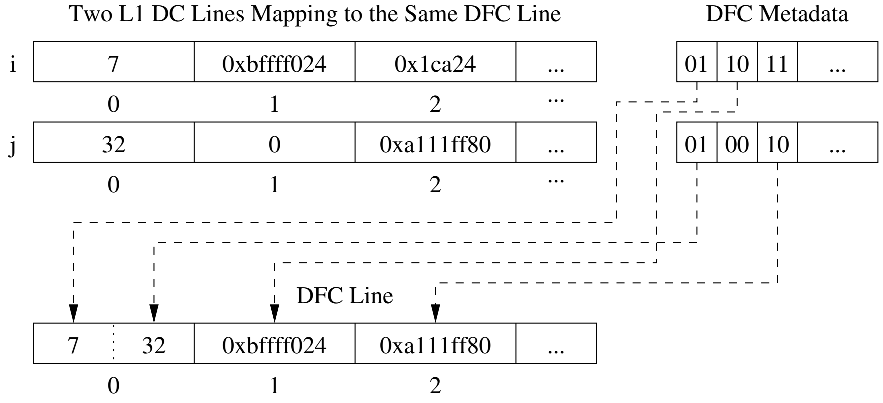


Figure 5.7: Example of Line Sharing and Data Packing

Figure 5.8 shows the taxonomy of the data values stored in a DFC. 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* value. The specific DFC word metadata within

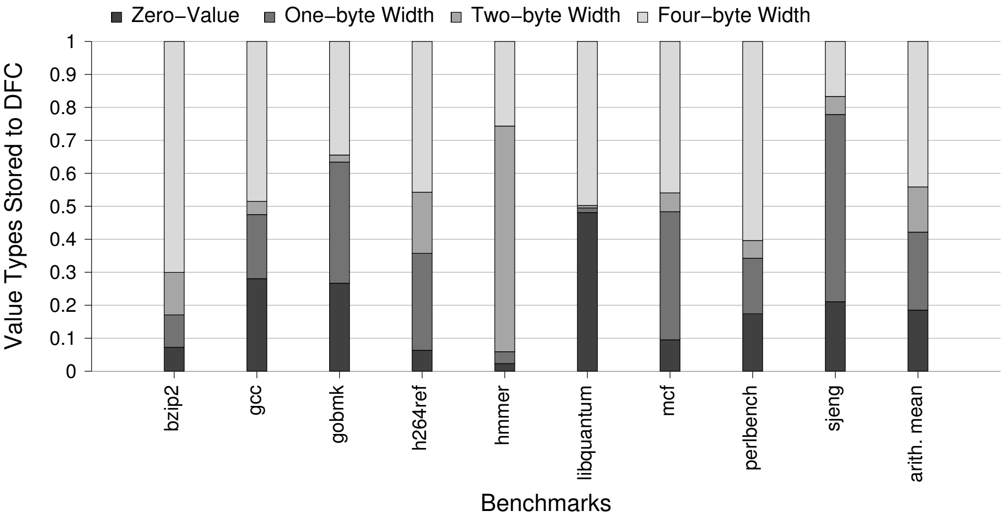


Figure 5.8: DFC Data Value Taxonomy

the set of corresponding DFC words is selected by which DFC tag matches and this metadata will indicate if the word is resident.

Figure 5.9 shows the average utilization of words in the DFC for the different *line sharing* (LS) and *data packing* (DP) techniques. Utilization indicates the average number of resident words that are in the DFC after each 1000 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 5.10 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 line misses. Data packing increases the number of word hits within the line, but does not change the number of line misses. Thus, the height of the bars with and without data packing are the same when sharing the same number of L1 DC (sub)lines. As the DFC size increases, the sum of the DFC word hits and line hits+word misses also increases. The figure also shows how valuable line sharing and packing is for improving the DFC hit rate. For instance, a 256 byte DFC that shares 4 L1 DC sublines with data packing (4xLS+DP) provides about the same DFC word hit rate (42%) as a 1024 byte DFC with no line sharing (43%). These results show that line sharing and data packing provide more flexibility so that the DFC can often better adapt to the data reference patterns in

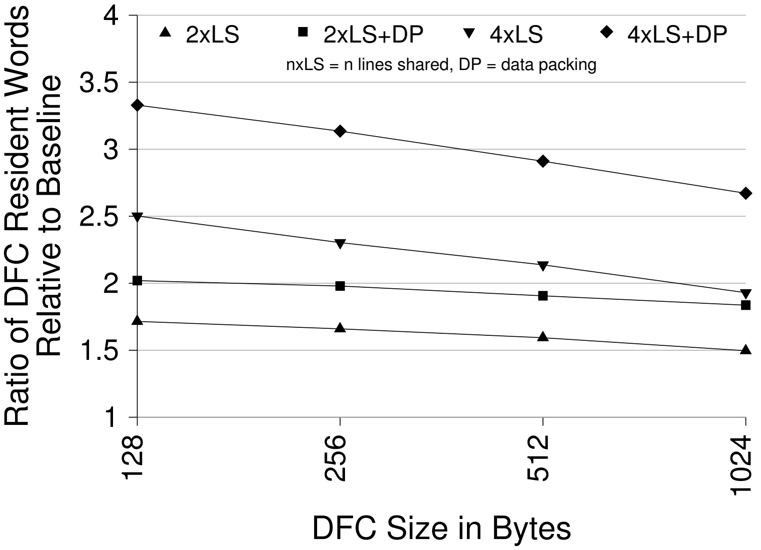


Figure 5.9: DFC Data Utilization

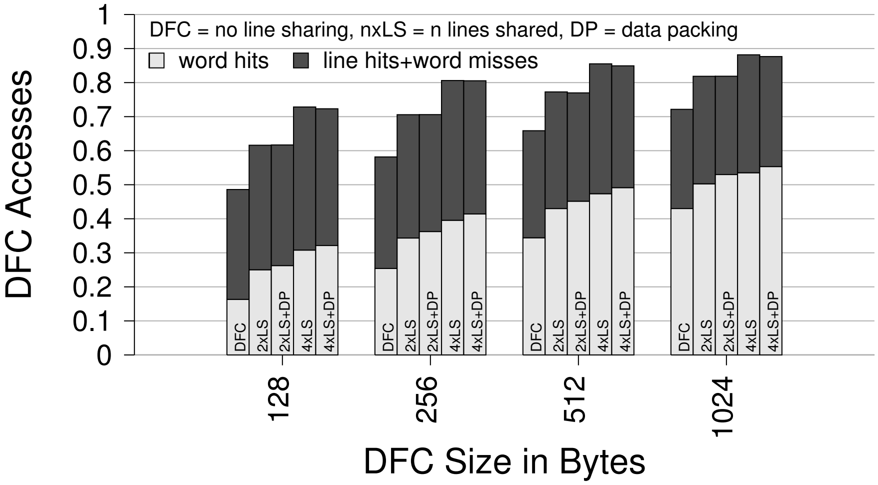


Figure 5.10: DFC Data Hit Rates

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 5.11 shows the average data access energy usage for the different combinations of line sharing and data packing techniques varied for different DFC sizes. 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

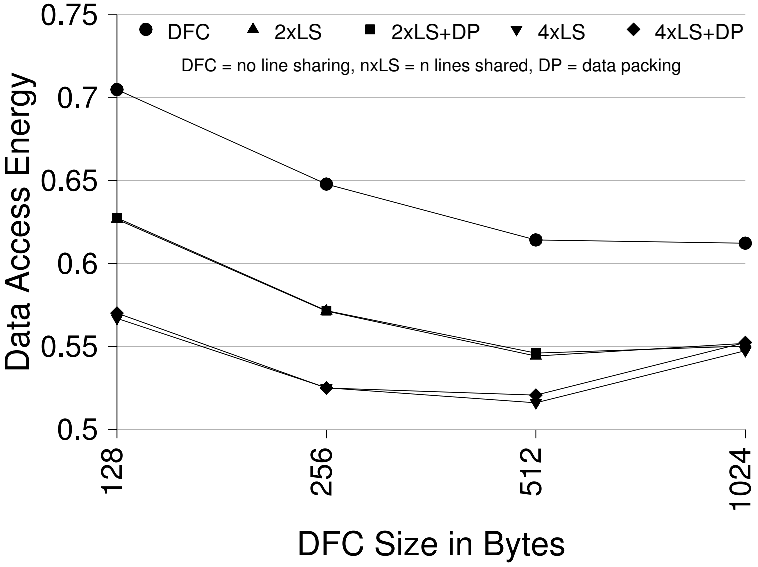


Figure 5.11: Data Access Energy

contention for DFC lines 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 5.9 and 5.10 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 only slightly less than without data packing.

Figure 5.12 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 5.11. 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 1024 byte DFC. Thus, the 512 byte DFC has slightly lower total data access energy.

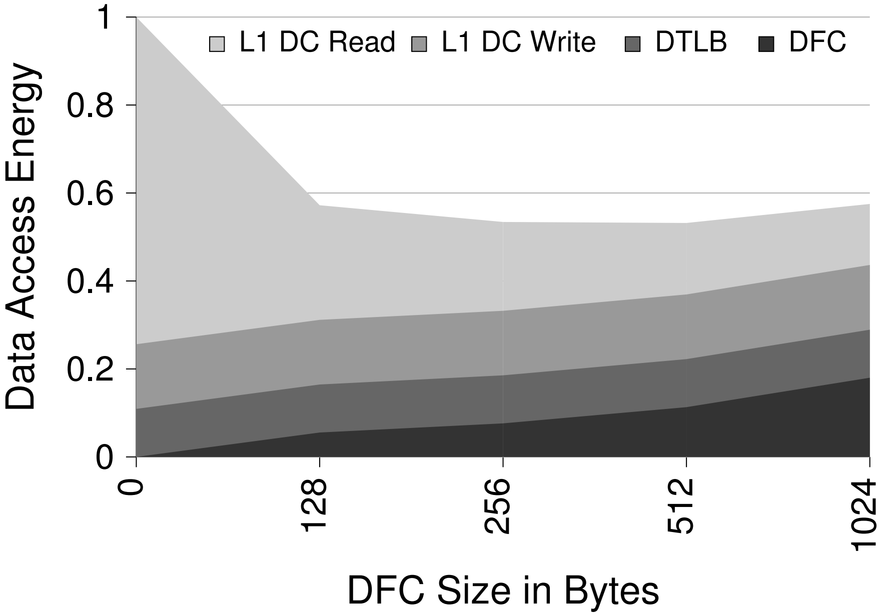


Figure 5.12: Component Data Access Energy

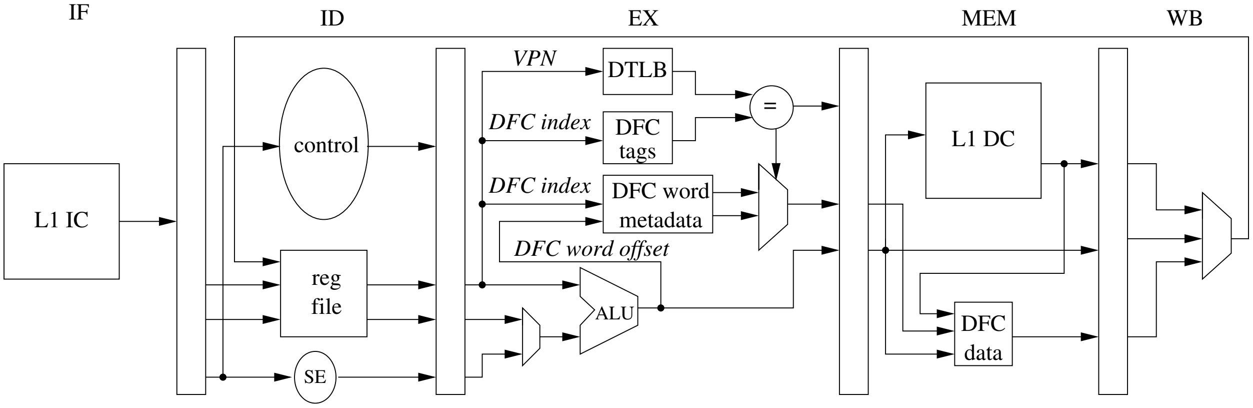


Figure 5.13: Modified Datapath to Support Guaranteed DFC Hits

5.5 Eliminating the DFC Miss Penalty by Only Accessing DFC Data on Guaranteed 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. A traditional instruction pipeline was revised to only load data from a DFC when a DFC word hit is guaranteed. DFC word hits are guaranteed 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 the processor can use the L0 DC data if the metadata shows the word is resident or the processor can use the L1 DC way if the metadata shows that the line is resident but the word is not. Otherwise, if the processor 1) is not able to speculate because the offset was too large, 2) speculatively accesses the metadata, but the calculation of the effective address modifies the bits used to access the metadata, or 3) has a line miss in the metadata, then the processor must access the L1 DC, but now with no performance penalty. Figure 5.13 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 6.1) of the base register value is used to speculatively access the DTLB during the EX stage. 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 6.1) of the base register value is used in the EX stage to speculatively access the DFC tag arrays so that 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 6.1) 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 the proposed 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 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. An inclusive cache policy and a DFC write-through policy are used 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 this proposed DFC design as the processor does 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 5.13, 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.

Figure 5.14 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 29.5% for a 1024B 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 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 was accessed and that either the entire line was not resident or the speculative address generation caused the DFC block number to change so that 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 DTLB 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 5.15 shows the data access energy when accessing DFC data only on guaranteed hits and without speculation (accessing DFC metadata after address generation) for each DFC size with its best configuration. 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 speculation, which is due to two reasons. First, useless DFC

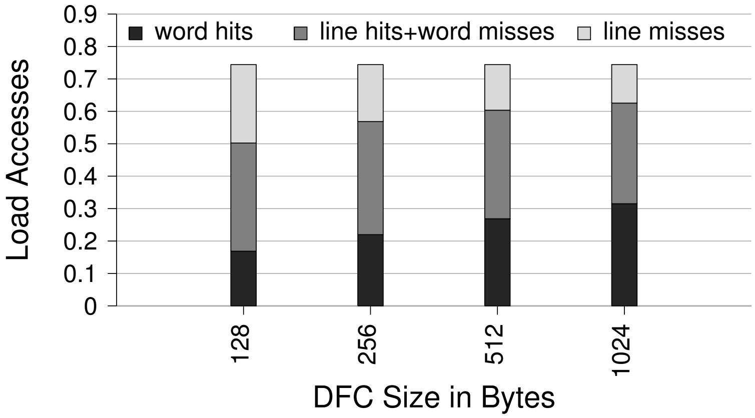


Figure 5.14: Taxonomy of Load Accesses

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 31.2% for a 128B DFC to 36.4% 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 5.15.

5.6 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 [33]. 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 that a separate frequent value L1 DC data array was accessed first to avoid the regular L1 DC data array if the value was frequent [30]. 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

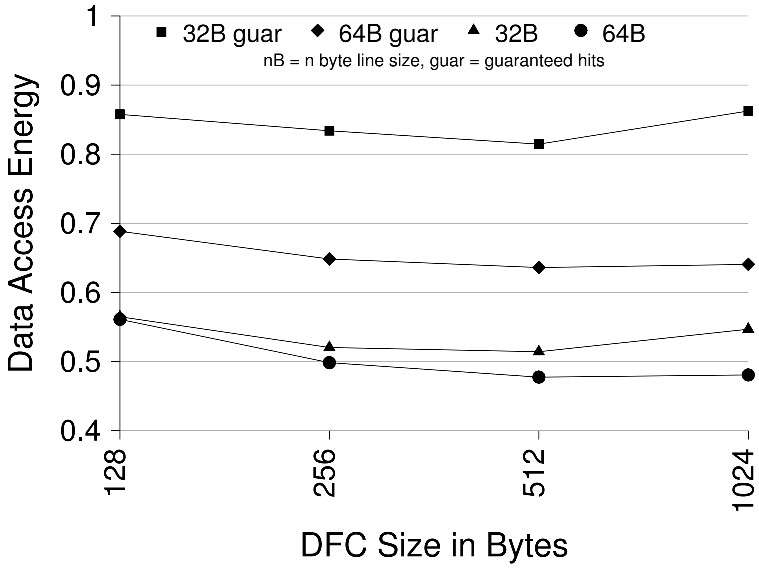


Figure 5.15: Data Access Energy with Guaranteed DFC Hits

replacement for an L1 DC has also been designed to hold frequent values [32]. 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 or not, which 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 [23]. 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 [28]. 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 [12]. 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 [13]. 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, no prior DFC or first-level cache technique 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 speculative tag access approach is used to reduce the energy for accessing the L1 DC [4]. 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. This design uses a similar speculative tag access approach, but instead accesses 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 [7]. This approach requires a very small DFC and/or a slow clock rate to be feasible. The practical DFC (PDFC) speculatively performs an DFC tag check in parallel with the effective address generation [3]. The speculative DFC access is only attempted when the load or store displacement is small so that 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. In contrast to these approaches, our design can support a much larger DFC due to timing issues since the DFC data access occurs in the MEM pipeline stage only after a DFC hit has been guaranteed in the EX stage.

A prior work speculatively accesses the DFC in the MEM stage when a reference is predicted to hit in the DFC by performing a partial tag comparison [16]. This approach is similar to our approach of guaranteeing a hit in the DFC, though their approach may occasionally result in a performance delay when the prediction is incorrect. Performance can also be potentially affected due to write backs of dirty DFC lines in their approach. Their approach also did not lazily fill DFC lines or use line sharing or data packing to decrease the miss rate.

5.7 Conclusions

This chapter described a design that allows a DFC to reduce energy usage and not degrade performance. This chapter showed that a DFC lazy word fill approach is more energy efficient than a DFC eager line fill approach. This chapter 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. Finally, a method was presented to eliminate the DFC miss performance penalty by only accessing DFC data when a DFC hit is guaranteed. This design should allow a DFC to be efficiently utilized in embedded processors.

CHAPTER 6

DECREASING THE MISS RATE AND THE MISS PENALTY OF A L1 DC

Level-one data caches (L1 DCs) need to be small to match the speed of processors and to reduce L1 DC energy usage. However, smaller cache sizes result in higher L1 DC miss rates, which can degrade performance. We introduce an L1 DC line sharing technique at the granularity of individual words that attempts to retain data values longer in an L1 DC by compressing values in an efficient manner that has a minimal impact on the L1 DC access time. This technique reduces the L1 DC miss rate by increasing the effective L1 DC capacity. We also show that our technique decreases the number of words fetched between the L1 DC and the level-two cache (L2C), which reduces both L1 DC stall cycles and L1 DC bus contention. When sharing four L2 sublines in each L1 DC line, a processor with a direct-mapped 16KB L1 DC has its average miss rate percentage decreased from 10.3% to 5.5% and the average number of data words fetched from the L2C decreased by 49%.

6.1 Introduction

In order to maintain fast level-one data cache (L1 DC) access times on contemporary processors, L1 DC sizes in recent years have largely remained the same. L1 DC misses incur stalls as the data must be retrieved from higher levels of the memory hierarchy, such as the shared level-two cache (L2C). Non-blocking caches in out-of-order processors attempt to hide this delay by allowing the L1 DC to continue servicing loads while outstanding load misses are still being completed. Factors such as the complexity of the circuitry as well as bus contention means that this approach can be extended to allow only a limited number of outstanding L1 DC misses to remain in flight before the L1 DC must be stalled. In addition, as the instruction issue width for OoO processors increases, loads often become the bottleneck for performance.

We introduce an L1 DC line sharing technique that allows multiple level-two (L2) cache sublines to share a single L1 DC line at a word-level granularity.[1] This technique allows a single L1 DC line to simultaneously hold data values from the same corresponding word in multiple L2 sublines by compressing these values into a single word. Furthermore, a subblocked L1 DC line can simultaneously hold values from multiple L2 sublines even if values cannot be compressed when they belong to separate subblocks in the same L1 DC line. By using sign-extension, we allow multiple values to be placed together inside a single word if they can be represented in halfword-width values, byte-width values, or zero-width values (a value of zero). Our compression technique has minimal impact on the L1 DC access time as the L1 DC line offset of the address doesn't change, which allows the same word to be loaded from an L1 DC line regardless of the L2 subline to which it belongs. In effect, a load from a direct-mapped L1 DC using our line sharing approach would access a single word of data, but would access multiple L1 DC tags that are associated with that L1 DC line to determine if the accessed L2 subline is resident within the L1 DC line and would access word metadata to determine how to extract the value from the word. We show that this line sharing technique increases the effective capacity of an 16KB, direct-mapped L1 DC by as much as 60% (160%) when sharing two (four) L2 sublines. By allowing multiple L2 sublines to share words in a single L1 DC line, we retain values longer as an existing word value in a resident metaline doesn't need to be evicted if it can be stored together with a word value from the same corresponding word in the incoming L2 subline. By retaining values longer, not only do we decrease the miss rate, but we also decrease the miss penalty associated with L1 DC line fills as fewer words need to be filled from an L2 subline when some of the words in that L2 subline already reside in the L1 DC line. On average, we reduce the number of data words needed to be fetched from the L2C and placed inside the L1 DC by 41% (48%) for a direct-mapped 16KB L1 DC that shares two (four) L2 sublines in each L1 DC line. Although our approach requires two (four) times the number of L1 DC tag bits to be accessed when we allow two (four) L2 sublines to share the same L1 DC line, our approach reduces data access energy usage as the L1 DC tag memory requires less power to access than the much larger L1 DC data memory and the number of L2 accesses is also decreased.

[1] We define an L2 subline as the portion of an L2C line that maps to an L1 DC line. Our L1 DC line sharing technique does not imply that the L2C line size must be larger than the L1 DC line size and does not imply that L2C lines are subblocked.

This paper makes the following contributions. (1) To the best of our knowledge, we present the first cache line compression technique that shares lines at the granularity of individual words that has a minimal impact on the cache access time. Sharing corresponding data words between multiple cache lines and allowing arbitrary words from these cache lines to be nonresident provides more flexibility for compressing values. Hence, line sharing is more effective than attempting to compress entire cache lines. (2) We outline novel L2 subline replacement and fill policies within a shared L1 DC line that significantly reduces unnecessary fetches of words from the L2C. (3) We provide an extensive empirical evaluation of our L1 DC line sharing technique.

The remainder of this paper is organized as follows. In Section 6.2 we describe how multiple L2 sublines can be shared within each L1 DC line. We illustrate in Section 6.3 how multiple word values can be compressed into a single word within an L1 DC line and how the word value can be efficiently decompressed when accessed. In Section 6.4 we outline when we fill the nonresident words in a L2 subline. We detail in Section 6.6 the processor design and parameters used to evaluate our approach. In Section 6.5 we illustrate the L2 subline replacement and line fill policies within the L1 DC. Section 6.7 presents the results of our analysis. We contrast in Section 6.8 our approach with other techniques that compress data in the L1 DC. We propose future evaluations of how line sharing affects other L1 DC parameters or techniques in Section 6.9. Finally in Section 6.10 we provide the conclusions of the paper.

6.2 Sharing Words between Multiple L2 Sublines

A data value in an L1 DC is accessed by using the *set index* and *line offset* portions of the address (see Figure 6.1) to index into the data line array and retrieve the data within the line, respectively. For a load from an m-way set associative cache, the data in all m ways are accessed and if there is a matching tag in the tag array, then only the data word associated with the matched tag is forwarded to the processor. In order to not affect the L1 DC access time, it's imperative that the location of the data word within the L1 DC line not be affected by accessing a compressed value so that the L1 DC data word can be accessed in parallel with performing the L1 DC tag check. To accomplish this, we restrict the scope of compression to a set of predetermined categories that allows a data word to be uniformly accessed from the L1 DC and only affects the logic for extracting the value from the data word. In this section we assume a direct-mapped L1 DC where a single L1 DC line is associated with each L1 DC set for ease of explanation. Our line sharing approach can be easily extended to set-associative L1 DCs and we show results for 1-way, 2-way, and 4-way set-associative line-shared L1 DCs in Section 6.7.

tag	set index	line offset

Figure 6.1: Partitioning of Address to Access the L1 DC

Assume an L2 subline is the portion of an L2 line that corresponds to an L1 DC line. During an L1 DC line fill, the L2 subline where the data resides is fetched from the L2C and placed inside the corresponding L1 DC line. An L1 DC line contains m word slots, where m is the L1 DC line size in words. Upon an L1 DC line fill, word 0 of the L2 subline is placed inside word slot 0 of the L1 DC line, word 1 of the L2 subline is placed inside word slot 1 of the L1 DC line, and so forth. We allow two or four L2 sublines to share each L1 DC line depending on the hardwired L1 DC configuration.

Figure 6.2 shows the organization of a direct-mapped line-shared L1 DC with n lines and m L2 sublines that can be placed in each L1 DC line. We refer to each L2 subline that can reside in an L1 DC line as a *metaline*. A conventional L1 DC line has a single valid bit and tag associated with it. In contrast, a line-shared L1 DC cache has a valid bit and tag associated with each metaline (L2 subline) that can simultaneously reside within the same L1 DC line. In addition, there are *word metadata* for each metaline, where a few bits are associated with each data word to describe whether or not a data value is resident and how it is compressed within the word when it is resident. Not shown in the figure is least recently used (LRU) information for the metalines that share an L1 DC line, which is used to determine which metaline to evict on an L1 DC line miss.

If the tag comparison on a data reference indicates that the referenced L2 subline is not currently sharing the L1 DC line, we evict one of the metalines according to the policy specified in Section 6.5 and mark all of the words associated with that evicted metaline as not resident. Likewise, the tag comparison may indicate that the referenced L2 subline is currently sharing the L1 DC line, but the referenced word within the L2 subline may not be resident within the L1 DC line. In this case the processor loads not only the nonresident word in the L2 subline, but all other nonresident words associated with the same L2 subline. Rather than evicting the values associated with the other

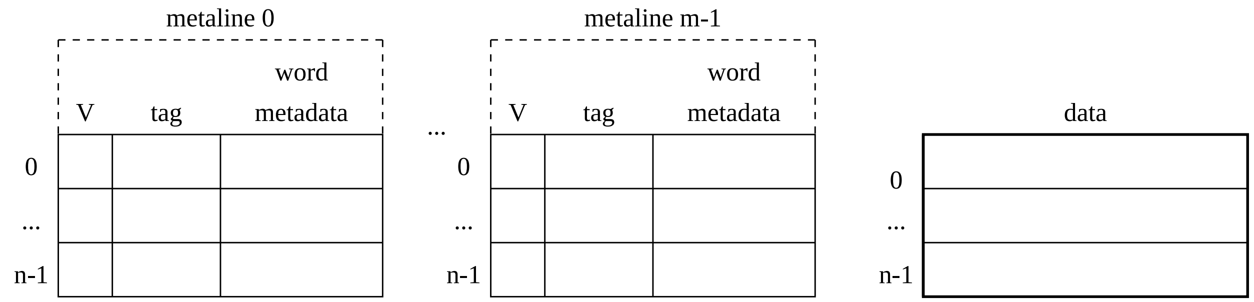


Figure 6.2: L1 DC with m Metalines for Each L1 DC Line

metalines currently sharing the L1 DC line, we allow these other metalines to share the L1 DC line with the words associated with the incoming L2 subline when possible. When placing the values of the words comprising the incoming L2 subline into the L1 DC line, we check for each word slot to see if the resident values and the incoming L2 subline can share the word slot. This approach decreases the miss rate as there are fewer L1 DC word evictions and the amount of data stored inside the L1 DC increases.

The L1 DC word metadata is organized into arrays, where there is a separate array for each set of corresponding words of L1 DC metadata from the L2 sublines. The specific array element to be accessed is determined by the *L1 DC word offset* value. The specific L1 DC word metadata within the set of corresponding L1 DC words is selected by which L1 DC tag matches and this metadata will indicate if the word is resident and how to extract the value from that word.

6.3 Compressing and Decompressing L1 DC Data

Our line-shared L1 DC attempts to compress a data value within a word during a store instruction or when an L1 DC word is being filled from the L2C. Note that a store instruction requires an L1 DC tag check before the value is actually stored. Thus, how other values are stored in the word is known before storing the new value as word metadata associated with the corresponding word from all of the metalines sharing the L1 DC line is checked in parallel with the tag comparisons. The type of encoding for the value to be stored is determined after the value to be stored is available. The word metadata with the corresponding data word for each metaline associated with an L1 DC line may need to be updated depending upon the new value to be stored as values from other metalines will be marked as nonresident when they cannot be compressed with the new value. Note that we do not attempt to compress a value if we are storing a byte or a halfword to the upper halfword of a word.

Table 6.1 shows that when using only two metadata bits for each data word, values associated with the same word slot from distinct L2 sublines can sometimes be compressed to reside within an L1 DC word slot at the same time. When a word is loaded from the L2C, the processor checks if the value is *narrow width*, meaning that the value can be represented in fewer bytes than a full data word through sign extension. 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 L1 DC line. The third column of Table 6.1 indicates when each value can be shared with other values in the same word. A nonzero narrow width (2-byte) value (code 01) can be shared in the same word with any value that is not full width (code 10). In other words, a 2-byte value (code 01) can share a word with a zero value (code 00), a 2-byte value (code 01), or a nonresident value (11). A zero value (code 00) can be shared in the same word with any other value.

Table 6.1: L1 DC 2-Bit Word Metadata Encoding

Code	Interpretation	Can Share a Data Word with
00	zero value	00, 01, 10, 11
01	2-byte value	00, 01, 11
10	4-byte value	00, 11
11	not resident	00, 01, 10, 11

The placement within the word of nonzero narrow-width values for a given L2 subline will be based on the *L1 DC metaline way* of the L2 subline as shown in Figure 6.2. When two L2 sublines can share the same L1 DC line (2xLS), a nonzero narrow width value will be placed in the lower halfword if the *L1 DC metadata way* was zero and the upper halfword if the *L1 DC metadata way* was one. When four L2 sublines can share the same L1 DC line (4xLS), a nonzero narrow width value will be placed in the lower halfword if *L1 DC metaline way* was zero or one and the upper halfword if the *L1 DC metaline way* was two or three.

Figure 6.3 shows an example of line sharing with two L2 lines sharing an L1 DC line (2xLS). The L1 DC metadata for these two lines use the metadata codes shown in Table 6.1. Word 0 in each of the two L2 sublines can be compressed into word 0 of the L1 DC line as both values are narrow width (can be represented in 2 bytes). Word 1 in each of the two L2 sublines can be compressed into word 1 of the L1 DC line as word 1 of L2 subline j is zero and is not stored in the L1 DC line. At most one value for word 2 can be stored as the value in word 2 for L2 subline j is full width.

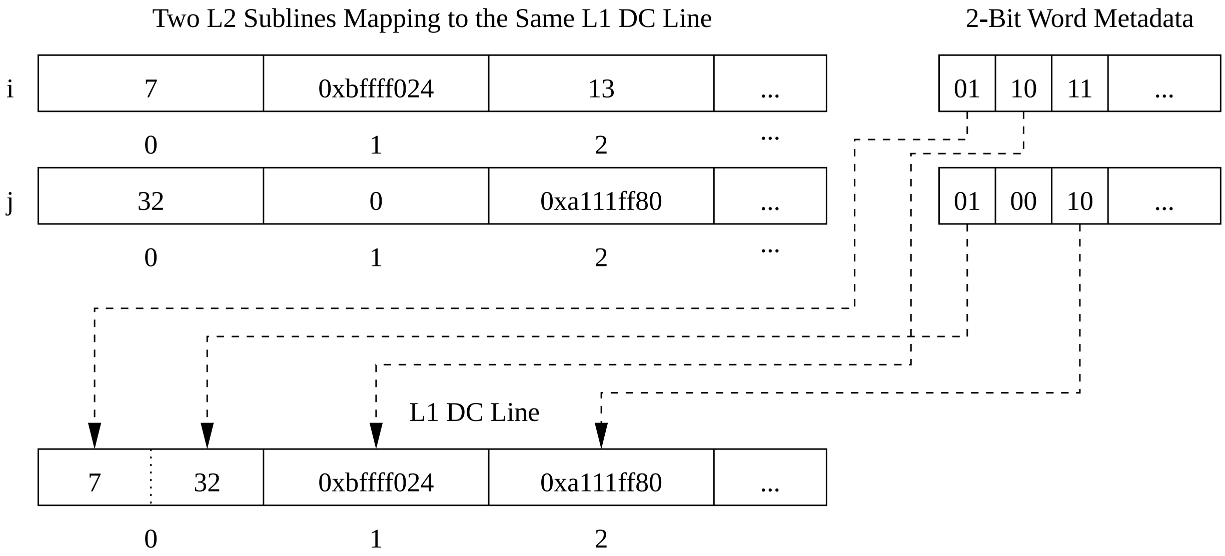


Figure 6.3: Line Sharing Example with 2-Bit Word Metadata

Figure 6.4 shows the taxonomy of the data values stored in an L1 DC for the SPECint 2006 benchmark suite. On average, 18.5% were zero values, which do not require an access to the L1 DC data to store the 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 compressed into a word with a value from a different L2 subline.

Table 6.2 shows an extended set of ways that data values can be compressed into a single word by using 3 bits for each word encoding. Code 001 allows a 1-byte sign-extended value to be represented, which has the advantage of allowing up to four byte values to share a single data word. If only two L2 sublines can share an L1 DC line, then this encoding would still reduce energy usage as the number of bytes being updated in the cache will be reduced. Code 011 indicates that the upper halfword will come from the upper half of the address used to access the L1 DC, which is comprised from the tag field and possibly a portion of the set index field (see Figure 6.1). The motivation for this encoding is that sometimes a word can contain a pointer to a value that is nearby to the address where the pointer itself is stored (within the same lower 16-bit or 64KB offset). Code 100 indicates that a common upper halfword is to be used. The idea is that often

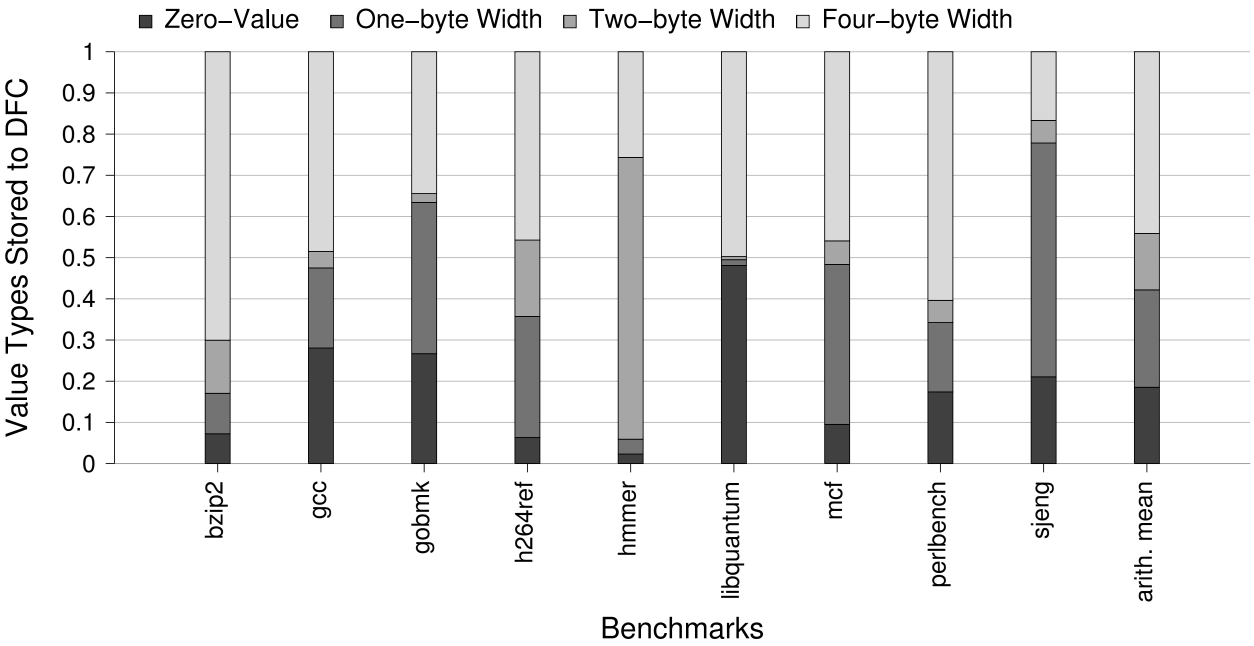


Figure 6.4: L1 DC Data Value Taxonomy

the same upper halfword value is used in different words in the same L2 subline (e.g. high half of pointer address). One common upper halfword value is stored with each metaline. If the valid bit for the common upper halfword is not set, then the upper halfword of a 4-byte value (code 110) being stored or filled will be placed in the common upper halfword. When there is a store word instruction or word fill, the L1 DC compares the upper halfword of the current value to be stored with this common halfword value associated with the entire metaline. If the current upper halfword matches this common upper halfword value, then only the lower halfword is stored in the data word. A valid bit is not set for the common upper halfword until a match is found indicating that the upper halfword value is the same as another word in the line. A counter for the common upper halfword is incremented on each match and the counter is decremented when a new value is stored that does not match the current upper halfword associated with the previous value or the current value using the common upper halfword becomes no longer resident. The common upper halfword can only be replaced when the counter is zero, which simplifies replacement since other word values within the same L1 DC line do not have to be updated. Code 101 contains the upper half of the $gp register, whose value is used to access global variables and is obtained at the start of the execution. If a global address is stored within this 64KB offset, then only the lower halfword of the value needs to be stored.

Table 6.2: L1 DC 3-Bit Word Metadata Encoding

Code	Interpretation	Can Share a Data Word with
000	zero value	000,001,010,011,100,101,110,111
001	1-byte value	000,001,010,011,100,101,111
010	2-byte value	000,001,010,011,100,101,111
011	upper half address	000,001,010,011,100,101,111
100	common upper half	000,001,010,011,100,101,111
101	$gp upper half	000,001,010,011,100,101,111
110	4-byte value	000,111
111	not resident	000,001,010,011,100,101,110,111

Figure 6.5 shows an example of line sharing with two L2 lines sharing an L1 DC line (2xLS) using the 3-bit word metadata codes shown in Table 6.2. Word 0 in each of the two L2 sublines can be compressed into word 0 of the L1 DC line as word 0 of L2 subline i has an upper halfword that is the same as the common upper halfword that is stored with this metaline and word 0 of L2 subline j is the upper halfword of the $gp value. Word 1 in each of the two L2 sublines can be compressed into word 1 of the L1 DC line as word 1 of L2 subline i also has the same upper halfword value as the common upper halfword and word 1 of L2 subline j has an upper halfword that has the same bits as the upper halfword of the address of the L1 DC line. Finally, word 2 in each of the two metalines can be stored in word 2 of the L1 DC line as word 2 of L2 subline i is zero.

A value is decompressed from a word in the L1 DC line when it is loaded from the cache due to a load instruction being executed. The word is loaded in parallel with performing the tag comparisons and checking the word metadata. Thus, accessing the data word is not delayed as the same *set index* and *line offset* of the address (see Figure 6.1) is used to access the word within the L1 DC line. The loading of a value from a conventional cache goes through a multiplexor as only a portion of the loaded word is used when a load byte (signed or unsigned) or load halfword (signed or unsigned) instruction is performed. The only change we require is that there is a greater number of values that will be fed into this multiplexor, which will now not only be controlled by the type of load instruction, but also by the word metadata encoding associated with that word value. Thus, loading a value using line sharing will increase the logic depth by at most one or two gates due to the use of a larger multiplexor to extract the value from the loaded word.

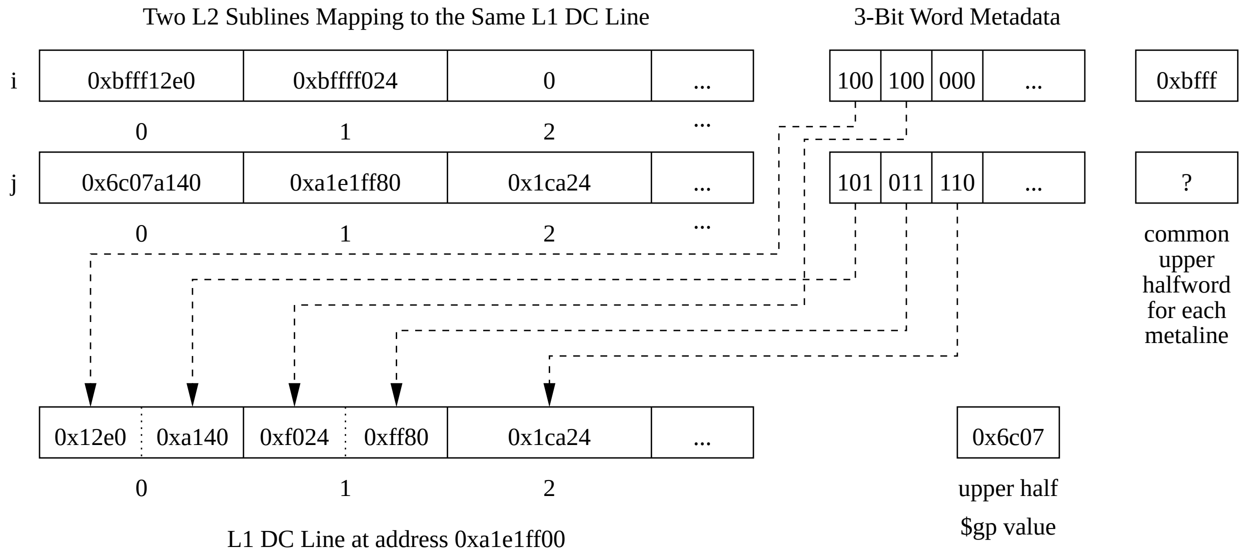


Figure 6.5: Line Sharing Example with 3-Bit Word Metadata

6.4 L1 DC Metaline Fill Policy

We define *metaline thrashing* as repeatedly fetching data words associated with metalines where most of the data words are not referenced before they are replaced. We selectively choose not to fill L1 DC lines if they could potentially evict useful data. There are three possible L1 DC access results: *word hit* (the accessed word is resident and other words in the same metaline may or may not be resident), *word miss+line hit* (the accessed metaline is resident within the L1 DC line, but the word in the metaline is not resident), and *line miss* (the accessed metaline is not resident within any of the L1 DC lines within the indexed L1 DC set). During an L1 DC access, it's possible that some words of the metaline are not resident for a *word hit* or *word miss+line hit*. An eager metaline fill approach would always fetch the missing words in a metaline from the L2C as this can potentially increase the L1 DC hit rate due to spatial locality. However, an eager metaline fill approach can result in metaline thrashing when two metalines in the same L1 DC line are alternatively referenced, as they would continuously replace the other metaline's data when the two data values cannot be compressed into the same word. We avoid this thrashing behavior in the following manner. If the L1 DC detects a *word hit* to a metaline that isn't the MRU metaline within the L1 DC line, then the L1 DC does not fetch the missing data words in that metaline from the L2C and updates the LRU information. Hence, the L1 DC doesn't evict data words from the MRU metaline within the L1 DC line. If the L1 DC detects a *word hit* to a metaline that is the MRU, then the L1 DC does fetch the missing data words in that metaline from the L2C. Thus, it takes two *word hit* references to a resident L1 DC metaline to trigger a fill of the nonresident values within that metaline.

6.5 L1 DC Metaline Replacement Policy for Set-Associative Organizations

The L1 DC metaline (L2 subline) replacement decisions become more complex as we move from a direct-mapped cache to a set-associative cache. Ideally, the most recently used (MRU) metalines will be mapped to separate L1 DC lines so they are not competing for a single line within an L1 DC set. For example, for a 4-way L1 DC using 4xLS (four metalines sharing a single L1 DC line), there are 16 metalines per set. It would be best if the four MRU metalines map to separate L1 DC lines in the set. Metaline thrashing can occur when the two MRU metalines map to the same L1 DC line within the L1 DC set, which will decrease the L1 DC hit rate and increase the number of words fetched from the L2C. In addition, maintaining LRU information for up to sixteen metalines is unnecessary complex in terms of circuitry. In our initial experiments we found that line sharing caused an increase in the miss rate and/or the number of words fetched from the L2C for some benchmarks in some configurations. Thus, we refined the L1 DC metaline replacement policy in a set-associative L1 DC (as well as the L1 DC metaline fill policy) to decrease thrashing and we refined the LRU information stored with each metaline to reduce the complexity of the LRU circuitry. After much experimentation, there are three metaline replacement policy refinements we found that eliminates most of the metaline thrashing and improves the overall L1 DC miss rate. The pseudocode for processing an L1 DC access with line sharing in a set-associative cache is shown in Figure 6.6.

```
select_metaline_to_replace(in set; out way, metaway) {
    way = LRU_line(set);

    // replacement refinement to support 2-level LRU info
    if (2xLS)
        metaway = LRU_metaline(set, way);
    else
        metaway = LRU_metaline_in_nonMRUgroup(set, way);
}

replace_metaline(in set; out way, metaway) {
    select_metaline_to_replace(set, way, metaway);
    invalidate_metaline(set, way, metaway);
    fill_metaline(set, way, metaway);

}

process_L1DC_access(in tag, set, word_offset) {
    if (line_hit(tag, set, way, metaway)) {
       if (word_miss(set, way, metaway, word_offset))

           // replacement refinement to separate 2 MRU metalines
           if (MRU_line(set) == way &&
              MRU_metaline(set, way) != metaway) {
                invalidate_metaline(set, way, metaway);
                replace_metaline(tag, set, way, metaway);

            }
            else
                fill_metaline(set, way, metaway);
        }

        // fill refinement for word hit
        else if (MRU_metaline(set, way, metaway))
            fill_metaline(set, way, metaway);
    }
    else
        replace_metaline(tag, set, way, metaway);
    return extract_value(tag, set, way, metaway, word_offset);
}
```

Figure 6.6: Processing an L1 DC Access in a Set-Associative Line-Shared Cache

First, we use a two-level LRU metaline replacement policy. For 2xLS (two metalines sharing a single L1 DC line), we find the LRU L1 DC line within the L1 DC set in the first level, and in the second level, we find the LRU metaline within that L1 DC line. Using a two-level LRU metaline replacement policy initally prevents the two MRU metalines within the entire L1 DC set from mapping to the same L1 DC line. We also use two levels of LRU information for 4xLS (four metalines sharing a single L1 DC line). In addition, we treat metalines in an L1 DC line as a member of one of two groups: group *a* for metalines 0 and 1 and group *b* for metalines 2 and 3. A two-byte narrow width value in a 4xLS line is placed in the lower halfword if the L1 DC metaline way is zero or one and is placed in the upper halfword if the L1 DC metaline way is two or three (see Section 6.3). We extend the two-level metaline replacement policy for 4xLS in the following manner. In the first level, we find the LRU L1 DC line. In the second level, we select the LRU metaline within the group that doesn't contain the MRU metaline. For example, assume there are four metalines 0, 1, 2, and 3 associated with a single L1 DC line. If the MRU metaline is in group *a* (metaline 0 or 1), then a metaline eviction would replace the LRU metaline from group *b* (metaline 2 or 3) as this would allow two narrow-width values from the two MRU metalines to share the same word in the L1 DC line. If we instead assigned the two MRU metalines of an L1 DC line to the same group, then narrow-width values stored in the two MRU metalines would be placed in the same halfword of the L1 DC word slot causing the first value placed within that word slot to be evicted.

Second, we decrease metaline thrashing in set-associative caches by selectively invalidating lines that could cause thrashing. If there is a *word miss+line hit* to a metaline that is not the MRU, but shares the L1 DC line with the MRU metaline within the entire L1 DC set, we mark the accessed metaline as invalid and return a miss. This approach decreases line thrashing as we don't allow the two MRU metalines to compete for a single L1 DC line. Instead, this approach forces the accessed metaline to be reallocated to another L1 DC line.

Third, sometimes a metaline is not filled due to other issues, which is not shown in Figure 6.6. A metaline is not filled when there is a *word hit* and the maximum number of outstanding L1 DC line fill requests has been reached since the memory access can be resolved without a line fill. Likewise, a prior outstanding line fill request is cancelled if it was a *word hit*, the limit on outstanding L1 DC line fill requests is reached, and a new *word miss+line hit* or *line miss* is encountered. Finally, a line fill associated with a *word hit* or *word miss+line hit* is not performed if there is already a pending line fill to the same line.

6.6 Evaluation Environment

In this section we describe the experimental environment used in the following section of the paper. 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 [21] to simulate both a conventional MIPS processor as the baseline and a modified processor containing a line shared L1 DC as described in this paper. Table 6.3 shows other details regarding the processor configuration that we utilize in our simulations. We use the ADL simulator combined with CACTI [17, 18] for the energy evaluation to model processor energy. CACTI was used assuming a 22-nm CMOS process technology with low-standy power (LSTP) cells and power gating. Tables 6.4 and 6.5 show the energy for accessing various components in the 16KB and 32KB L1 DCs, respectively. Leakage energy was gathered assuming a 1 GHZ clock rate.

Table 6.3: Processor Configuration

	Single-stage MIPS processor
L1 DC Processor	16/32KB, 64B line size, 1/2/4-way associative, 1-cycle hit time, 10-cycle miss penalty
L2C	512KB, 64B line size, 8-way associative, 8-byte bus width, 100 cycle miss time

Table 6.4: Energy for 16KB L1 DC Components (pJ)

L1 DC Size	16384B			
Associativity	1-way			
L1 DC Read	2.900			
Configuration	1xLS	2xLS	4xLS	4xLS-ext
Tag Array	0.381	0.728	0.981	0.981
Meta Array	0.000	0.378	0.718	0.951
Associativity	2-way			
L1 DC Read	4.129			
Configuration	1xLS	2xLS	4xLS	4xLS-ext
Tag Array	0.403	0.782	1.091	1.091
Meta Array	0.000	0.381	0.725	0.961
Associativity	4-way			
L1 DC Read	6.617			
Configuration	1xLS	2xLS	4xLS	4xLS-ext
Tag Array	0.448	0.892	1.275	1.275
Meta Array	0.000	0.388	0.739	0.982

Table 6.5: Energy for 32KB L1 DC Components (pJ)

L1 DC Size	32768B			
Associativity	1-way			
L1 DC Read	3.444			
Configuration	1xLS	2xLS	4xLS	4xLS-ext
Tag Array	0.701	0.927	1.499	1.499
Meta Array	0.000	0.647	1.226	1.823
Associativity	2-way			
L1 DC Read	5.065			
Configuration	1xLS	2xLS	4xLS	4xLS-ext
Tag Array	0.728	0.981	1.661	1.661
Meta Array	0.000	0.718	1.233	1.834
Associativity	4-way			
L1 DC Read	8.193			
Configuration	1xLS	2xLS	4xLS	4xLS-ext
Tag Array	0.782	1.091	2.019	2.019
Meta Array	0.000	0.725	1.247	1.854

6.7 Results

Table 6.6: L1 DC Hit Rates

L1 DC Configuration			Line Sharing Technique				
Size	Assoc.	Sub-block Size	1xLS	2xLS	2xLS-ext	4xLS	4xLS-ext
16KB	1-way	16B	82.9%	90.9%	91.1%	91.7%	92.1%
		32B	87.1%	92.7%	92.7%	93.2%	93.8%
		64B	89.7%	93.7%	94.1%	94.1%	94.5%
	2-way	16B	84.2%	91.7%	91.8%	92.5%	92.6%
		32B	88.7%	93.7%	93.8%	94.1%	94.3%
		64B	91.5%	94.7%	94.8%	95.0%	95.1%
	4-way	16B	86.0%	91.7%	91.9%	92.8%	92.9%
		32B	90.8%	93.9%	94.0%	94.5%	94.6%
		64B	93.2%	94.9%	95.0%	95.3%	95.4%
32KB	1-way	16B	90.5%	92.8%	92.9%	93.3%	93.4%
		32B	92.8%	94.2%	94.6%	94.6%	95.0%
		64B	92.7%	95.3%	95.3%	95.4%	95.6%
	2-way	16B	92.0%	93.2%	93.2%	93.8%	93.9%
		32B	94.2%	95.0%	95.1%	95.4%	95.4%
		64B	95.0%	95.9%	95.9%	96.1%	96.2%
	4-way	16B	92.4%	93.5%	93.5%	93.9%	94.0%
		32B	94.4%	95.2%	95.2%	95.5%	95.6%
		64B	95.5%	96.1%	96.1%	96.2%	96.3%

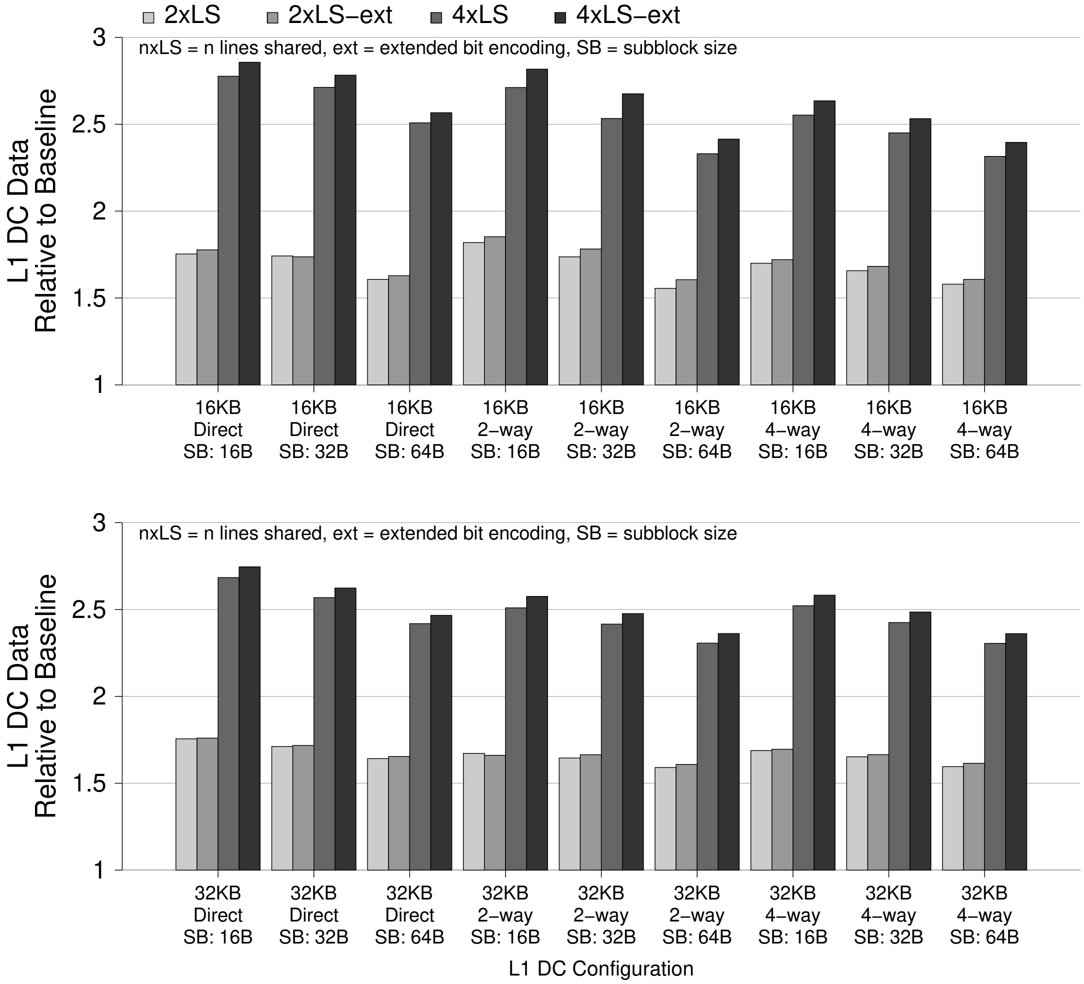


Figure 6.7: Data Utilization of L1 DC

Figure 6.7 shows how well we utilize the L1 DC in terms of storage. L1 DC utilization is a measure of the amount of data stored inside the L1 DC relative to the baseline. Here, the baseline is an L1 DC of the same size, associativity, and sub-block size but without line sharing and data packing. For 2xLS and 4xLS(-ext), the maximum amount of data we can hold relative to the baseline is two times or four times, respectively. The L1 DC size has little effect on the amount of data that can be stored relative to the baseline for our apporach. The difference between a 16KB, 4-way, 64 byte sub-blocked cache using 4xLS versus a 32KB cache with the same configuration is less than two percent. However, as we increase the sub-block size the amount of data we store relative to the baseline decreases. For a 16KB, 4-way L1 DC using 4xLS, the amount of data stored relative to the baseline decreases by roughly 24% as we move from a 16-byte sub-block size to a 64-byte sub-block size. Similarly, the amount of data stored relative to the baseline decreases as we increase the associativity. A direct-mapped, 16KB cache with a 64-byte sub-block size using 4xLS can store about 20% more data relative to the baseline than a 4-way associative 16KB cache with a 64-byte sub-block size. As previously stated, there are more opportunities for line sharing with a small sub-block size.

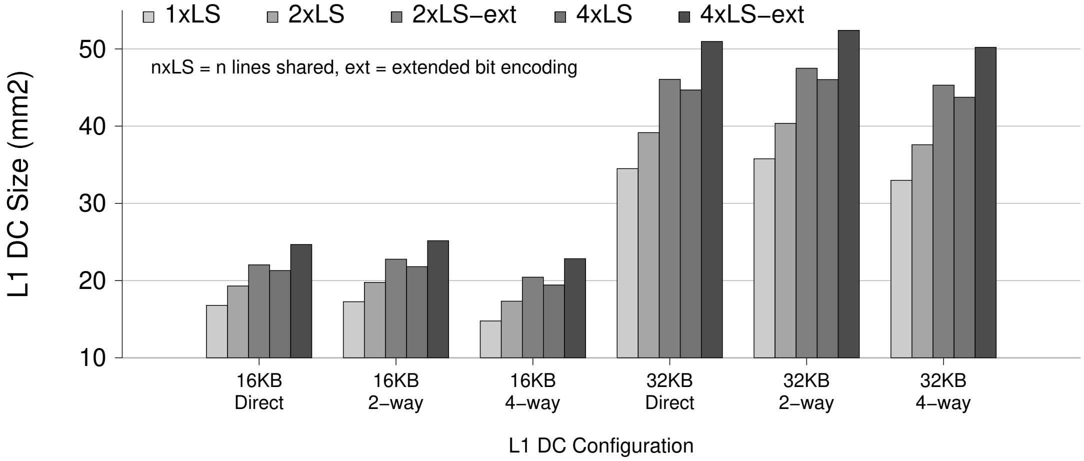


Figure 6.8: Increase in L1 DC Size

Figure 6.8 shows how our approach increases the size of the L1 DC for different L1 DC sizes and associativities. As we increase the associativity, the size of the cache decreases. This is because it is inefficient in terms of space to have long, narrow data structures as the decoding logic increases. By splitting the structure into multiple data arrays, the decoding logic decreases and the height of the L1 DC and the structure becomes more symmetrical. For 2xLS(-ext) and 4xLS(-ext), we increase the size of the L1 DC as we increase the number of tags per line by two and four times, respectively. As expected, 2xLS has the smallest footprint as this requires only two tags per L1 DC line and 32 bits (64B cache line = 16 words, 2 bits per word) per tag for the metadata array. For a 4-way, 16KB L1 DC, 2xLS increases the size of the L1 DC by roughly 17% for a 16KB cache and 14% for a 32KB L1 DC. Using 4xLS-ext has the largest footprint as it requires four tags per L1 DC line and 48 bits (64B cache line = 16 words, 3 bits per word) per tag for the metadata array. For a 4-way, 16KB L1 DC, 4xLS-ext increases the size of the L1 DC by 55% and by 52% for a 32KB L1 DC.

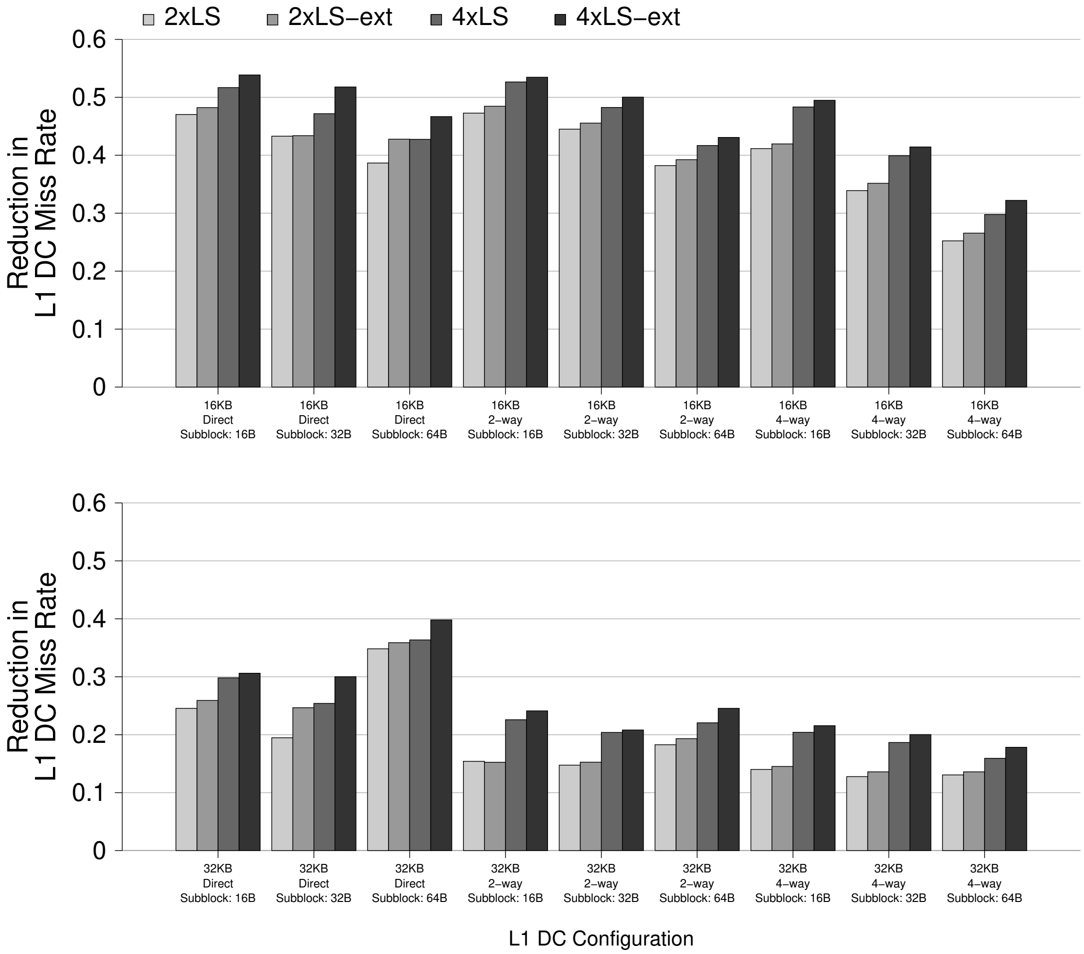


Figure 6.9: Reduction in L1 DC Miss Rate

Figure 6.9 shows the effect our approaches have on the L1 DC miss rate. Our approach allows an L1 DC to approach the performance of a higher-associativity data cache. The benefits of our approach rely on two key factors. The first is the taxonomy of values placed inside the L1 DC, which determines how often data can be compressed inside our L1 DC. Compressed values can share space with other compressed values and therefore can be retained longer as they are less likely to evict other values and are less likely to be evicted by other values. The second factor is how often references switch between meta-lines in a set. If all accesses are to the MRU meta-line, then there would be no benefit to our approach as the performance would perform the same as the baseline. However, if accesses switch between the lines in a set, then our approach will see benefits.

The factor that has the largest impact on the miss rate reduction is the associativity. For example, the miss-rate reduction for a 16KB cache with a sub-block size of 64 bytes using 4xLS decreases from 42.7% to 29.8% as we increase the associativity from 1-way to 4-way. This is because our approach obtains some of the benefits of a higher associative L1 DC, but there are diminishing returns to increasing the associativity. For example, the difference in reducing the miss rate is larger as we go from a direct-mapped cache to a cache with four ways per set than when we go from a cache with four ways per set to a cache with sixteen ways per set. However, increasing the associativity has drawbacks, most notably the energy and time required to access the L1 DC. Our approach can allow a processor to use a smaller, direct-mapped L1 DC with a smaller sub-block size without affecting the hit rate and also potentially decreasing the number of cycles for an L1 DC hit. As we vary the approach from 2xLS, 2xLS-ext, 4xLS, and 4xLS-ext, the miss rate improves. However, the benefits of reducing the miss rate must be compared against the cost of increasing the L1 DC size as well as the energy required to access it.

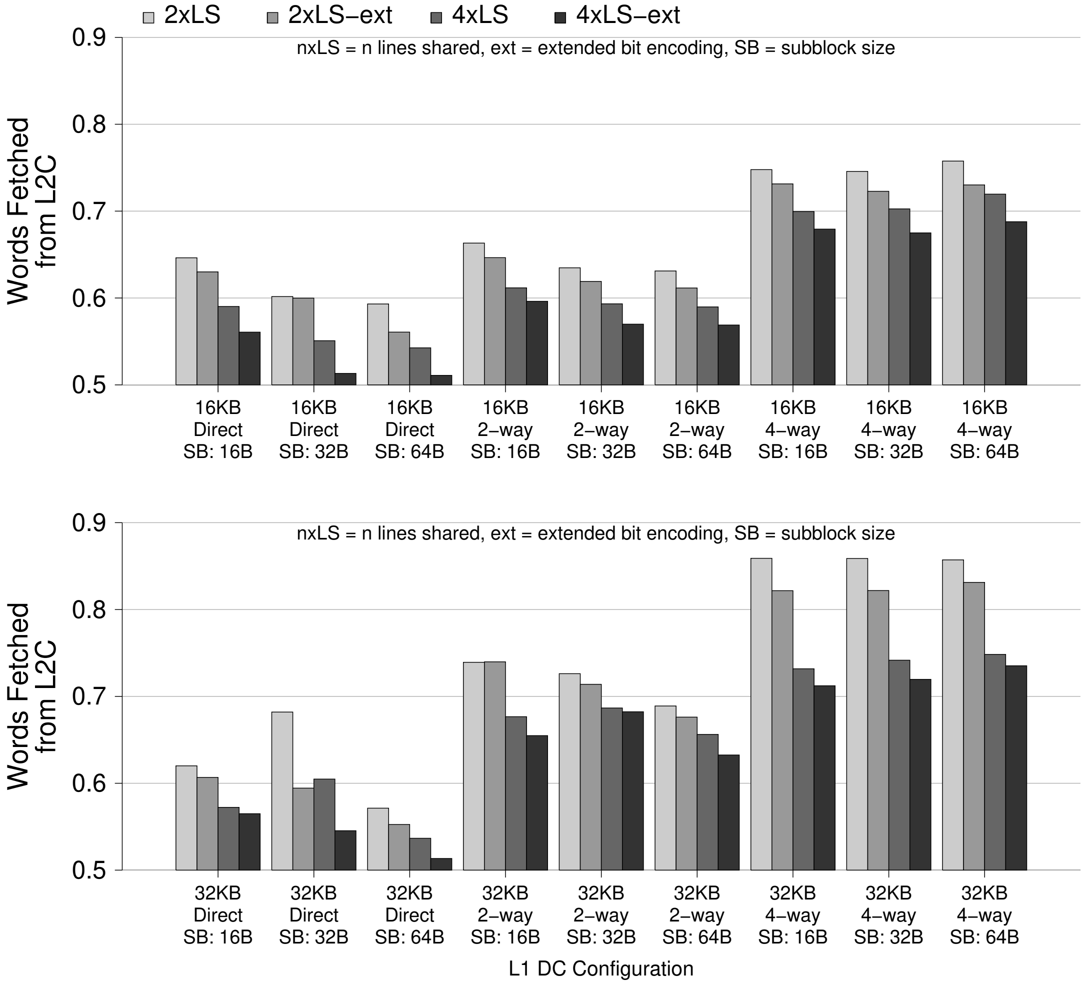


Figure 6.10: Words Fetched from L2

Figure 6.10 shows the the number of words fetched from the L2 and placed in the L1 DC relative to the baseline. As our approach can retain values longer, we don't need to fetch words from the L2 if their values still remain in the L1 DC at the time we are filling that line. In other words, upon an L1 DC word miss but tag hit, we only need to fill the words missing from the L1 DC meta-line. The word-fill rate tells us how much we reduce the words needed to be filled in lines and is measured by counting the number of words retrieved from the L2 and placed in the L1 DC compared to the baseline. Lowering the word-fill rate impacts performance, as L1 DC line fills don't take as long, as well as energy, as fewer words are fetched from the L2. As we increase the associativity of the cache, the number of words we fetch from the L2 approaches that of the baseline. For a 16KB L1 DC with a 64 byte sub-block size using 4xLS, the ratio of words retrieved from L2 increases from 54.2% to 72% as we increase the associativity from 1-way to 4-way. For a 16KB, direct-mapped L1 DC using 4xLS, the ratio of words that need to be retrieved from L2 increases from 59% to 54%. As we increase the size of the sub-block, the ratio of the words fetched from L2 relative to the baseline tends to decrease. This is because the baseline fetches fewer words in total for a smaller sub-block size. For a direct-mapped L1 DC using 2xLS, the number of words fetched from L2 decreases from 62% to 57% as we increase the sub-block size from 16 bytes to 64 bytes.

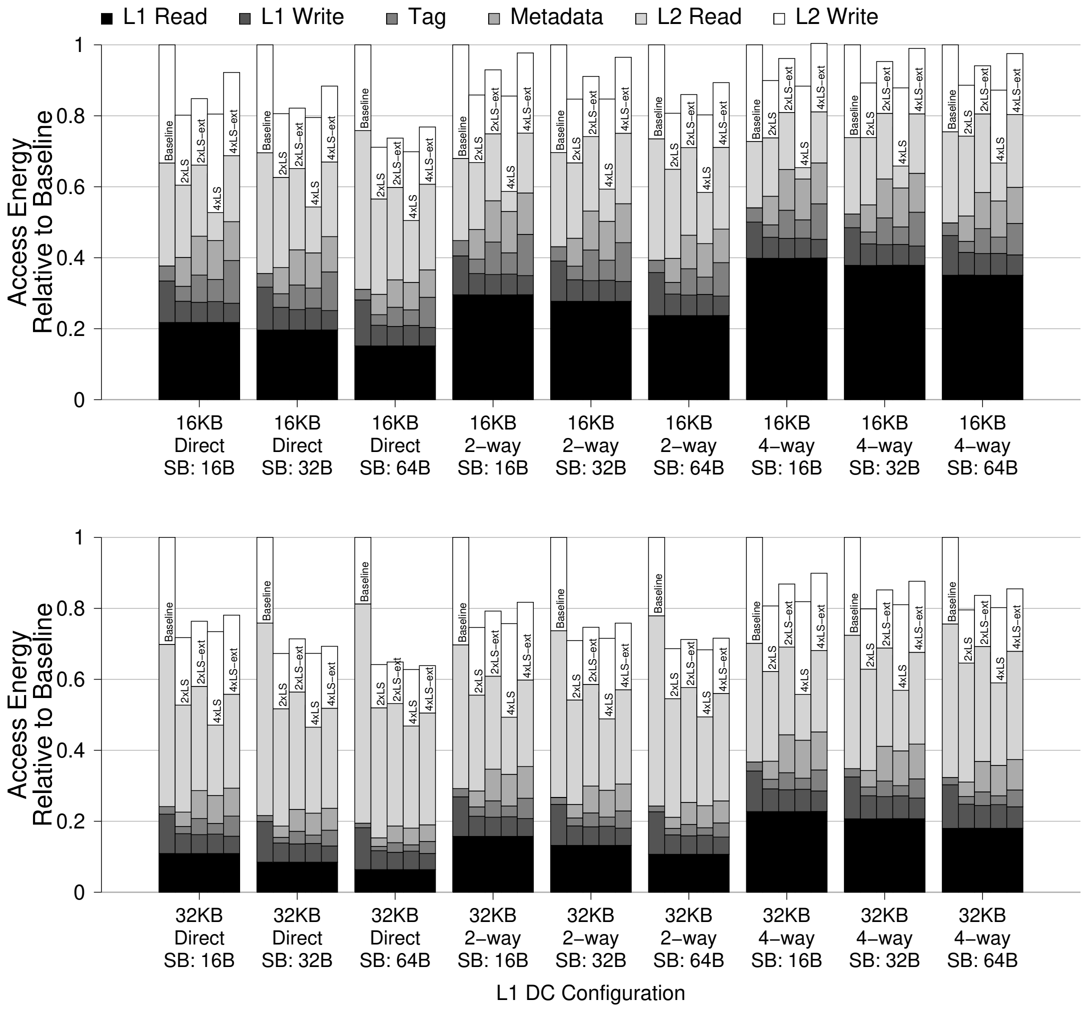


Figure 6.11: Breakdown of L1 DC Energy by Component

Figure 6.11 shows the energy consumption of our modified L1 DC relative to the baseline for various L1 DC configurations. As the size of the L1 DC decreases, the energy consumed by the L2 begins to dominate the total energy expenditure. As we decrease the energy expended by the L2 by decreasing the L1 DC miss rate as well as decreasing the number of words that need to be fetched from the L2, we are able to achieve significant energy savings (over 18% for 16KB/Direct/16B Sub-Block/4xLS). However, as we increase the size, associativity, and sub-block size of the L1 DC, the energy expended by the L1 DC dominates as there are fewer L1 DC misses and the energy to access the L1 DC increases. This overhead energy is offset by decreasing the energy expended by writing to the L1 DC, as we write only the bytes necessary to be able to retrieve the data using sign-extension and entirely avoiding writing to the L1 DC for zero values. The total energy expended approaches the (97.7%) for a 32KB, 4-way associative, 64B sub-block size L1 DC using 4xLS-ext. The most energy efficient of our approaches is 2xLS, as this doubles the size of the tag array and only requires 64 bits per cache line for the metadata array. The most expensive in terms of energy usage is 4xLS-ext, as this quadruples the size of the tag array and requires 192 bits per cache line for the metadata array (3 bits per word, 16 words per meta-line, 4 meta-lines per L1 DC line). All approaches, on average, decrease the energy usage relative to the baseline.

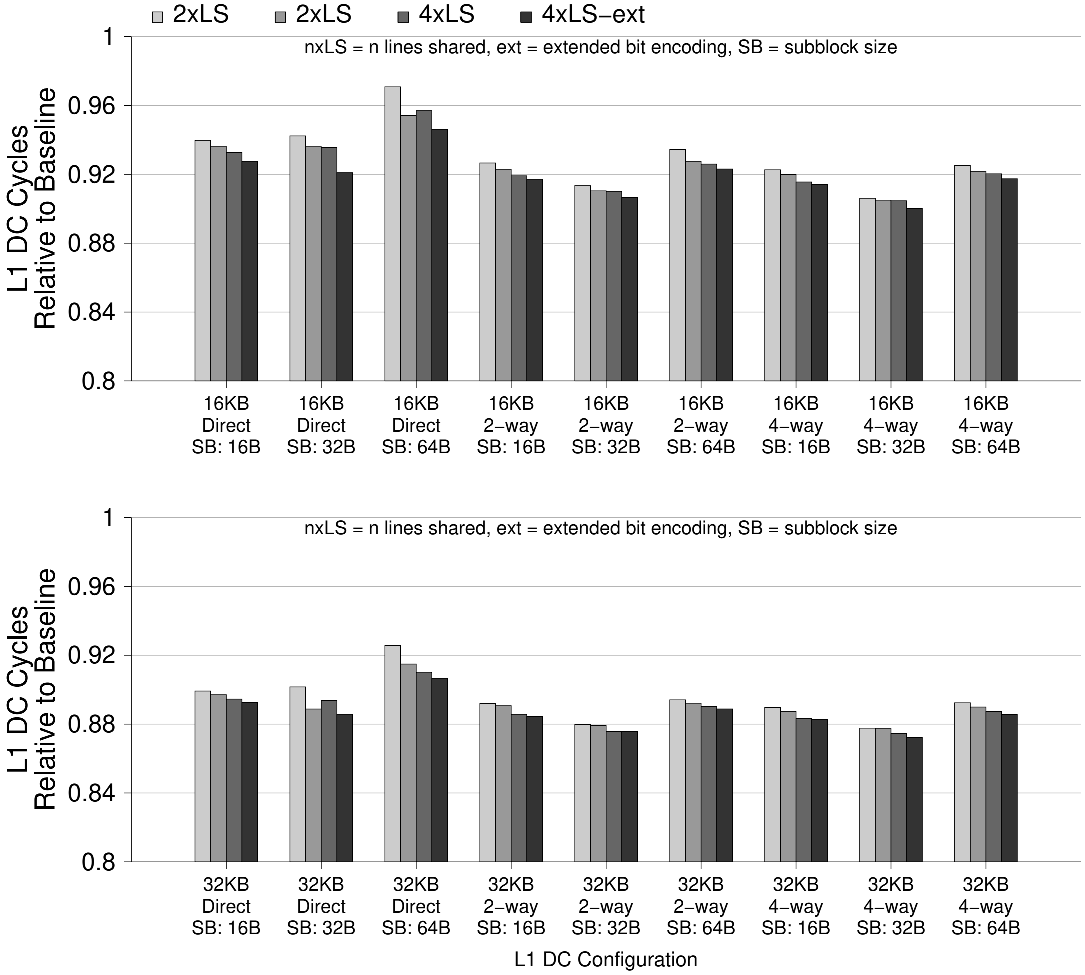
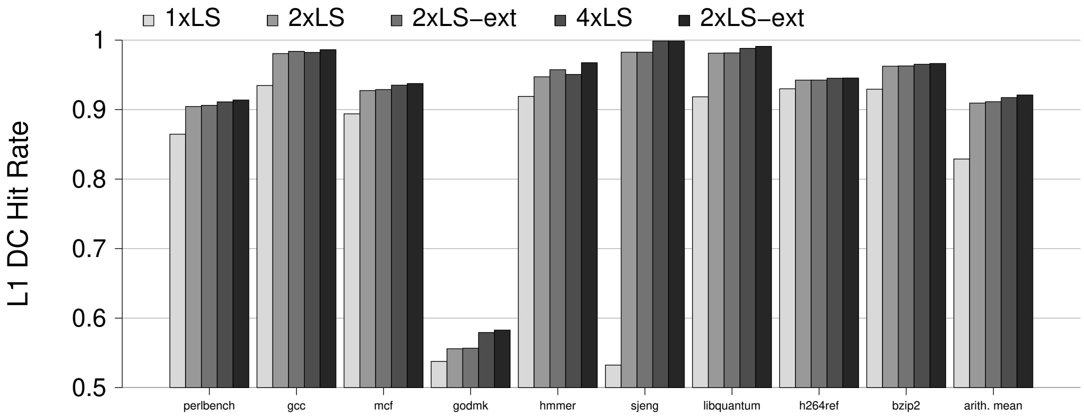


Figure 6.12: Performance Relative to Baseline

Figure 6.12 shows the number of cycles executed by the processor using our modified L1 DC designs relative to *a baseline 16KB, direct-mapped, 16B sub-blocked L1 DC*, **not** the baseline for a processor with the same size, associaitivty, and sub-block size. By using a constant baseline for all approaches, we can compare the performance between sizes and configurations. The performance is largely determined by the L1 DC miss rate as well as the number of words fetched from the L2. Decreasing the L1 DC miss rate reduces the number of load-delay hazards and decreasing the number of words fetched from the L2 decreases the number of cycles required to fill the L1 DC line. While the miss rates for a 16KB 4-way, 64B sub-block size L1 DC using 4xLS-ext is nearly identical to a baseline 32KB, 4-way, 64-B sub-block size L1 DC without our approach L1 DC (95.4%

versus 95.5%), the performance of the 32KB is still superior (roughly 2% fewer cycles relative to the baseline). This is the 16KB cache fetches more words from the L2, requiring more stalls due to L1 DC line fills. Even though the L1 DC uses critical-word first, which decreases the miss penalty for a load that misses in the L1 DC, a subsequent memory operation must stall if it tries to access the L1 DC while the rest of the line is being filled. From Figures 6.9 and 6.10, we see that the miss rate reduction increases and the number of words filled relative to the baseline decreases as we decrease the L1 DC size, associativity, and sub-block size.



16KB, 4-way set-associative Cache with 64B Sub-block

Figure 6.13: Hit Rate by Benchmark

Figure 6.13 shows the hit rates for individual benchmarks as we vary the approach for a 16KB, direct-mapped, 16-byte sub-block size L1 DC. On average, we improve the hit rate by 8.1%, 8.25%, 8.84%, and 9.2% as we vary the approach from 2xLS, 2xLS-ext, 4xLS, and 4xLS-ext, respectively. It should be noted that it's impossible for our approach to perform worse than the baseline. The largest improvement was for libquantum, which demonstrates nearly pathological cache behavior as it iterates sequentially through two arrays, causing constant thrashing for cache lines.

6.8 Related Work

The need for maintaining a fast access time has limited the exploration of data compression techniques in L1 DCs. This is because the overhead for loading and decompressing the data extends the critical path for memory accesses. In order to not increase the access time, the set index and line offset portions must remain unchanged during data decompression so as to not increase the time required to load the compressed data. In addition, data decompression must be fast enough that it does not increase the time required to forward the data to the next pipeline stage. We limit this section to first-level data cache compression techniques, which are most relevant to this paper.

[23] is a L1 DC compression technique that compresses data inside the L1 DC at the granularity of L1 DC lines. If each word in an L2 subline can be represented in two bytes using sign-extension, then the L2 subline is stored in only half of the corresponding L1 DC line. A single L1 DC line can be shared by two L2 sublines if all the values in each L2 subline can be represented in two bytes using sign extension. To increase the number of opportunities for compressing an entire L2 subline, the technique also allows an L2 subline to be compressed if it has only a small number of values that can't be stored in two bytes. In this case, the upper two bytes of these values are stored in a separate cache which is read in parallel to the data access. Each word in the L1 DC line has an additional extra storage bit indicating that the word's upper two bytes are in this additional half-word storage. As with our approach, [23] compresses data inside an L1 DC, increasing the amount of data relative to an uncompressed cache. Both approaches don't affect the access time for the L1 DC while also improving the hit rate. However, our approach compresses data inside an L1 DC line at the granularity of words rather than cache lines, which increases the number of opportunities available for data compression and thus increases the amount of data held inside the L1 DC. In addition, our approach also decreases the time required to fill L1 DC lines due to words often already being resident.

The compression cache (CC) is another L1 DC compression technique that compresses data inside the L1 DC at the granularity of L1 DC lines [32]. Each cache line can either hold one uncompressed line or two cache lines compressed to half their lengths. This is done by encoding values that appear frequently during program execution using only a few bits. If at least half the words can be represented using an encoding for a freqently-occuring value, then the line can be compressed to half its size and placed inside the L1 DC alongside another compressed line. 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.

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 [33]. 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 that a separate frequent value L1 DC data array was accessed first to avoid the regular L1 DC data array if the value was frequent [30]. This approach reduced L1 DC energy usage at the expense of delaying access to a nonfrequent value by a cycle.

Both [23] and [32] increase cache complexity as updating a value may require decompressing a compressed cache line. In [23], a value that could previously be stored in two-bytes could be overwritten with a 4-byte value, and, similarly, an encoded frequent value in [32] could be overwritten with a value with no encoding. In both cases, the cache line must be decompressed potentially causing an execution penalty. Our approach, however, updates the word in the cache and only needs to update the metadata corresponding to L2 subline values conflicting for the same word slot in the cache. In addition, the cache line size affects the amount of data that can be compressed for both [23] and [32], whereas the cache line size does not affect our approach. As the cache line size increases, the likelihood that at least half of the values in [23] can be stored in two bytes using sign extension or that at least half of the values are frequent values [32] and thus can be encoded using a shorter sequence drops. Finally, it appears that both [32] and [33] requires profile runs of programs in order to identify frequently occurring values and also increases L1 DC access time as the location of the data inside the cache can change depending on whether or not it belongs to a compressed cache line or decompressed cache line.

Dynamic zero compression adds a zero indicator bit (ZIB) to every byte in a cache [28]. 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 [12]. 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 [13]. 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 first-level cache technique that shares 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.

Way prediction techniques [11, 22] are now commonly used to predict which way of the L1 DC data array is being accessed and this prediction is verified by performing a DTLB access and an L1 DC tag comparison. Way prediction can both reduce energy usage (a single L1 DC tag array and a single L1 DC data array are accessed) and improve L1 DC load hit time (the requested data from one L1 DC data array can be sent to the CPU without going through a multiplexor that selects the data word based on which tag comparison is a hit). Way prediction imposes a performance penalty when the L1 DC way is incorrectly predicted. Newer versions of way prediction are more accurate, but require a custom SRAM implementation to mitigate the latency of accessing way prediction information before the regular L1 DC tag and data access by using a hash of the virtual address. A direct-mapped line-shared L1 DC can provide the energy and access time benefits of way prediction without its disadvantages. Way prediction could also be used in conjunction with an associative L1 DC that supports our line sharing approach.

6.9 Future Work

The fragmentation problem occurs when not all of the block that is filled into a cache is used before it is evicted. Subblocking is sometimes used to address the fragmentation problem with large cache lines to reduce the fill time requiring that a subblock is only fetched when accessed and not resident. Line sharing provides additional benefits for subblocked L1 DCs. Much of the space available in a conventional subblocked L1 DC line may go unused as subblocks in the line will not be filled when they are not referenced before the line is evicted. Allowing multiple L2 sublines to share the same L1 DC line can make better use of this available space in the presence of subblocking. As long as distinct L2 sublines refer to subblocks in different portions of the L1 DC line, the L1 DC can simultaneously hold entire subblocks from different L2 sublines even without compressing values.

Thus, we are considering using subblocking with line sharing and experiment with different line sizes. With a small subblock size, there are more opportunities for line sharing as L2 sublines values won't compete for words in the L1 DC line if they don't belong to the same referenced subblock. With a larger subblock size, line sharing can overcome some of the disadvantages of subblocking such as a higher miss rate and also reduce the size of the L1 DC tag memory.

There are other L1 DC parameters or techniques that would be interesting to investigate to determine the impact of line sharing. A larger line size may be beneficial to reduce L1 DC tag storage and not suffer as much from the fragmentation problem with line sharing. Prefetching of lines when using line sharing may be more beneficial as fewer data words may be evicted.

6.10 Conclusions

In this paper we described an L1 DC line sharing technique where multiple L2 sublines can be shared in each L1 DC line. The technique shares data at the granularity of individual words within each L1 DC line and uses a tag for each L2 subline to determine if it is resident and metadata for each word to determine if the word is resident and how the word can be extracted. Metaline fill and replacement policies were presented to decrease the amount of metaline thrashing. The results showed reductions in L1 DC miss rates and words fetched from the L2C. These reductions positively impacted both performance and energy usage.

L1 DC line sharing is a relatively simple technique to implement with minimal impact on the L1 DC access time. L1 DC line sharing was shown to be beneficial over a variety of configurations with the most benefit achieved when used with simpler (less associativity) and smaller caches. The benefits in performance and energy usage make L1 DC line sharing applicable to a variety of types of processors.

CHAPTER 7

CONCLUSIONS

This dissertation has provided techniques that both reduce the energy usage of first-level data accesses as well as improves performance by decreasing the number of stall cycles due to load and store hazards. We studied four different techniques that help to achieve these goals.

Chapter 3 proposed a design that memoizes L1 DC access information associated with the register used to access the data. This allows subsequent memory accesses using the same register to perform direct, non-speculative accesses to set-associative data caches. We also propose a small ALU modification to preserve this access information during register updates if the cache line the register points to doesn't change. Furthermore, this chapter also proposed the *DCAS Refresh Buffer* to retain L1 DC access information longer. This approach doesn't require changes to binaries or extensive pipeline modifications. By using a non-speculative approach to accessing a set-associative cache, the DTLB access as well as the L1 DC tag array access can be avoided during the L1 DC access. In addition, only a single way of a set is accessed during loads, avoiding accesses to the remaining *n-1* ways. The approach presented in this chapter provides significant energy savings over traditional, set-associative L1 DCs. Using this approach alone failed to achieve energy saving benefits over way prediction or way caching. This is because way prediction and way caching achieve very high hit rates by accessing a small structure between the time of the effective address calculation and accessing memory, which is not always feasible to access without increasing the cycle time. However, this chapter also showed that this approach is complementary to way prediction, as our non-speculative approach can be used to avoid the DTLB, tag array, and a speculative direct L1 DC access when possible and using way prediction's speculative approach otherwise, which achieved the largest energy savings.

Chapter 4 proposed a method that allowed a processor to achieve the energy-saving benefits of a L0 DC while *improving* performance. This design introduced an additional structure, the *base register structure*, which holds the base register value used during loads and stores. With this structure, the base register value for a load or store can be accessed during the instruction decode stage to calculate the effective address during the register fetch stage. Since the effective address is available one cycle before the memory access stage, we can access the L0 DC one cycle before the memory access stage. If the access is a hit, then the data is available one cycle earlier, possibly avoiding load-delay hazards. If the access is a miss, we can still access memory in the same stage we normally would, avoiding the performance penalty typically associated with L0 DCs. In addition, we also introduced the *load-store vector* as well as the *base register-index structure* to reduce the number of reads and writes to the base register structure, greatly reducing its energy consumption.

Chapter 5 presented a design for an L0 DC that can both reduce data access energy and eliminate the performance penalty normally associated with an L0 DC. We showed that a *word-filled* L0 DC can achieve significant energy saving benefits over a *line-filled* L0 DC. This is because fetching a L0 DC line in a single cycle from the L1 DC consumes significantly more energy than fetching a single word. This increase in L1 DC access energy outweighs the benefits of reducing the L0 DC miss rate. This chapter also proposed a method of improving the hit rate of L0 DCs by allowing multiple L1 DC (sub)lines to share a single L0 DC line, called *line sharing.* If the words of two or more L1 DC (sub)lines refer to different portions of an L0 DC line, then the values don't conflict and can be stored in the L0 DC line simultaneously. Chapter 5 also proposed a method that potentially allows values belonging to two different L1 DC (sub)lines referring to the same word of an L0 DC (sub)line to be stored simultaneously if both values taken together can fit within a single word, called *data packing.* Using a word-filled L0 DC with line sharing and data packing, this thesis achieved significant energy saving benefits by greatly reducing the miss rate of a L0 DC. Finally, Chapter 5 also introduced a method of removing the performance penalty typically associated with L0 DCs. This was done by speculatively accessing the metadata for the L0 DC line during the ALU stage of the processor using the base register value. If the metadata access using the base register value shows that the data is available and the cache line associated with the base register value and the effective address are the same, we guarantee that the L0 DC access will be a hit, removing any potential performance penalty. This comes at the cost of reducing the L0 DC service rate, defined to be the number of loads that retrieve their data from the L0 DC, as only loads with small displacements will be able to use the L0 DC.

Chapter 6 uses the line sharing and data packing method as described in Chapter 5 for the L1 DC. The approach is evaluated using direct-mapped as well as set-associative L1 DCs in order to provide a more thorough analysis. In addition, sub-blocked L1 DCs are also evaluated as they can see additional benefits when using line sharing as L2 (sub)lines can share different sub-blocks of the same L1 DC line. Finally, Chapter 6 extensively evaluates possible data compression techniques by adding four additional categories for compressing data values. While there are few benefits to extending the compression categories as well as increasing the number of L2 (sub)lines that map to a single L1 DC line past two, this paper shows that allowing two L2 (sub)lines to share a single L1 DC line provides significant benefits with very little overhead in the space required. These benefits include increasing the performance as well as decreasing the energy consumption of the memory hierarchy. The performance benefits achieved were due to decreasing the number of L1 DC misses as well as decreasing the miss cycle penalty by decreasing the number of words that need to be fetched from the L2 cache. The energy saving benefits come from fewer L2 cache reads and writes, as well as decreasing the energy consumed during L1 DC writes as only the bytes necessary for decompressing the value are written to the L1 DC.

The techniques laid out in this dissertation are applicable to a wide range of processors. None of the four techniques shown in Chapters 3-6 degrade performance while Chapters 4 and 6 discuss techniques that can be used to improve the performance of a processor. Furthermore, all of these techniques can be used to reduce the energy consumed by a processor. Taken together, the techniques discussed in this dissertation provide realistic methods to improve the memory hierarchy by reducing its energy usage as well as by reducing the number of stalls due to load and store hazards.

BIBLIOGRAPHY

[1] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining data cache access with fast address calculation. In *Proc. Int. Symp. on Computer Architecture*, pages 369–380, New York, NY, USA, June 1995. ACM.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proc. Int. Symp. on Microarchitecture*, pages 82–92. ACM/IEEE, 1995.

[3] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. Designing a practical data filter cache to improve both energy efficiency and performance. *ACM Transactions on Architecture and Compiler Optimizations (TACO)*, 10(4):54:1–54:25, December 2013.

[4] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. Speculative tag access for reduced energy dissipation in set-associative l1 data caches. In *Proceedings of the IEEE International Conference on Computer Design (ICCD 2013)*, October 2013.

[5] A. Basu, M. Hill, and M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, pages 297–308, June 2012.

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

[7] N. Duong, T. Kim, D. Zhao, and A. Veidenbaum. Revisiting level-0 caches in embedded processors. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 171–180, New York, NY, USA, October 2012. ACM.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Int. Workshop on Workload Characterization*, pages 3–14, December 2001.

[9] S. Hines, P. Gavin, Y. Peress, D. Whalley, and G. Tyson. 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*, pages 119–128, June 2009.

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

[11] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proc. IEEE Int. Symp. on Low Power Design (ISLPED)*, pages 273–275, August 1999.

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

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

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

[15] J. Kin, M. Gupta, and W.H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proc. Int. Symp. on Microarchitecture*, pages 184–193, December 1997.

[16] J. Lee and S. Kim. Filter data cache: An energy-efficient small l0 data cache architecture driven by miss cost reduction. *IEEE Trans. Computers*, 64(7):1927–1939, July 2015.

[17] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. pages 469–480, December 2009.

[18] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. pages 694–701, 2011.

[19] R. Megalingam, K. Deepu, I Joseph, and V. Vandana. Phased set associative cache design for reduced power consumption. In *Proceedings of International Conference on Computer Science and Information Technology*, pages 551–556, 2009.

[20] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero. Fast speculative address generation and way caching for reducing l1 data cache energy. In *Proceedings of International Conference on Computer Design*, October 2007.

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

[22] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proc. ACM/IEEE Int. Symp. on Microarchitecture (MICRO)*, pages 54–65, December 2001.

[23] Prateek Pujara and Aneesh. Restrictive compression techniques to increase level 1 cache capacity. In *International Conference on Computer Design*, pages 327–333, 2005.

[24] A. Sembrant, E. Hagersten, and D. Black-Shaffer. Tlc: A tag-less cache for reducing dynamic first level cache energy. In *Proc. 46th ACM/IEEE Int. Symp. on Microarchitecture (MICRO)*, pages 351–356, December 2013.

[25] Michael Stokes, Ryan Baird, Zhaoxiang Jin, David Whalley, and Soner Önder. Decoupling address generation from loads and stores to improve data access energy efficiency. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2018, pages 65–75, New York, NY, USA, 2018. ACM.

[26] C. Su and A Despain. Cache design trade-offs for power and performance optimization: A case study. In *Proc. Int. Symp. on Low Power Design (ISLPED)*, pages 63–68, 1995.

[27] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proc. Int. Conf. on Computer Design*, pages 68–73, Washington, DC, USA, 2001. IEEE Computer Society.

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

[29] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanović. Direct addressed caches for reduced power consumption. In *Proc. 34th ACM/IEEE Int. Symp. on Microarchitecture (MICRO)*, pages 124–133, December 2001.

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

[31] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. *ACM Transactions on Architecture and Compiler Optimizations (TACO)*, 2(1):34–54, March 2005.

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

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

[34] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Tag check elision. In *International Symposium on Low Power Electronics and Design*, pages 351–356, New York, NY, USA, 2014. ACM.

BIOGRAPHICAL SKETCH

I was born in Miami. I attended Hialeah High School and went on to Miami Dade College for my Associates in Arts and then finished my undergraduate at Florida State University. There, I also obtained my Masters in Computer Science and then pursued my doctorate in Computer Science.