Improving Energy Efficiency by Memoizing Data Access Information

Michael Stokes[1], Ryan Baird[1], Zhaoxiang Jin[2], David Whalley[1], and Soner Onder[2]

[1]Computer Science Department, Florida State University, Tallahassee, FL, USA
{mstokes, baird, whalley}@cs.fsu.edu
[2]Computer Science Department, Michigan Technological University, Houghton, MI, USA
{zjin3, soner}@mtu.edu

***Abstract*—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. We propose 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 memory address, we detect whether the calculated address shares the cache line memoized with the base register. If so, we avoid the L1 DC tag array access and the DTLB access to determine the L1 DC way and instead use the memoized information. 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. Our nonspeculative memoization approach can be applied before a speculative approach, allowing a significant reduction in data access energy usage for existing executables with no ISA modifications.**

***Index Terms*—Caches, Data Translation Lookaside Buffers**

I. Introduction

Level-one data cache (L1 DC) and data translation lookaside buffer (DTLB) 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 [1]. 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 [5]. 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* cache [5]. 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 cache 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.

Contemporary architectures designed using RISC principles attempt to implement each instruction using a single micro-operation (μop). However, memory operations involve many hidden μops performed by the hardware that not only form dependence chains, but can also use a significant amount of energy. 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 (*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 field and the indicated *way* to access the cache data and write the value into the destination register.

978-1-7281-2954-9/19/$31.00 ©2019 IEEE

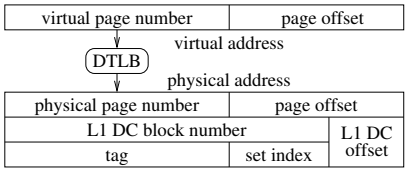


Fig. 1: Address Fields

These μops form dependence chains that can increase the latency of memory operations. 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) to be performed in parallel by accessing all ways of data in the L1 DC set at the expense of significant energy usage.

We propose the *Data Cache Access Memoization* (DCAM) technique to retain data access information so that subsequent memory accesses dereferencing the same register avoid performing redundant μops. These simple memoization techniques often avoid the DTLB access and L1 DC tag check and directly access a single L1 DC data array.

II. Background

We describe our proposed techniques in the context of an in-order pipeline where the benefits are more obvious. However, our proposed techniques to avoid L1 DC associative data accesses, L1 DC tag checks, and DTLB accesses could also be adapted for out-of-order (OoO) processors.

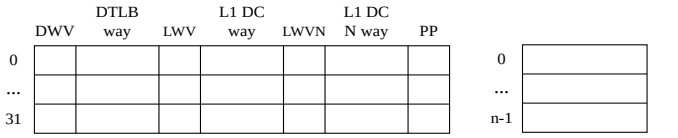
Figure 1 shows the address fields used to access the DTLB and the L1 DC. [1] The *virtual page number* is used to access

[1]In this figure we depict the *physical page number* and the *tag* fields being the same size, which is often the case for many processors, but the *physical page number* field could be smaller for a VIPT cache. To simplify the description, we assume these two fields are the same size in the paper.

```
r6=...;                 r20=...
...                   L3:r2=M[r20];
...=M[r6];              ...
...                     r20=r20+4;
M[r6]=...;              PC=r20!=r21,L3;
```

(a) Redundant Accesses (b) Strided Accesses

Fig. 2: Memoization Examples



(a) Data Cache Access Structure (DCAS) (b) DCAS Valid Info (DCAV)

Fig. 3: Data Cache Access Information

the DTLB to produce the corresponding *physical page number*. The *page offset* remains the same. The *L1 DC block number* uniquely identifies the L1 DC line. The *L1 DC offset* indicates the first byte of the data 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.

To load a value from an *n*-way set associative L1 DC the virtual memory address is generated by adding a displacement to a base address in an address generation stage. The displacement is a sign-extended immediate and the base address is obtained from the register file. In the L1 DC access stage the data translation lookaside buffer (DTLB), the L1 DC tag memory, and the L1 DC data memory can all be 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. 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.

III. 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 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 2(b) shows an example of accessing sequential array locations, where an L1 DC line is likely to be repeatedly accessed.

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. We track 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.

We associate L1 DC access information with the source register number of a load or store instruction and detect when updates to this register do not invalidate this information. Consider the data cache access structure (DCAS) in Figure 3(a) that contains fields associated with each integer register used as a base register in load and store instructions. The *DWV* (DTLB Way Valid) 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* (L1 DC Way Valid) 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. The *LWVN* (L1DC Way Valid Next sequential) 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 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* (Page Protection) 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(b) depicts the DCAV structure used to invalidate DCAS entries 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. Each time a DCAS entry shown in Figure 3(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(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.

IV. 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 4 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*

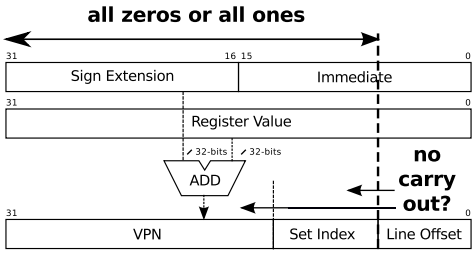


Fig. 4: Detecting Address Changes

```
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
```

Fig. 5: DCAS Refresh Example

number as shown in Figure 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, we 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.

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.

V. The DCAS Refresh Buffer

Frequently, a DCAS entry is invalidated but its contents continue to point to the correct cache line. The load instruction in Figure 5 sets DCAS entry 20, as shown in Figure 6(a), and is overwritten during the function call to foo. During foo's epilogue code, `r20`'s value is restored, again pointing to the same cache line in its DCAS entry. If we 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 we can restore the DCAS entry contents.

To accomplish this, we store 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

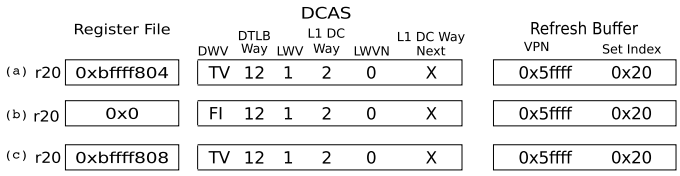


Fig. 6: DCAS Refresh Buffer Example

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 we compare the virtual tag and set index portions of the base register with the virtual tag and set index portions stored alongside the DCAS entry. If they match, then we 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 we 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 5, the DCAS contents still refers to the same DTLB way and L1 DC way shown in Figure 6(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 (set the DWV field to *true valid*) as shown in Figure 6(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.

VI. Evaluation Framework

In this section we describe the experimental environment. We use 17 benchmarks from the MiBench benchmark suite [2], which is a representative set of embedded applications. All benchmarks are simulated using the large dataset option and compiled using gcc with the *-03* option.

We used the ADL simulator [8] to simulate both a conventional MIPS processor as the baseline and the modified

processor as described in this paper. ADL 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 as shown in Table I. Table II shows other details regarding the processor configuration we utilized in our simulations. Note we separate the *DWV* bit from the rest of the DCAS structure and access this bit during the RF (register fetch) pipeline stage, which allows us to avoid accessing the rest of the DCAS structure when the *DWV* bit is not set.

TABLE I: DCAM Pipeline Stages

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

TABLE II: 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 III: Energy for L1 DC and DTLB Components

Component	**Energy**
Read L1 DC Tags - All Ways	0.495 pJ
Read L1 DC Data - All Ways	5.860 pJ
Read L1 DC Tag - One Way	0.124 pJ
Write L1 DC Data - One Way	2.730 pJ
Read L1 DC Data - One Way	1.369 pJ
Read DTLB - Fully Associative	1.240 pJ
Read DTLB - One Way	0.067 pJ
Read DCAS - 1 Entry	0.028 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

We used CACTI to estimate L1 DC and DTLB energy usage assuming 22-nm CMOS process technology with low standby power (LSTP) cells. Table III shows the energy required for accessing the various components. Leakage energy was gathered assuming a 1 GHZ clock rate.

VII. Results

Figure 7 shows the ratio of L1 DC data array load accesses that are direct (single L1 DC way) or set associative (all L1 DC ways). About 63% of the loads 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 data is updated.

Figure 8 shows the ratio of tag checks and DTLB accesses that remain after applying the DCAM technique. On average about 65% of the L1 DC tag checks are eliminated and about 71% of the fully associative DTLB accesses are eliminated.

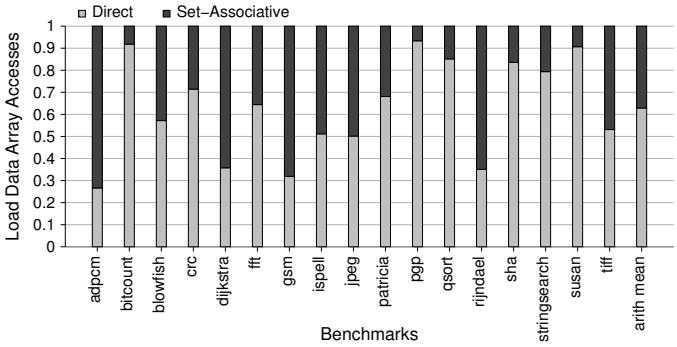


Fig. 7: L1 DC Data Array Load Accesses

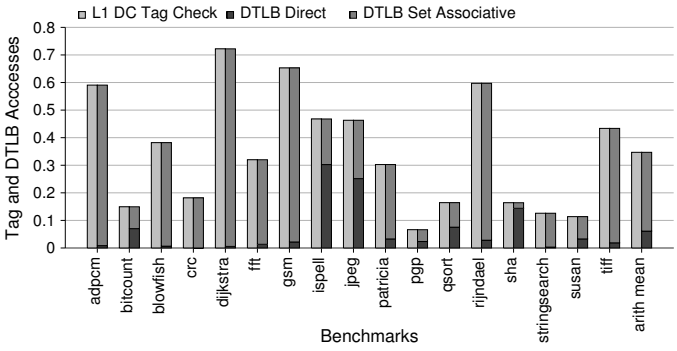


Fig. 8: Remaining DTLB and Tag Checks

About 6% of the original DTLB accesses are now just accessing a single way of the DTLB, which occurs when the *set index* field is updated, 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 III. On average about 6.6% of these avoided L1 DC tag checks are due to memoizing the next sequential line and 9.4% are due utilizing the DCAS refresh buffer.

Figure 9 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 our DCAM technique. On average about 0.7% of the energy is due to leakage. For the average baseline energy, 59.6% is due to data array reads from loads, 13.5% is due to data array writes from stores, 7.5% is due to L1 DC tag checks, and 18.7% is due to DTLB accesses. DCAM reduces the energy on average for data array reads to 31.4%, L1 DC tag checks to 2.6%, and DTLB accesses to 5.3%. Note that 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 55.1% of the baseline on average. The overall data access energy savings ranges from 71.1% for the *susan* benchmark to 20.8% for the *adpcm* benchmark. These energy reductions are significant given that these benefits are obtained on existing binaries with no ISA changes.

We simulated other L1 DC energy usage reducing techniques and used CACTI to estimate their energy usage (Figure 10). Using DCAM alone (55.1%) does worse than way

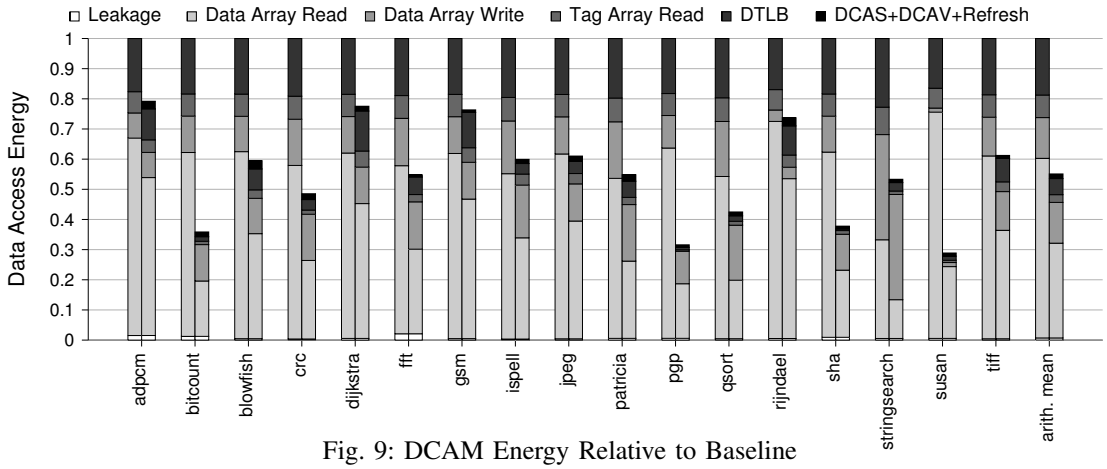


Fig. 9: DCAM Energy Relative to Baseline

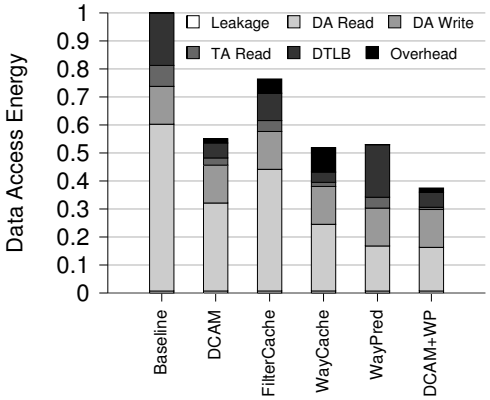


Fig. 10: Comparison of Energy Techniques

prediction (53.0%) and way caching (51.9%). Way caching fails when the address generation affects the L1 DC tag bits as we optimistically assumed it uses virtual tags to avoid DTLB accesses and L1 DC index bits are computed before the way cache tag comparison. DCAM in combination with way prediction achieves the best results (37.5%) 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 higher cost in overhead energy than DCAM, such as way caching. The disadvantages of other evaluated techniques are described in Section VIII.

VIII. 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 IV.

Way prediction (WP) predicts the data cache way to be accessed and performs a L1 DC tag check and DTLB access (TD) to verify this prediction. Unlike our DCAM approach, WP can have a performance penalty of several percent [3], [9] (OM). Newer WP versions are more accurate, but require a custom SRAM implementation (CS) to mitigate accessing WP information before the L1 DC. Nicolaescu et al. propose to save the L1 DC way of the last 16 cache lines in a table (WC) and perform a fully associative tag search on this table during address generation (CP). If there is a match, then only the corresponding way is activated [7]. In contrast, DCAM structures are much less expensive to access. Way halting (WH) is another method for reducing the number of tag comparisons [13], where partial tags are stored in a fully associative memory (the halt tag array) with as many ways as there are cache sets. 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, which halts access to ways that cannot contain the data. WH requires a specialized SRAM implementation that might have a negative impact on the maximum operational frequency (CS). A speculative way halting approach has been proposed that avoids these problems [6]. WP and WH could be combined with our DCAM approach to reduce energy usage even further (COM).

A tagless cache (TLC) design has been proposed that uses an extended TLB (ETLB) to avoid tag checks [10]. While

TABLE IV: 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	X
WC			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			

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 [12]. 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 [4]. 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 our DCAM approach [14]. 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 IV, 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 [11].

IX. Conclusions

We have described an approach 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 we are often able to avoid L1 DC tag array accesses and DTLB accesses, and access a single L1 DC data array for loads. Furthermore, we show a technique to retain this information across pointer updates if the updated value falls within the same cache line or page of the source register. We were able to obtain these energy benefits on unmodified binaries.

X. Acknowledgements

We thank the anonymous reviewers for their comments. This work was supported in part by the US National Science Foundation (NSF) under grants DUE-1259462, CCF-1533828, CCF-1533846, DGE-1565215, DRL-1640039, CRI-1822737, CCF-1823398, and CCF-1823417. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of the NSF.

References

[1] 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.

[2] 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, Dec. 2001.

[3] K. Inoue, T. Ishihara, and K. 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, Aug. 1999.

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

[5] 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.

[6] D. Moreau, A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. Practical way halting by speculatively accessing halt tags. In *Proceedings of the IEEE Design, Automation, and Test in Europe (DATE 2016)*, Mar. 2016.

[7] 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*, Oct. 2007.

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

[9] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. 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, Dec. 2001.

[10] 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, Dec. 2013.

[11] M. Stokes, R. Baird, Z. Jin, D. Whalley, and S. Onder. 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.

[12] 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.

[13] 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, Mar. 2005.

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