Improving Data Access Efficiency by Using Context-Aware Loads and Stores

Alen Bardizbanyan
Chalmers University of Technology
Gothenburg, Sweden
alenb@chalmers.se

Magnus Själander
Uppsala University
Uppsala, Sweden
magnus.sjalander@it.uu.se

David Whalley
Florida State University
Tallahassee, Florida, USA
whalley@cs.fsu.edu

Per Larsson-Edefors
Chalmers University of Technology
Gothenburg, Sweden
perla@chalmers.se

Abstract

Memory operations have a significant impact on both performance and energy usage even when an access hits in the level-one data cache (L1 DC). Load instructions in particular affect performance as they frequently result in stalls since the register to be loaded is often referenced before the data is available in the pipeline. L1 DC accesses also impact energy usage as they typically require significantly more energy than a register file access. Despite their impact on performance and energy usage, L1 DC accesses on most processors are performed in a general fashion without regard to the context in which the load or store operation is performed. We describe a set of techniques where the compiler enhances load and store instructions so that they can be executed with fewer stalls and/or enable the L1 DC to be accessed in a more energy-efficient manner. We show that using these techniques can simultaneously achieve a 6% gain in performance and a 43% reduction in L1 DC energy usage.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors- compilers, optimization

General Terms Algorithms, Measurements, Performance.

Keywords Energy, Data Caches, Compiler Optimizations.

1. Introduction

Certain CPU operations, such as loads, are more critical than other operations as they may have a greater impact on both performance and energy usage. Even when load instructions result in level-one data cache (L1 DC) hits, they still often cause pipeline stalls due to the loaded register being referenced before the data value is available. An L1 DC access also requires significantly more energy than a register file access since the L1 DC is much larger, is typically set-associative requiring access to multiple ways, and requires both a tag check and data access.

Conventionally a CPU performs a load or a store operation in the same manner each time the operation is executed. However, the most efficient method for performing a memory operation depends on the context in which it is executed. Thus, we propose a hardware/software co-design that provides contextual information for load and store operations so they can be more efficiently executed with respect to both performance and energy usage. We introduce a set of techniques which combines context-aware code generation with new L1 DC access schemes and auxiliary structures so that the execution of load and store instructions can take advantage of the context in which they are executed. The contribution of this paper is that it demonstrates that conventional load and store operations can be enhanced based on information of their context. These enhancements require only minor instruction set and architectural changes and can in exchange bring about substantial improvements in terms of energy usage and performance.

2. Instruction Contexts

A typical CPU performs all data accesses in a general fashion. However, the context of a data access can affect how the L1 DC can be most efficiently accessed. Relevant parameters for the data access context include the size of the displacement from the base register, the distance in instructions from a load to the first use of the register, and whether or not the memory access is strided. Here we describe how the context of a data access can be used to improve the performance and energy usage of load and store operations.

Address Displacement Context: Conventional CPUs that use a displacement addressing mode always perform the address calculations the same way, by adding a displacement to a base register. For large displacements, the traditional approach should be used, where the effective address is needed to ensure that the correct L1 DC tags and data are accessed. For smaller displacements, however, it is possible to access data more efficiently. We have previously shown that it is beneficial to speculatively access the tag arrays when the magnitude of the displacement is less than half the L1 DC line size [1]. The address calculation is unnecessary when the displacement is zero. During compilation we can detect the displacement size of load and store operations and use this information to access the L1 DC earlier in the pipeline when possible.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LCTES'15, June 18–19, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3257-6//...$15.00
http://dx.doi.org/10.1145/2670529.2754960

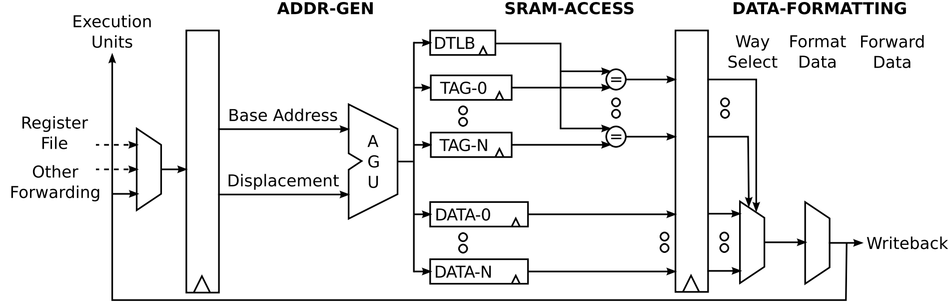

Figure 1: Conventional L1 DC Pipeline Access

Use-Distance Context: Conventional processors utilize set-associative caches to improve hit rate. For loads, the tags and data of all ways are accessed in parallel to reduce the number of stall cycles, as subsequent dependent instructions otherwise would have to wait longer for the loaded value. The disadvantage with this parallel access is that it wastes energy as all data arrays are read, even though the data can only reside in at most one array. In many cases the distance (in instructions) between a load instruction and the first use of the loaded value is long enough for the tags and data to be accessed in a serial (also called phased) manner, which enables loads to only access a single data array. During compilation we can detect the distance from a load instruction to the first use of the loaded value. We can use this distance to determine if the L1 DC can be serially accessed without incurring a stall with a subsequent instruction.

Strided Access Context: There are situations in which the next value to be accessed from the L1 DC is likely to be in the same line as the last time that same load or store instruction was executed. Strided load and store accesses with a stride shorter than half the L1 DC line size can be detected during compilation. This stride information can be stored in a small structure to avoid L1 DC tag checks and to more efficiently access the data.

For the three contexts above, the optimizations can be combined, e.g., by speculatively accessing the tags early when the displacement is small and by serially accessing the tags and data when no stalls will be introduced.

3. HW Enhancements for Loads and Stores

In this section we describe our proposed hardware enhancement for exploiting the context in which a data access is performed to improve the performance and energy usage of load and store operations. While these enhancements are applied within an in-order pipeline where the performance benefits are more obvious, these same enhancements could also be potentially beneficial to an out-of-order processor pipeline.

3.1 Conventional L1 DC Accesses

Figure 1 shows how a conventional *N*-way associative L1 DC access is pipelined for a load operation. The virtual memory address is generated by adding a displacement to a base address, where the displacement is a sign-extended immediate and the base address is obtained from the register file. In the first stage of the L1 DC access, the data translation lookaside buffer (DTLB), the tag memory, and the data memory are accessed in parallel and the tag from the tag memory is compared to the tag that is part of the physical page number from the DTLB. The next stage selects the data based on which of the *N* tags is a match, the retrieved data is formatted (sign or zero extended) for byte and halfword loads, and the data is forwarded to the appropriate unit.

3.2 Basic Definitions

Table 1 describes the various pipeline actions that are taken to perform a memory access as defined in this paper. The address generation (AG) is performed to calculate the effective address. The accesses to all tag arrays (TA) occur in parallel to determine the way in which the data resides. The accesses to all data arrays (DA) also occur in parallel for a load instruction to avoid pipeline stalls. A single data array (DA1) is accessed for a store, which occurs after the tag check has been performed. The first five actions are commonly performed when a load or store instruction is processed in a conventional CPU. The last two actions (TI and SD) access additional state and will be described later, in Section 3.4.1.

Table 1: Pipeline Actions for Data Access

Action	Explanation
AG	The sum of the base register and the immediate value is calculated to determine the address generation.
TA	All L1 DC tag arrays in the set-associative L1 DC are simultaneously accessed so that the tag checks can be performed.
DA	All L1 DC data arrays in the set-associative L1 DC are simultaneously accessed for a load operation.
DA1	A single L1 DC data array in the set-associative L1 DC is accessed for a store operation, which occurs during the cycle after the tag check.
DF	The loaded value is formatted (sign or zero extended) and forwarded to the execution unit.
TI	A tag and index are accessed from the strided access structure (SAS).
SD	Data are accessed from the strided access structure (SAS).

Figure 2 shows the fields of an address used to access the L1 DC. The *offset* indicates the first byte of the data to be accessed within the L1 DC line. The *index* is used to access the L1 DC set. The *tag* represents the remaining bits that are used to verify that the line resides in the L1 DC.

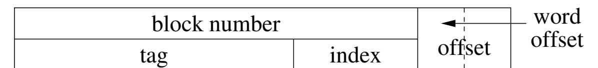

Figure 2: L1 DC Address Fields

Our compiler detects the cases shown in Table 2 for the size of the displacement. A *large* displacement indicates that a conventional address generation stage is required before the L1 DC tags can be accessed. A *small* displacement implies that the L1 DC tags can be speculatively accessed using the L1 DC index field from the base register value with a high likelihood of success since it is unlikely that adding the displacement will affect the L1 DC index field of the effective address. A *zero* displacement means that address generation altogether is unnecessary.

Table 3: Cases for Pipelining Non-Strided Load Accesses

Case	Pipeline Stage Actions				Requirements				Benefit
	EX	**DC1**	**DC2**	**DC3**	**Disp**	**Dep Inst**	**Stride**	**Example**	
L0	AG	TA+DA	DF		large	1-3	stride $> \frac{1}{2}$ line size || stride unknown	**r[2]**=M[r[4]+**76**]; r[3]=r[3]+**r[2]**;	
L1	TA+DA	DF			zero	1-2		**r[2]**=M[r[4]]; r[3]=r[3]+**r[2]**;	avoids stall
L2	TA AG+TA	DA1	DF		zero small	> 2 1-3		**r[2]**=M[r[4]+**4**]; r[3]=r[3]+**r[2]**;	1 DA
L3	AG	TA	DA1	DF	small || large	> 3		**r[2]**=M[r[4]+**4**]; *<3 or more insts>* r[3]=r[3]+**r[2]**;	1 DA without spec TA

Table 2: Memory Access Displacement Sizes

Size	Description
large	The displacement magnitude is greater than half the line size.
small	The magnitude of a nonzero displacement is less than or equal to half the L1 DC line size.
zero	The displacement is zero.

3.3 Enhancements for Non-Strided Accesses

We support four different ways of accessing the L1 DC for loads when the stride is either too large to exploit (greater than one half of the L1 DC line size) or is unknown. The four cases are depicted in Table 3. The EX stage is where a conventional integer operation is performed, the DC1 and DC2 stages represent the conventional first and second cycles to access the L1 DC, and the DC3 stage represents one additional cycle to access the L1 DC. In general, our highest priority is to avoid pipeline stalls and our secondary priority is to perform the memory access in the most energy-efficient fashion.

Case L0: The CPU accesses the L1 DC in the conventional manner for loads. The CPU accesses all the tag arrays (TA) and data arrays (DA) in the DC1 stage. In this default situation, the displacement (*Disp*) is large, which means a speculative tag access is not possible, and the loaded register is referenced in one of the following three instructions (*Dep Inst*). A one-cycle stall is avoided by accessing all the tag and data arrays in parallel. Note that the benefits of all the remaining cases for load instructions are expressed relative to this default case.

Case L1: The CPU accesses both the L1 DC tag arrays (TA) and data arrays (DA) in the EX stage. The requirements for this case are that 1) the displacement is zero and 2) the loaded register is referenced in the first or second instruction after the load. The CPU can access the L1 DC early using the address in the base register since the offset is zero making the address known before the EX stage is entered. The CPU accesses both the tag and data arrays in parallel, since the pipeline would stall an extra cycle if the L1 DC data array is accessed in the following cycle. Figure 3 shows an example of how *case L1* can avoid a stall cycle.

	1	2	3	4	5	6	7
`r[2]=M[r[4]];`	...	ID	AG	TA+DA	DF	...	
`r[3]=r[3]+r[2];`		...	ID	stall	stall	EX	...

➡

	1	2	3	4	5	6	7
`r[2]=M[r[4]];`	...	ID	TA+DA	DF	...		
`r[3]=r[3]+r[2];`		...	ID	stall	EX	...	

Figure 3: Example of Case L0 Replaced by Case L1

Case L2: The CPU accesses the tag arrays (TA) in the EX stage and a single data array (DA1) in the DC1 stage to reduce energy usage. One set of requirements for this case is that 1) the displacement is zero and 2) the loaded register is not used in the two following instructions. Here, there would be no execution time penalty and only a single data array (DA1) is accessed, which is in contrast to *case L1*. A different set of requirements for *case L2* is that 1) the displacement is small and 2) the loaded register is used in one of the following three instructions. With a small displacement the CPU can speculatively access the tag arrays in parallel with the address generation (AG) in the EX stage. However, the line offset is not known so the data arrays cannot be accessed in the EX stage. A single data array is therefore accessed in the DC1 stage, as depicted in Figure 4.

	1	2	3	4	5	6	7
`r[2]=M[r[4]+4];`	...	ID	AG	TA+DA	DF	...	
`r[3]=r[3]+r[2];`		...	ID	stall	stall	EX	...

➡

	1	2	3	4	5	6	7
`r[2]=M[r[4]+4];`	...	ID	AG+TA	DA1	DF	...	
`r[3]=r[3]+r[2];`		...	ID	stall	stall	EX	...

Figure 4: Example of Case L0 Replaced by Case L2

Case L3: The CPU accesses all the tag arrays (TA) in the DC1 stage and a single data array (DA1) in the DC2 stage, given that 1) the displacement is nonzero and 2) the loaded register is not used in the following three instructions. The CPU is guaranteed not to stall, since there are no dependent instructions in the pipeline, and energy usage is reduced due to the access of only a single data array. The advantage of *case L3* over *case L2* for small displacements is that there can be no speculative tag access failure.

We support two different ways of accessing the L1 DC for stores when the stride is either too large to exploit (greater than one half of the L1 DC line size) or is unknown. The two cases are depicted in Table 4. Note that all store operations are required to access the tag arrays (TA) before accessing the single data array (DA1) since the tag check must match for the data to be updated.

Case S0: The displacement is nonzero and the store operation accesses the tag arrays (TA) in the DC1 stage, which is how stores are conventionally processed.

Case S1: The displacement is zero, which enables the store operation to access the tag arrays (TA) in the EX stage, which may help to avoid subsequent structural hazards. Normally store operations are considered less critical than loads as a value stored to memory is not typically used in the instructions that immediately follow the store. Thus, speculative accesses to the tag arrays are not performed when the displacement is small since a speculation failure would require accessing the tag arrays a second time.

Table 4: Cases for Pipelining Non-Strided Store Accesses

Case	Pipeline Stage Actions				Requirements			Benefit
	EX	DC1	DC2	DC3	Disp	Stride	Example	
S0	AG	TA	DA1		not zero	stride > $\frac{1}{2}$ line size || stride unknown	M[r[4]+**4**]=r[2];	
S1	TA	DA1			zero		M[r[4]]=r[2];	1 cycle early

3.4 Enhancements for Strided Accesses

Strided accesses with a stride shorter than half the L1 DC line size have a high probability of accessing the same cache line more than once. The compiler can convey information regarding such strided accesses to the CPU. By retaining a small amount of additional state we can then directly access the cache line without any parallel tag or data accesses.

Figure 5 shows an example of a strided memory access that is followed by an increment to the base register in a loop. Figure 6 shows an example of a sequence of memory accesses whose addresses vary by a constant stride, which commonly occurs in the prologue or epilogue of a function for saving and restoring registers, respectively. The compiler can convey information regarding strided accesses to the CPU so that more efficient data accesses can occur by retaining a small amount of additional state.

```
L3: r[2]=M[r[4]];
    ...
    r[4]=r[4]+4;
    PC=r[4]!=r[5],L3;
```

Figure 5: Strided Load

```
...
r[22]=M[r[sp]+100];
r[21]=M[r[sp]+96];
r[20]=M[r[sp]+92];
...
```

Figure 6: Sequential Accesses

3.4.1 The Strided Access Structure

We introduce a strided access structure (SAS) that is used to maintain information about load or store instructions that the compiler determines have a strided access. Figure 7 shows the information that is stored in the SAS. The *V* bit indicates if the entry information is valid. The *L1 DC tag* and *L1 DC index* fields contain the virtual tag and physical index, respectively, associated with the L1 DC line (see Figure 2). The *L1 DC way* indicates in which way the associated L1 DC line resides. The *L1 DC index* and *L1 DC way* enables the CPU to check if the L1 DC line being accessed by the current load or store operation matches the L1 DC line information stored in the SAS. The *PP* field contains page protection bits from the DTLB.

	V	L1 DC tag	L1 DC index	L1 DC way	PP	**DV**	**word offset**	**strided data (SD)**
1								
...								
2^n–1								

Figure 7: Strided Access Structure (SAS)

Prior work has shown that it can be beneficial to always read as much data as possible from the L1 DC even if the requested data is narrow [2], e.g., reading a double word when only a byte is requested. Additional data were stored in a separate associative structure that is checked on subsequent loads and on a hit the L1 DC access is avoided. During compilation we identify strided accesses that have a shorter stride than the maximum readable data width from the L1 DC, such as iterating over a character array. The data is then stored in the SAS and subsequent strided accesses can access the much smaller SAS instead of the L1 DC. Figure 7 shows in bold the additional information that is stored in the SAS to support this type of strided accesses. The data valid (*DV*) bit indicates that the data stored in the SAS is valid. The *word offset* field contains the higher-order bits of the line offset (see Figure 2) identifying the data word of the L1 DC line that is stored in the SAS. The *strided data (SD)* contains a copy of the data identified by the *word offset*. The *DV* bit is needed as a data value is not fetched from the L1 DC on a store with a word mismatch.

We store virtual tags in the SAS, which enables the processor to avoid accessing the DTLB when the SAS entry being accessed is valid. It is easier to address the problems of using virtual tags in the SAS than in a much larger L1 DC. Synonyms (multiple virtual addresses mapping to the same physical address) can lead to problems after processing stores as there can be different values associated with the same location. Synonyms can be handled by invalidating other SAS entries that have the same *L1 DC way* and *index* when a new entry is allocated. Homonyms (one virtual address mapping to multiple physical addresses) can be resolved by invalidating all the SAS entries on a context switch, which is simplified due to using a write-through policy between the SAS entries and the L1 DC. The *PP* bits obtained from the DTLB when an SAS entry is allocated are used to ensure the data is properly accessed. L1 DC evictions due to L1 DC line replacements or multiprocessor cache coherency invalidations can be invalidated in the SAS by checking the *L1 DC way* and *index* stored in the SAS entries as the SAS is strictly inclusive to the L1 DC. Likewise, all the SAS entries should be invalidated when a page is replaced.

3.4.2 Strided Access Cases

We support four different accesses for loads with a stride less than or equal to half the L1 DC line size, as depicted in Table 5. A stride less than or equal to half the L1 DC line size means that the line is likely to be referenced at least twice. Invariant address references are viewed as having a stride of zero. The SAS tag and index (TI) pipeline action indicates that the tag and index from the SAS are compared against the tag and index of the memory address referenced. The SAS data (SD) pipeline action indicates that the data in the SAS is accessed and the appropriate value is extracted. The width of L1 DC is the number of bytes that can be transferred between the L1 DC and the CPU in one access, which is assumed to be 4-bytes in this paper. Only the last two cases access the data in the SAS, which occurs when the stride is less than or equal to half of the size of a word since it is likely that the data word in the SAS will be referenced at least twice. The benefits listed in the table are again relative to *case L0*, which is the conventional method for pipelining loads.

Case L4: The CPU checks the SAS tag and index (TI) field and a single data array (DA1) is accessed in the EX stage as the displacement is zero. The data array is speculatively accessed as there is a dependence in the next two instructions that would cause a stall. The benefits are that DTLB and tag array (TA) accesses are avoided, only one data array (DA1) is accessed, and one stall cycle is avoided.

Case L5: The CPU accesses a single data array (DA1) after the SAS tag and index (TI) check. This sequence of pipeline actions occurs when the displacement is zero and the loaded register is not used in the next two following instructions. Accessing the data array one cycle later will not cause a stall in this case and the data array will not be unnecessarily accessed if the SAS tag or index do not match. The same sequence occurs when the displacement is not zero as the data array cannot be accessed in the EX stage since

Table 5: Cases for Pipelining Strided Load Accesses

Case	Pipeline Stage Actions			Requirements				Benefits
	EX	DC1	DC2	Disp	Dep Inst	Stride	Example	
L4	TI+ DA1	DF		zero	1-2	$\frac{1}{2}$ L1 DC width $<$ stride && stride $\leq \frac{1}{2}$ line size	**r[2]**=M[r[4]]; r[3]=r[3]+**r[2]**;	no DTLB + no TA + 1 DA + avoid 1 stall
L5	TI	DA1	DF	zero	> 2		r[2]=M[r[4]+**4**];	no DTLB + no TA + 1 DA
	AG+TI			not zero			...	
L6	TI+ SD+DF			zero	1	stride $\leq \frac{1}{2}$ L1 DC width	**r[2]**=M[r[4]]; r[3]=r[3]+**r[2]**;	no DTLB + no TA + no DA + avoid 2 stalls
L7	TI	SD+ DF		zero	> 1		**r[2]**=M[r[4]+**4**]; r[3]=r[3]+**r[2]**;	no DTLB + no TA + no DA + 1 cycle early
	AG+TI			not zero				

Table 6: Cases for Pipelining Strided Store Accesses

Case	Pipeline Stage Actions			Requirements			Benefits
	EX	DC1	DC2	Disp	Stride	Example	
S2	TI	DA1		zero	$\frac{1}{2}$ L1 DC width $<$ stride && stride $\leq \frac{1}{2}$ line size	M[r[4]]=r[2];	no DTLB + no TA + 1 cycle early
	AG+TI			not zero			
S3	TI	SD+ DA1		zero	stride $\leq \frac{1}{2}$ L1 DC width	M[r[4]+**4**]=r[2];	no DTLB + no TA + 1 cycle early
	AG+TI			not zero			

the line offset has not yet been calculated. Note the tag and index comparison will occur near the end of the cycle after the address generation (AG) has been completed. The benefits are that DTLB and tag array (TA) accesses are avoided and only one data array (DA1) is accessed.

Case L6: The CPU checks the SAS tag and index (TI) field and the SAS data (SD) data value is extracted and formatted (DF) in the EX stage. The value can be extracted from the SAS data and formatted in a single cycle since the SAS can be quickly accessed. All of these pipeline actions occur in the EX stage to avoid a stall with the following instruction that referenced the loaded register. The benefits are that DTLB and tag array (TA) accesses are avoided, no data arrays are accessed, and two stall cycles are avoided.

Case L7: The CPU checks the SAS tag and index (TI) field in the EX stage, and the SAS data (SD) value is extracted and formatted (DF) in the DC1 stage. This case is applied either when the displacement is zero and the next instruction does not use the loaded register, or when the displacement is not zero as the SAS data (SD) cannot be accessed in the EX stage since the offset is not known at that time. The benefits are that DTLB and tag array (TA) accesses are avoided, no data arrays are accessed, and the data is obtained one cycle early, which in some cases may avoid a stall.

It is possible that the SAS tag and index will match, but the SAS word offset does not match in *cases L6* and *L7*. If so, then the data will be accessed from the data array in a manner similar to *case L5*. Likewise, it is possible that the SAS tag or index will not match for any of the cases in Table 5. In these situations the data will be accessed from the L1 DC in a manner similar to *case L0* to ensure stalls are minimized.

We support two different accesses for stores with a stride less than or equal to half the L1 DC line. The two cases are depicted in Table 6. In *case S2* the data value is *not* written to the SAS, while in *case S3* it is. Both cases have the benefits of not accessing the DTLB and the tag arrays (TA) and completing the store operation one cycle earlier than a conventional store. These benefits can reduce energy usage and may help to avoid structural hazards.

The L1 DC data array is updated on each store instruction to ensure consistency with SAS data. Each conventional store instruction also requires an associative check of the tag, index, and offset stored in all valid SAS entries so that any entry that matches will have its data field updated. This capability is possible since the SAS consists of only a few entries and is implemented in flip flops. To avoid aliases in the structure, a strided access entry is invalidated when it matches the L1 DC tag and index of a different entry being allocated. Strided access entries that match an L1 DC line being evicted are also invalidated.

4. Enhanced Load and Store Compilation

A number of compiler modifications are required to support the enhanced loads and stores described in this paper. First, the compiler uses the algorithm shown in Figure 8 to allocate memory references to the SAS entries shown in Figure 7 for the cases depicted in Tables 5 and 6. Conventional basic induction variable analysis is used to determine strides at compile time. Strided references with a stride that is less than one half of the L1 DC line size are allocated starting with the innermost loops first. Note that invariant address references are treated as having a stride of zero.[1] A reference is merged with an existing entry if it is within one half of a line's distance of a reference associated with that entry. If there are more entries allocated than are available, then the entries with the most estimated references are selected. Memory references even at the outermost level of a function are allocated when they are detected to likely be in the same line as a previous reference, which is common when multiple registers are saved and restored in the function prologue and epilogue, respectively, as shown in Figure 6. Strided access entries are allocated without regard to which entries may be allocated to references in a called function since the tag and index (TI) field is checked on each access to the strided access structure.

Next, the compiler classifies each memory reference, which requires detecting the displacement size (*large*, *small*, *zero*) and the distance between a load and the first instruction that uses the loaded register. Since other transformations can affect both this displacement and distance, memory references are classified after all other conventional compiler optimizations have been applied, including instruction scheduling that attempts to separate loads from dependent instructions. The load and its first dependent instruction may not always be in the same basic block. Thus, a forward recursive search to return the minimum distance in instructions is performed in all successor blocks if a dependent instruction is not found in the

[1] A loop-invariant address reference indicates that the address used to access memory does not change in the loop. Note this is not the same as a loop-invariant value. For instance, a global variable could be referenced inside of a loop that contains a function call, which prevents the value of the global variable from being hoisted out of the loop.

```
FOR each loop level in function
    in order of innermost first DO
   FOR each strided or loop-invariant load and store
       in the loop DO
      IF stride <= 1/2 line size THEN
         IF reference <= 1/2 line size distance from
                 another entry for the same loop THEN
            Merge the reference with the existing entry;
         ELSE
            Allocate the reference to a new entry;
IF too many entries THEN
   Select the entries with the most estimated references;
```

Figure 8: Algorithm for Allocating Strided Access Entries

basic block containing the load and the maximum number of instructions (three in our pipeline) has not already been encountered. Figure 9 shows an example where the dependent distance between the load and an instruction that uses the loaded register is conservatively calculated to be two instead of three as the branch could be taken. A call or a return instruction can also be encountered before the maximum distance to an instruction with a loaded register is found. In such cases the compiler assumes the distance to the first dependent instruction is four or more instructions.

```
    r[2]=M[r[4]];
    PC=r[3]!=r[4],L5;
    r[4]=r[6];
    r[6]=r[6]+r[2];
L5: r[3]=r[3]+r[2];
```

Figure 9: Load with Multiple First Dependents Example

Once all the memory references have been classified, another compilation pass reclassifies memory references that can lead to structural hazards. Consider the example code segment in Figure 10(a). Both load instructions have small displacements and assume they are not strided. The first load has a dependence four instructions later and has been classified as *case L3* in Table 3. The second load has a dependence three instructions later and has been classified as *case L2* in Table 3. However, classifying these two loads in this manner results in a stall due to a structural hazard for the tag array (TA), one data array (DA1), and data formatting (DF), as shown in Figure 10(b). In this case the compiler reclassifies the first load instruction from *case L3* to *case L2* as shown in Figure 10(c) to avoid the structural hazard stall. We analyzed all possible pairs of classifications of memory operations in order to determine how to best avoid each structural hazard.

In addition to the required compiler modifications previously outlined, the hardware enhancements of our techniques enable us to perform the following optimization: The compiler hoists a loop-invariant displacement address from a load instruction out of a loop when avoiding the address generation could eliminate a stall or reduce energy usage. For instance, consider the example in Figure 11(a). Each of the load instructions uses a loop-invariant displacement address, as `r[sp]` is invariant in the loop. Assume these loads are not hoisted out of the loop as these memory references may be updated elsewhere in the loop. Further assume all of the strided access entries have been allocated to other memory references. The first load instruction has a dependence five instructions away, so it is classified as *case L3* in Table 3. The displacement address of the first load is not hoisted out of the loop as this will not eliminate a stall or avoid access to the tag arrays (TA) or a data array (DA1). However, the second and third load instructions each has a dependence in the second instruction following the respective load. Because they each has a large displacement value, these two loads are classified as *case L0* in Table 3. Assume there is only one

```
load1:  r[2]=M[r[4]+8];
load2:  r[3]=M[r[5]+4];
        r[4]=r[4]+8;
        r[5]=r[5]+4;
        r[8]=r[3]+r[2];
```

(a) Instructions

	1	2	3	4	5	6
load1 [L3]:	...	AG	**TA**	DA1	DF	...
load2 [L2]:		...	stall	AG+TA	**DA1**	**DF**

(b) Original Classification of Loads

	1	2	3	4	5	6
load1 [L2]:	...	AG+TA	DA1	DF	...	
load2 [L2]:		...	AG+TA	DA1	DF	...

(c) After Reclassifying Load 1

Figure 10: Reclassifying Memory Operations Example

available register within the loop. The compiler hoists one of the two loop-invariant displacement addresses out of the loop as shown in Figure 11(b). Since the two displacement addresses use the same invariant base register and the difference between the two is *small*, the compiler modifies the displacement address of the third load to use the newly allocated register `r[8]` and the displacement to be relative to the second load's original displacement. The initial and revised classifications of the second and third loads are shown in Figures 11(c) and 11(d), respectively. The second load that now has a zero displacement is classified as *case L1* in Table 3 to obtain the data one cycle earlier to avoid a stall cycle. The third load that now has a small displacement is classified as *case L2* in Table 3 to enable accessing only a single data array (DA1), which will reduce energy usage.

```
    ...
L4: r[3]=M[r[sp]+96];
    r[4]=M[r[sp]+72];
    r[5]=M[r[sp]+76];
    r[6]=r[6]+r[4];
    r[6]=r[6]+r[5];
    r[6]=r[6]-r[3];
    ...
    PC=r[7]!=0,L4;
```

(a) Original Loop Code

```
    ...
    r[8]=r[sp]+72;
L4: r[3]=M[r[sp]+96];
    r[4]=M[r[8]];
    r[5]=M[r[8]+4];
    r[6]=r[6]+r[4];
    r[6]=r[6]+r[5];
    r[6]=r[6]-r[3];
    ...
    PC=r[7]!=0,L4;
```

(b) After Hoisting a Loop−Invariant Address

	1	2	3	4	5	6
load2 [L0]:	...	AG	TA+DA	DF	...	
load3 [L0]:		...	AG	TA+DA	DF	...

(c) Initial Classification of 2nd and 3rd Loads

	1	2	3	4	5	6
load2 [L1]:	...	TA+DA	DF	...		
load3 [L2]:		...	AG+TA	DA1	DF	...

(d) Reclassification of 2nd and 3rd Loads

Figure 11: Hoisting Loop-Invariant Displacement Example

5. Conveying Load/Store Context to the HW

There has to be a means of conveying to the hardware the information extracted by the compiler about the context in which a memory operation is performed. We assume an instruction set architecture (ISA) with a displacement-based addressing mode where the load and store instruction formats have the displacement in an immediate field, e.g., the MIPS I instruction format shown in Figure 12(a). One solution to convey the additional information is to restrict the immediate field to use fewer bits and use the remaining bits to encode the necessary information. The higher-order bits of the immediate field in Figure 12(b) are used to indicate the type of enhanced memory reference to perform.

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate

(a) MIPS Instruction I Format

6 bits	5 bits	5 bits	2+n bits	14−n bits
opcode	rs	rt	**meminfo**	immediate

(b) Enhanced Load and Store Instruction Format

Figure 12: Instruction Format for Loads and Stores

Table 7 shows how the *meminfo* field in Figure 12 is interpreted. Two bits are used to identify the type of memory reference depicted in Tables 3 through 6. Note that the instruction opcode indicates whether or not the memory operation is a load or a store. Two bits are needed to distinguish *cases L4-L7* since all four cases can have a displacement value of zero and the distance to the first dependent instruction and the stride is unknown when a conventional load instruction is decoded. The remaining *n* bits indicate which of the $2^n - 1$ strided access entries is referenced if a case in Tables 5 or 6 is specified. Note that a value of zero indicates that a case in Tables 3 or 4 is to be used instead. These ISA changes are quite small in comparison to the benefits they achieve.

Table 7: *meminfo* Value When n=2

2 bits	Non-Strided Accesses		Strided Accesses	
	Entry	Case	Entry	Case
00	00	L0 or S0[a]	not 00[b]	L4 or S2[a]
01	00	L1 or S1[a]	not 00[b]	L5 or S3[a]
10	00	L2	not 00[b]	L6
11	00	L3	not 00[b]	L7

[a] Depending on the instruction's opcode.

[b] The *entry* is used to index into the SAS.

6. Evaluation Framework

We evaluated our techniques in the context of in-order processors, which is a class of processors of growing importance as computation is facing ever more stringent power and energy requirements. In-order processors are common in mobile devices, ranging from sensors to high-end phones, and are showing great promise for data center and scale-out workloads [3].

We used the VPO compiler [4] to implement the optimizations discussed in Section 4. Based on 20 different MiBench benchmarks [5], we used SimpleScalar [6] with a heavily modified load and store pipeline to extract micro-architectural events. We estimated the data access energy based on the number of micro-architectural events and the energy per event. The energy dissipated for the different L1 DC and SAS events (see Table 8) was estimated from placed and routed netlists in a 65-nm commercial process technology at nominal process conditions and 1.2 V.

Note that the presented results are for benchmarks where the run-time library has not been compiled with the presented techniques and it is very likely that both performance and energy would improve if they also would be compiled.

Table 8: Energy for Different Events

Event	Energy (pJ)
Read tags for all ways	57.3
Read 32-bit data for all ways	84.4
Read 64-bit data for all ways	168.8
Write 32-bit data	20.4
Write 64-bit data	40.8
Read 32-bit data	21.2
Read 64-bit data	42.4
Compare SAS tag and index (1/3/7 entry)	0.10 / 0.23 / 0.75
Read SAS word (1/3/7 entry)	0.5 / 1.1 / 3.4
Write SAS word (1/3/7 entry)	1.8 / 2.4 / 3.2
DTLB (16 entries, fully associative)	17.5

The simulated architecture is an in-order processor with a 3-stage L1 DC pipeline, as shown in Figure 1. The L1 instruction and data caches are 16kB 4-way associative and the translation lookaside buffers (TLBs) have 16 entries. The L1 DC is implemented using two 32-bit wide SRAM banks for each way. This results in an energy-efficient L1 DC implementation; single word accesses only need to probe a single SRAM, while the L1 DC still can handle double word accesses in a single cycle by probing both the SRAMs. Figure 13 shows the L1 DC configuration that is used for the energy estimation. Each way consists of two data banks, which can be independently probed for read operation. Since each of the data way is 4kB for a 16kB 4-way associative cache, we use two 512wX32b-m4 banks for each way. This configuration allows to access double words with the same latency of a single word, while trying to keep the overhead as low as possible for single word accesses, by probing only a single bank for those accesses. The ARM Cortex A7 is an example of an in-order processor that utilizes this type of L1 DC configuration. We have not included the energy overhead of the control logic, which results in slightly optimistic energy savings.

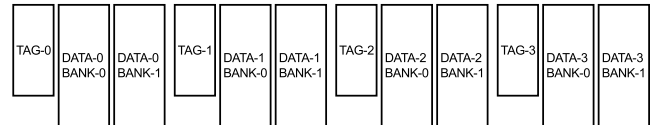

Figure 13: 16kB 4-way Set Associative Cache Configuration with Support for Efficient Single and Double Word Accesses

7. Results

We first varied the number of SAS entries. Note there are actually two separate SAS structures. One structure does not contain data and is used for *cases L4*, *L5*, and *S2*. The other structure does contain data (bold portion of Figure 7) and is used for *cases L6*, *L7*, and *S3*. Figure 14 shows the average effect on L1 DC and DTLB energy when context-aware load and store operations are used and the number of strided access entries is varied from 1, 3, and 7.[2] The energy is normalized to a conventional L1 DC and DTLB when no enhanced operations and no SAS structures are used. As the number of SAS entries is increased, the L1 DC and DTLB energy is reduced since the number of accesses to the L1 DC and DTLB is decreased. However, the difference in L1 DC and

[2] We only estimated L1 DC/DTLB/SAS usage energy. While our approach can avoid many address generation additions, this is not included in our energy results.

DTLB energy between three and seven SAS entries is very small, which indicates that three SAS entries for each structure are enough to capture most of the strided references. Also as the number of SAS entries increases, the energy expended by the SAS increases. The 3-entry SAS is the most energy efficient and reduces L1 DC and DTLB energy by 43.5%. Figure 15 shows the average effect on execution time when the number of strided access entries is varied from 1, 3, and 7. The execution time decreases as the number of SAS entries is increased since more loads can potentially avoid stalls (*cases L4*, *L6*, and *L7*). The benefit from avoiding load stalls is partially offset by additional instructions that are executed when large displacements can no longer fit in load and store instructions, as depicted in Figure 12. The performance difference between using 3 SAS entries (6.2%) and 7 SAS entries (6.6%) is very little. The remaining results assume 3-entry SAS structures, which are very small and occupy only about 3% of the L1 DC area.

The proposed instruction format has a smaller immediate field compared to the base case that we use for comparison. Our compiler therefore has to constrain the range of the displacement to fit in the format shown in Figure 12(b). The more restricted immediate field has little effect on the number of executed instructions since most load and store displacement values require only a few bits.

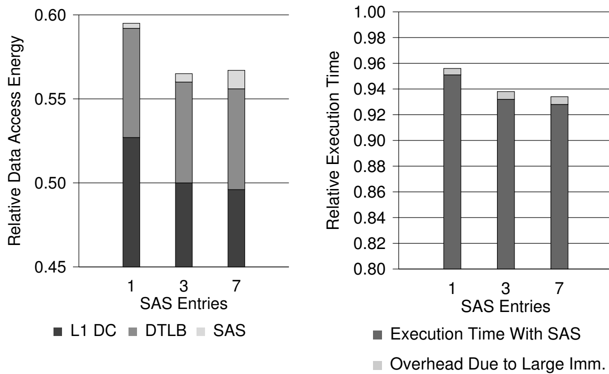

Figure 14: Access Energy

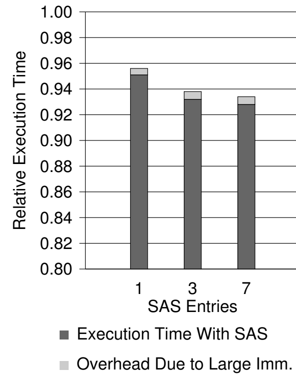

Figure 15: Execution Time

Figure 16 shows the dynamic frequency of the load and store classifications. About 50% of the loads and 63% of the stores were classified as strided (*cases L4* to *L7* and *cases S2* to *S3*). About 35% of the loads and 17% of the stores were classified as not strided (*cases L0* to *L3* and *cases S0* to *S1*). Only about 4% of the loads and 10% of the stores received the default classification of *L0* or *S0*, respectively. However, a significant fraction of the strided classifications (*cases L4-L7* and *S2-S3*) were converted to the default classifications (*L0* and *S0*) at run-time due to not matching the SAS *tag+index*. Likewise, some strided loads that access data (*cases L6* and *L7*) were converted to not access data (*case L5*) due to a *word offset* miss. The portion that was converted is depicted in the bars by the color that matches the color of *L0* and *S0* or the bottom portion of *L5*. Note there are no performance penalties for converting to other classifications at run-time since these misspeculations are discovered before the L1 DC is accessed. We did not compile the entire run-time library, which resulted in 15% of the loads (UL) and 19% of the stores (US) being unclassified and treated as *cases L0* and *S0*, respectively. Note that classifying these additional loads and stores should further improve the energy and execution time benefits.

Figure 17 shows the effect that each classification had on reducing L1 DC and DTLB energy usage as compared to the *L0* and *S0* default classifications. *Cases L7* and *S2* had the most impact since they were the most frequently used classifications (see color of bars in Figure 16) and could completely avoid both DTLB and L1 DC

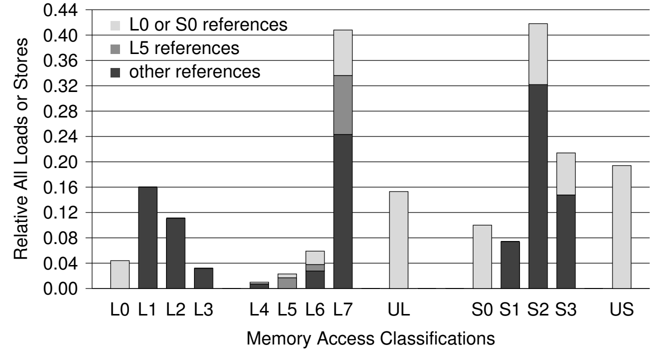

Figure 16: Distribution of Cases for Loads and Stores (UL and US are unclassified loads and stores)

accesses. If an SAS structure was not utilized, then *cases L2* and *L3* would have a more significant impact on energy usage. Note that *cases L0*, *L1*, *S0*, and *S1* have no impact on energy, as *L0* and *S0* are the default classifications and *L1* and *S1* access the cache early, but in a conventional manner.

Figure 18 shows the impact of the various cases on execution time as compared to the default classifications of *L0* and *S0*. Only *L1*, *L4*, *L6*, and *L7* can improve performance. Many of the loads classified as *L6* and *L7* were converted to *L5* loads when the *word offset* did not match the SAS entry data word. Likewise, a significant percentage of the *L4*, *L6*, and *L7* loads did not hit in the SAS and were converted to *L0* loads. *L6* has a higher relative impact on performance than implied by Figure 16 since each reference avoids two stall cycles.

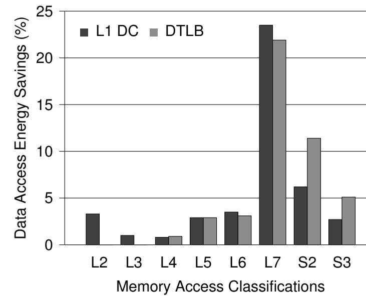

Figure 17: Energy Gain by Cases

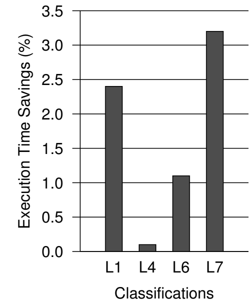

Figure 18: Execution Time Gain by Cases

Figures 17 and 18 clearly illustrate that not all cases contribute equally to the energy and performance gain. *Case L2*, *L3*, *L4*, and *L5* show limited improvements in both energy and performance. One could consider excluding these cases to simplify the hardware implementation and encode the fewer cases more compactly in load and store instructions. *Case L7* is responsible for much of the load energy reduction and *cases L1*, *L6*, and *L7* are responsible for most of the execution time improvement. An alternative instruction encoding could be to use a single bit to encode the nonstrided *L0* and *L1* and the strided *L6* and *L7* cases. For a 3-entry SAS, this would mean decreasing the number of bits used to encode the load or store context from four (2+2) to three (1+2) (see Figure 12). Such a decrease could be beneficial for instruction set architectures with more restrictive displacement fields like the ARM instruction set that only has 12-bit displacements for loads and stores.

Figure 19 shows the decrease in DTLB and L1 DC tag array accesses and L1 DC data array accesses. Decreases in DTLB and L1 DC tag array accesses are obtained from hits in the SAS structure (*L4-L7* and *S2-S3*). Decreases in L1 DC data array accesses come

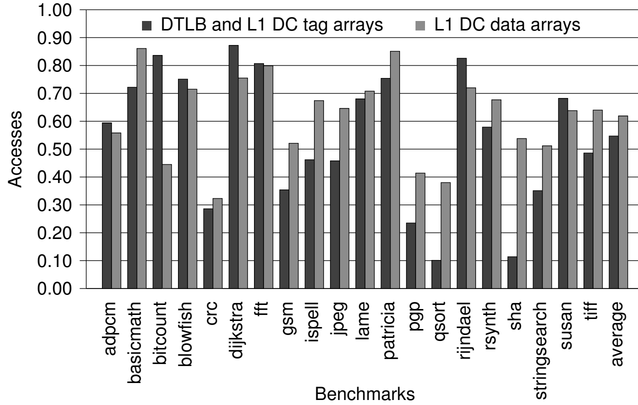

Figure 19: DTLB/L1 DC Tag Array and L1 Data Array Accesses by Benchmark

from accessing 1 of 4 (*L2-L5*) or accessing 0 of 4 (*L6-L7*) L1 DC data arrays. On average, 45% of the DTLB accesses and L1 DC tag checks and 38% of the L1 DC data array accesses are avoided.

Figure 20 shows the effect on L1 DC and DTLB energy for each benchmark. The energy reductions varied from as low as 16.8% for *dijkstra* to as high as 81.1% for *qsort* with an average of 43.5%. These variations are affected by the context in which the frequent load and store instructions are performed that determined their classification. Since the execution time improvements of our proposed techniques are quite significant the overall CPU energy dissipation will be reduced. However, the energy usage improvements presented in this paper are for the L1 DC and the DTLB only.

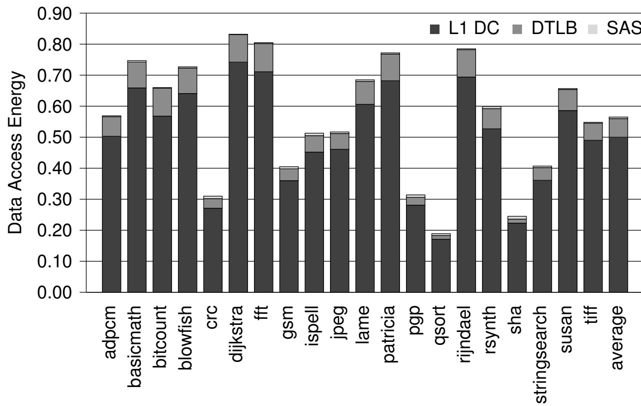

Figure 20: Data Access Energy by Benchmark

Figure 21 shows the effect on execution time for each benchmark. The execution time reductions varied from almost no effect up to 20% with an average of about 6% with every benchmark providing some improvement.

8. Related Work

It is common for level-two (L2) or level-three (L3) caches to sequentially access their tag and data arrays to reduce energy usage [7]. This approach is practical because memory accesses to the L2 and L3 caches are relatively few compared to L1 cache accesses, hence the execution time overhead is low. It has been shown that always sequentially accessing the tag and data arrays for load operations to the L1 DC incurs an unacceptable performance penalty [8].

Many techniques have been proposed to reduce energy in set-associative L1 DCs. Unlike our context-aware techniques, way-prediction techniques have a relatively high performance penalty

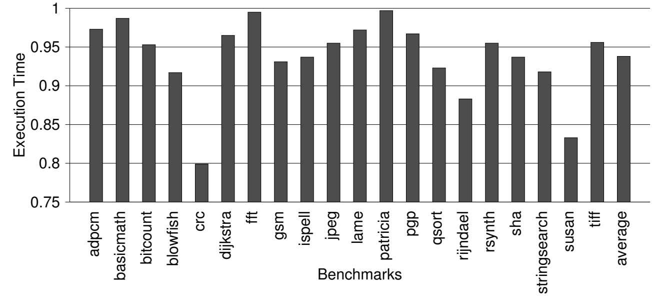

Figure 21: Execution Time by Benchmark

of several percent [8, 9]. A way cache was used to save the L1 DC way of the last 16 cache accesses in a table, and each memory access performs an associative tag search on this table [10]. If there is a match, then only a single L1 DC way is activated. The energy to access this way cache is likely similar to the energy of accessing a DTLB. In contrast, our strided access structure is much smaller and at most only a single entry is accessed for each load reference. Way halting is another method for reducing the number of tag comparisons [11], 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 specialized memory implementations that might have a negative impact on the maximum operational frequency.

Witchel et al. propose a hardware-software design for data caches using direct address registers (DARs) [12]. 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 DAR. The DAR approach is somewhat similar to the approach we use for *case L5* in Table 5 as both approaches annotate load and store instructions and store the way in which the data resides to avoid tag checks and *n*-1 data array accesses for an *n*-way associative L1 DC. However, the DAR approach requires several compiler transformations, such as loop unrolling and stripmining, to make these guarantees, which can result in code size increases. In contrast, our strided access structure contains a tag and index field to ensure that the entry matches the reference, which enables our approach to potentially avoid more L1 DC tag checks with fewer changes to the compiler. In addition, our techniques also avoid more L1 DC data array accesses (*L2-L3*, *L6-L7*) and improve performance (*L1*, *L4*, *L6-L7*).

Other small structures have been proposed to reduce L1 DC energy usage. A line buffer can be used to hold the last line accessed in the L1 DC [13]. The buffer must however be checked before accessing the L1 DC, placing it on the critical path, which can degrade performance. The use of a line buffer will also have a high miss rate, which may sometimes increase L1 DC energy usage due to continuously fetching full lines from the L1 DC memory. A small filter (L0) cache accessed before the L1 DC has been proposed to reduce the power dissipation of data accesses [14]. However, filter caches reduce energy usage at the expense of a significant performance penalty due to their high miss rate, which mitigates some of the energy benefits of using a filter cache and has likely discouraged its use. A technique to speculatively access an L0 data cache in parallel with its address calculation was proposed to not only eliminate the performance penalty but also provide a small performance improvement compared to not having an L0 data cache [15]. While this approach does not require any ISA changes, our context-aware

loads and stores approach is simpler to implement in hardware and provides comparable energy and greater performance benefits. A tagless access buffer (TAB) has been proposed to reduce the energy usage of strided access [16] by accessing a compiler managed structure instead of a more expensive L1 DC. However, the use of a TAB requires additional instructions to be inserted and prefetching of data, which results in both greater ISA changes and a more complex hardware implementation than the presented approach.

One proposed scheme, which claims to eliminate about 20% of the cache accesses, always reads the maximum word size even though the load may only be for a byte. The additional data read can then be used in a later memory reference without accessing the L1 DC [2]. This approach requires an associative check of *block numbers* and *word offsets* of all the entries in the load queue. In contrast, our approach only requires checking one strided access entry.

9. Future Work

There are several enhancements that could be made to improve the energy and performance benefits of context-aware loads and stores. Compiling the entire run-time library would enable enhancement of more loads and stores, which should improve both energy usage and performance. There are often cases where an SAS entry is allocated in a loop, but is evicted before being accessed again due to a function being called in that same loop. Performing interprocedural analysis would enable different SAS entries to be used, which should result in fewer SAS evictions. The techniques proposed in this paper were evaluated for an in-order pipeline, but they are also potentially beneficial for out-of-order (OoO) pipeline. While it is likely that some of the load stalls can be avoided in an OoO processor, much of the energy benefits from exploiting context-aware loads and stores may be possible. An experimental study is needed to determine how to best apply these techniques within an OoO pipeline and their potential benefits.

Currently the classification of loads and stores is performed after all compiler optimizations have been performed, which includes instruction scheduling. The VPO scheduler attempts to separate loads from dependent instructions as far as possible. One possibility is to integrate the classification of loads and stores with the instruction scheduler in an attempt to order instructions to achieve the most effective classifications.

10. Conclusions

We have demonstrated in this paper that both data access energy and performance can be improved by applying different techniques for memory operations based on the compile-time context in which they are performed. We simultaneously achieved a 6% decrease in execution time and a 43% reduction in L1 DC and DTLB energy usage. Our techniques are relatively simple to implement in a processor, can be achieved without aggressive compiler optimizations, and require only small changes to an instruction set architecture.

11. Acknowledgements

We appreciate the comments provided by the anonymous reviewers of this paper. This research was supported in part by the Swedish Research Council grant 2009-4566 and the US National Science Foundation grants CNS-0964413 and CNS-0915926.

References

[1] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. Speculative tag access for reduced energy dissipation in set-associative L1 data caches. In *IEEE Int. Conf. Computer Design*, pages 302–308, October 2013.

[2] L. Jin and S. Cho. Macro data load: An efficient mechanism for enhancing loaded data reuse. *IEEE Trans. on Computers*, 60(4):526–537, April 2011.

[3] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kockberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Annual Int. Symp. Computer Architecture*, pages 500–511, June 2012.

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 329–338, June 1988.

[5] 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 *IEEE Int. Workshop/Symp. on Workload Characterization*, pages 3–14, December 2001.

[6] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[7] D. Williamson. ARM Cortex A8: A high performance processor for low power applications. In E. John and J. Rubio, editors, *Unique Chips and Systems*. CRC Press, 2007.

[8] 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 *IEEE/ACM Annual Int. Symp. Microarchitecture*, pages 54–65, December 2001.

[9] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Int. Symp. Low Power Electronics and Design*, pages 273–275, August 1999.

[10] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero. Fast speculative address generation and way caching for reducing L1 data cache energy. In *IEEE Int. Conf. Computer Design*, pages 101–107, October 2006.

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

[12] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *IEEE/ACM Annual Int. Symp. Microarchitecture*, pages 124–133, December 2001.

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

[14] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *IEEE/ACM Annual Int. Symp. Microarchitecture*, pages 184–193, December 1997.

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

[16] A. Bardizbanyan, P. Gavin, D. Whalley, M. Själander, P. Larsson-Edefors, S. McKee, and P. Stenström. Improving data access efficiency by using a tagless access buffer (TAB). In *Int. Symp. Code Generation and Optimization*, pages 269–279, February 2013.