Decoupling Address Generation from Loads and Stores to Improve Data Access Energy Efficiency

Michael Stokes
Florida State University
Tallahassee, Florida, United States
mstokes@cs.fsu.edu

Ryan Baird
Florida State University
Tallahassee, Florida, United States
baird@cs.fsu.edu

Zhaoxiang Jin
Michigan Technological University
Houghton, Michigan, United States
zjin3@mtu.edu

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

Soner Onder
Michigan Technological University
Houghton, Michigan, United States
soner@mtu.edu

Abstract

Level-one data cache (L1 DC) accesses impact energy usage as they frequently occur and use significantly more energy than register file accesses. A memory access instruction consists of an address generation operation calculating the location where the data item resides in memory and the data access operation that loads/stores a value from/to that location. We propose to decouple these two operations into separate machine instructions to reduce energy usage. By associating the data translation lookaside buffer (DTLB) access and level-one data cache (L1 DC) tag check with an address generation instruction, only a single data array in a set-associative L1 DC needs to be accessed during a load instruction when the result of the tag check is known at that point. In addition, many DTLB accesses and L1 DC tag checks are avoided by memoizing the DTLB way and L1 DC way with the register that holds the memory address to be dereferenced. Finally, we are able to often coalesce an ALU operation with a load or store data access using our technique to reduce the number of instructions executed.

CCS Concepts • **Hardware → Power estimation and optimization**; • **Software and its engineering → Compilers**;

Keywords Data Accesses, Energy Efficiency, Compiler Optimizations

ACM Reference Format:
Michael Stokes, Ryan Baird, Zhaoxiang Jin, David Whalley, and Soner Onder. 2018. Decoupling Address Generation from Loads and Stores to Improve Data Access Energy Efficiency. In *Proceedings of 19th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3211332.3211340

1 Introduction

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

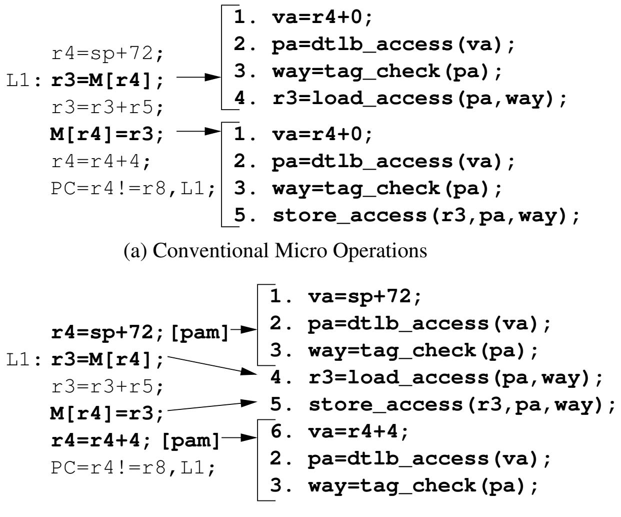
Figure 1(a) shows code containing a load and a store along with the μops that implement these instructions. The load μops are: #1 Add the base register value and the offset to obtain the virtual address (*va*); #2 Access the data translation lookaside buffer (DTLB) using the *va* to get the physical address (*pa*); #3 Perform the tag check to identify the *way* where the data resides in a set-associative cache; and #4 use the *pa* index and the *way* to access the cache data and update the register. The store uses identical μops #1, #2, and #3, but #5 assigns the data to the cache line. Given that the load and store access the same location, the first three μops for the store are clearly redundant, which can be eliminated if the store instruction can use the results of prior μops. Unfortunately, these μops are not visible to the compiler with conventional ISAs and it would be expensive to implement each μop as an ISA instruction in terms of code size, fetch bandwidth, and energy.

Eliminating redundant μops can be accomplished without exposing each μop as an explicit instruction to the compiler. In our example, μop (1) `va=r4+0;` can be combined with the two instructions that update `r4`. Coupling μops (2) and (3) into these instructions effectively creates a prepare to access memory (*pam*) instruction, yielding the code shown in Figure 1(b) that eliminates the redundant virtual address calculation, DTLB access, and L1 DC tag check in the loop.

LCTES'18, June 19–20, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5803-3/18/06…$15.00
https://doi.org/10.1145/3211332.3211340

Table 1. Last Instruction to Compute a Data Address

Size	Type	Operation	MIPS Inst Effect	Source Operands
Scalar	Local	(1) integer immediate add	rd = rs + immed	stack pointer and offset
	Global	(2) bitwise immediate OR	rd = rs | immed	high | low portions of global address
	Pointer	(3) integer load	rt = M[rs]	pointer variable address
Composite	Array Element	(4) integer register add	rd = rs + rt	array base address and element offset
	Structure Field	(1) integer immediate add	rd = rs + immed	struct base address and field offset
	Pointer Arithmetic	(5) integer register sub	rd = rs - rt	pointer - var offset



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

(a) Conventional Micro Operations

```
                          1. va=sp+72;
    r4=sp+72; [pam]       2. pa=dtlb_access(va);
L1: r3=M[r4];             3. way=tag_check(pa);
    r3=r3+r5;             4. r3=load_access(pa,way);
    M[r4]=r3;             5. store_access(r3,pa,way);
    r4=r4+4; [pam]        6. va=r4+4;
    PC=r4!=r8,L1;         2. pa=dtlb_access(va);
                          3. way=tag_check(pa);
```

(b) Decoupled Micro Operations

Figure 1. Memory Access Micro Operations

Note that the *pam* in the figure is simply an *annotation*. An instruction whose destination result can be used as the address input of a memory operation can be annotated, including integer load instructions used for pointer chasing. Hence, *μ*op results, such as *pa*, and *way* are effectively coupled with the result register (*r4*) of the *pam* instruction by the micro-architecture and essentially extend the live-ranges of these values to other instructions.

In this paper we propose the *Decoupled Address Generation and Data Access* (DAGDA[1]) technique to separate the micro-operations associated with memory accesses and distribute them to other instructions. This separation facilitates energy optimizations, some of which rely on simple memoization techniques that are implemented by using the destination register number of *pam* annotated instructions as an index into simple small tables maintained by the micro-architecture.

2 Background

In this paper we describe our proposed techniques in the context of an in-order pipeline where the benefits are more obvious. In-order pipelines are commonly used in many embedded processors, are the only option for extreme low-power systems, and are growing in importance as computation is facing more stringent power and energy requirements. However, our proposed techniques are also applicable in out-of-order (OoO) processors.

Figure 2 depicts how a classical in-order pipeline performs a load from an *n*-way set-associative L1 DC. The virtual memory address is generated by adding a displacement to a base address obtained from the register file in an address generation (ADDR-GEN) stage. In the SRAM-ACCESS stage the DTLB, the L1 DC tags, and the L1 DC data are all accessed in parallel to minimize load hazard stalls and the tag value of the physical address is compared to the tag value of the physical page number from the DTLB.[2] This organization is energy inefficient as all data arrays are accessed, but the value can reside in at most one way within a cache set.

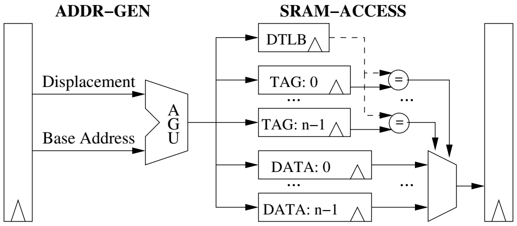


Figure 2. Conventional L1 DC Pipeline Load Access

Figure 3 shows the address fields used to access the DTLB and the L1 DC. The virtual page number is used to access the DTLB, which produces the corresponding physical page number. The virtual and physical page offsets remain the same. The L1 DC block number uniquely identifies the L1 DC line being accessed. The *L1 DC offset* indicates the first byte of the data to be accessed in the L1 DC line. The *set index* is used to access the L1 DC set. The *tag* contains the remaining bits that are used to verify if the line resides in the L1 DC.[3]

3 Decoupling the Address Generation and Memory Access Operations

The SRAM-ACCESS pipeline stage as depicted in Figure 2 is inefficient with respect to energy usage since all L1 DC data arrays must be accessed for loads as the data access occurs in parallel with the L1 DC tag check. It is possible

[1] A simple, but very powerful Celtic God.

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

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

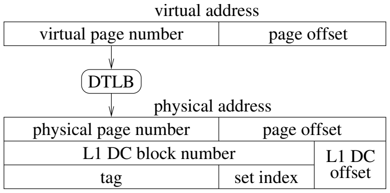


Figure 3. Address Fields

to extend the instruction pipeline to have a separate stage for accessing the DTLB and L1 DC tag arrays so only a single L1 DC data array needs to be accessed. We found that across the MiBench benchmarks [6] the execution time increases by 8% on average for an in-order processor whose L1 DC access is increased from two stages to three stages to facilitate sequential tag and data accesses [2]. The change is impractical since the reduced energy usage for the L1 DC accesses would be largely offset by the increased energy required for longer execution times of applications.

As shown in Figure 2, an in-order instruction pipeline includes separate pipeline stages for generating the memory address (ADDR-GEN) and accessing the data cache (SRAM-ACCESS) within a load instruction. An address generation step is included in an instruction pipeline since most processors support a displacement addressing mode for a memory operation, where the effective address is the sum of a base register value and a sign-extended immediate offset (i.e., *M[rs+immed]*). We propose that the DTLB access and L1 DC tag check be decoupled from the L1 DC data access by associating these operations with different instructions. The possible last instructions in the computation of the addresses of variables in a C/C++ application for almost all cases are shown in Table 1. Thus, these five instructions can either be annotated or separate opcodes can be used to indicate that the integer destination register can be subsequently used to dereference a data value. The actual data access to perform the load or store will now use only a register indirect addressing mode (i.e., *M[rs]*).

Decoupling address generation and data access into separate instructions does not significantly increase the instructions executed for the following reasons. (1) Load and store instructions in our compiler used a zero displacement 46% of the time for the *MiBench* benchmark suite. Many memory references sequentially access array elements and the displacement becomes zero after performing the loop strength reduction optimization, where the array base address is assigned to a register before the loop and an integer addition is used to calculate the next array element address in the loop. (2) Sometimes the address generation calculation is redundant. In fact, a memory address using a non-zero displacement is often loop invariant, such as referencing a local variable that uses the stack pointer register, and can be hoisted out of loops. (3) We will later show in Section 5 that we can encode an ALU operation with load and store operations that use a simple indirect register addressing mode.

Table 2 shows the DAGDA pipeline stages for a traditionally pipelined in-order processor extended with decoupled address generation and data accesses. The first five are conventional stages found in a traditional pipeline. The AG stage includes generating an address through an integer addition or bitwise OR operation. The TC and DA stages in the table comprise other actions that are typically associated with a single conventional data cache access (MEM) pipeline stage. The TC stage accesses the DTLB to obtain the physical page number and accesses the L1 DC tag arrays to check if the desired line is resident in the L1 DC. Both the DTLB and L1 DC tag accesses occur in parallel. The DA stage accesses a single L1 DC data array to either load or store a value.

Table 2. DAGDA Inst Pipeline Stages

Stage	Explanation	Stage	Explanation
1. IF	inst fetch	6. AG	address generation
2. ID	inst decode	7. TC	DTLB access and L1 DC tag check
3. RF	register fetch		
4. EX	execute	8. DA	L1 DC data access to load/store a value
5. WB	write back		

Table 3 shows DAGDA pipeline stages applied for various instructions. Unlike a conventional pipeline, the data access (DA) stage is performed before the execution (EX) stage. Stages shown in *italics* font indicate that information is passed through the stage, but no action is taken. For instance, a conventional ALU instruction does not perform a data access. A *pam* ALU instruction performs a DTLB access and L1 DC tag check (TC stage) and updates the register file (WB stage) after the address is calculated (AG stage). The TC stage is performed immediately following the AG stage allowing the DTLB access and L1 DC tag check to be peformed in the following cycle. We use a distinct adder for the AG stage so that the address can be performed a cycle earlier to decrease stalls with a dependent load or store instruction. This same adder can also be used for branch target address calculations. The EX stage is not used for the address generation for load and store instructions since the address has already been generated by a *pam* instruction and memory references are only performed with a register indirect addressing mode. We will describe in Section 5 how to exploit the EX stage associated with a load or store to perform an ALU operation in addition to a memory access. The TC stage is not used for regular loads and stores as the DTLB access and L1 DC tag check are previously performed

Table 3. DAGDA Stages Used by Instructions

Instruction	Pipeline Stages					
ALU inst	IF	ID	RF	*DA*	EX	WB
pam ALU inst	IF	ID	RF	AG	TC	WB
load inst	IF	ID	RF	DA	*EX*	WB
pam load inst	IF	ID	RF	DA	TC	WB
store inst	IF	ID	RF	DA	*EX*	*WB*

Table 4. DAGDA Instruction Pipeline Example

Instruction	1	2	3	4	5	6	7	8	9	10
1. pam add	IF	ID	RF	**AG**	**TC**	WB				
2. other		IF	ID	RF	DA	EX	WB			
3. pam load			IF	ID	RF	**DA**	**TC**	WB		
4. other				IF	ID	RF	DA	EX	WB	
5. load					IF	ID	RF	**DA**	EX	WB

in a *pam* instruction. A *pam* load instruction will perform the TC stage after the address has been loaded from the L1 DC (DA stage). The write back (WB) stage is not used for a store instruction.

Table 4 depicts a sequence of instructions in a DAGDA in-order instruction pipeline. The AG, DA, and TC stages used in the example are depicted in **boldface** in the pipeline diagram. Instruction 1 calculates an address during the AG stage and performs a DTLB access and a tag check in the TC stage to determine the L1 DC way of the set where the desired data line resides. Instruction 3 loads an address value from the L1 DC. The DA is performed in cycle 6 and the L1 DC way can be forwarded from instruction 1, which is available at the end of cycle 5. Instruction 3 also performs the TC stage in cycle 7 since it is a *pam* load. Instruction 5 uses the loaded value to dereference memory in cycle 8. Note in this pipeline one instruction is required to be executed between the point that an address calculation is performed (instruction 1) or a pointer address is loaded from memory (instruction 3) and the point that the address is dereferenced (instructions 3 and 5). Scheduling at least one independent instructions between a *pam* instruction and the point of a load is often easier than the conventional problem of scheduling an independent instruction between a load and the use of a loaded value since effective address calculations typically do not have many dependences with other instructions. We show in the next section that in many cases the L1 DC way will be known after the AG stage of a *pam* instruction allowing the TC stage and potential hazards between *pam* and data access (load and store) instructions to be avoided. In the infrequent case when the L1 DC way is unknown after the AG stage and the *pam* instruction can only be scheduled immediately before the data access instruction, there are two options: (1) the pipeline could either stall the data access instruction for a cycle to allow the TC stage of the *pam* instruction to complete before the DA stage of the data access instruction or (2) all L1 DC data arrays could be accessed in parallel in the DA stage of the data access instruction with the L1 DC tag check in the TC stage of the *pam* instruction.

4 Memoizing L1 DC and DTLB Ways

The L1 DC way must be stored in a structure since the *pam* instruction and the corresponding data access instruction may be separated by many instructions, which would prevent forwarding the L1 DC way in the instruction pipeline. In fact, a DTLB access and L1 DC tag check will often be unnecessary since the same line may be accessed again. Figure 4(a) shows code for loading from and storing to the same variable. While the load needs a corresponding *pam* instruction, the store can use the same L1 DC way as the value of r6 has not been changed. Figure 4(b) shows an example of accessing sequential array locations. The *pam* instruction (r20=r20+4;) need not perform a DTLB access or L1 DC tag check when the *L1 DC block number* field of the L1 DC address (see Figure 3) is not updated.

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

(a) Redundant Address

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

(b) Strided Accesses

Figure 4. Memoization Examples

A simple and efficient approach to avoid redundant DTLB accesses and L1 DC tag checks is to associate information with the destination register number of a *pam* instruction and to detect when updates to this register do not invalidate this information. Consider the address generation structure (AGS) in Figure 5(a) that contains fields associated with each integer register used as an indirect address register in load and store instructions. The AGS structure could also be utilized for an OoO processor by associating the AGS entry information with each physical register. The *DWV* bit indicates if the *DTLB way* field is valid. The *DTLB way* field holds the DTLB way in which the associated physical page number resides. If the *DWV* bit is not set, then the rest of the AGS entry is considered invalid. The *LWV* bit indicates if the *L1 DC way* field is valid. The *L1 DC way* field holds the L1 DC way in which the associated cache line resides. By only allowing an indirect addressing mode, the L1 DC *set index* field (see Figure 3) of the register value indicates the L1 DC set and need not be stored in the AGS. The *PP* field contains page protection bits from the DTLB entry since the AGS structure allows DTLB references to be avoided. The AGS entry needs to be accessed during the RF stage to allow a data access (DA) for a load in the following cycle. Figure 5(b), which deals with the coherence issue of L1 DC evictions, will be described later in this section.

An address is often updated and still resides in the same cache line and more frequently in the same page. Figure 6 shows how to easily detect if the cache line to be accessed will change during a *pam* integer immediate addition (*rd* = *rs* + *immed*). First, the magnitude of the immediate has to

	DWV	DTLB way	LWV	L1 DC way	PP
0					
...					
31					

(a) Address Generation Structure (AGS)

0	
...	
n−1	

(b) Address Generation Valid Information (AGV)

Figure 5. Address Generation Information

be less than the *line offset* size. Second, the carry out values are inspected during the addition. If the *set index* field is updated, then the *L1 DC way* may no longer be accurate and the L1 DC tag arrays and a single way in the DTLB have to be accessed in the TC stage. If the *virtual page number* (*VPN*) field is updated, then the *DTLB way* may have also changed and all the ways in the DTLB have to be accessed. A *pam* integer add with registers can be handled in a similar manner as the magnitude of the register source values can be checked during the integer addition.

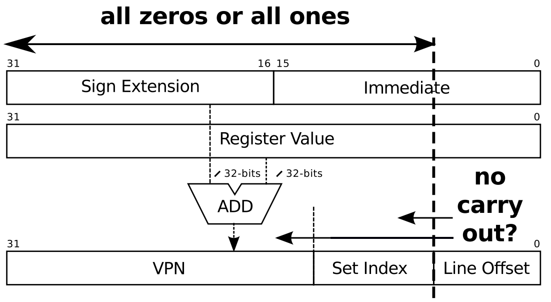


Figure 6. Detecting Address Changes

Conventional compiler optimizations, such as common subexpression elimination and loop-invariant code motion, are used to reduce the number of *pam* address generation calculations and the associated L1 DC tag checks and DTLB accesses. However, we also perform another optimization to avoid more L1 DC tag checks and DTLB accesses. Consider Figure 7(a) where two nearby addresses are dereferenced. Figure 7(b) shows that the second *pam* instruction can be expressed using the destination register of the first *pam* instruction. If the addition of -8 is still within the same line as the `r17` source value, then the L1 DC tag check and DTLB access can be avoided. Even if the two addresses do not reside in the same line, the associative DTLB access can be avoided if the addresses reside in the same page.

```
r2=sp+80;  [pam]      r17=sp+80;  [pam]
...                   ...
r2=M[r2];             r2=M[r17];
...                   ...
r2=sp+72;  [pam]      r17=r17+-8;  [pam]
...                   ...
r2=M[r2];             r2=M[r17];
```

(a) Original Insts (b) Updated Insts

Figure 7. Accessing Nearby Addresses

Figure 5(b) depicts the AGV structure that is used as one possible method to invalidate AGS entries when an L1 DC line is evicted or invalidated. Each entry in the structure contains a bit vector, where each bit represents an integer register. An entry is indexed by the *L1 DC way*, where *n* is the associativity level for the L1 DC. Each time an AGS entry shown in Figure 5(a) is associated with a line, the bit corresponding to the register number used in the indexed entry of the AGV structure is set. Each time the *LWV* bit (see Figure 5(a)) is cleared due to a non-*pam* instruction updating an integer register, the bit corresponding to that register is also cleared in every AGV entry. The AGV structure is read when an L1 DC line is replaced or invalidated and the corresponding bits set in the entry accessed by the *L1 DC way* of that line are used to determine which AGS entries will have their *LWV* bit cleared. Thus, this structure contains an inverse mapping between one L1 DC way and the AGS entries. Additional AGV entries in Figure 5(b) could be added by using the low-order bits of the *set index* field of the address shown in Figure 3 along with the L1 DC way to distinguish between different AGV entries. DTLB entries are less frequently replaced. All the *DWV* bits in the AGS structure and the values in the AGV structure are cleared upon a DTLB eviction.

5 Coalescing ALU Operations with Memory Data Accesses

We perform an ALU operation when possible in the instruction that encodes a DAGDA load or store data access to both decrease code size and improve performance. Supporting direct ALU memory operands is problematic for a 32-bit instruction set when nonzero displacements are allowed. We do not have these problems with DAGDA. Consider Figure 8(a) that shows the MIPS I format that is conventionally used to encode immediate instructions that include load and store data accesses. The 16-bit *immediate* field is no longer used in DAGDA load and store operations since a displacement addressing is not allowed. Thus, 16 bits are available to encode another operation. These ALU operations performed with loads and stores can be implemented without requiring an extra ALU in the processor. For a load operation, a funct field and either a register or a short immediate can be encoded in the available 16 bits and an ALU operation can be performed on the loaded value since the DA pipeline stage occurs a cycle before the EX pipeline stage is performed (see Table 3). Figure 8(b) shows how the MIPS R instruction format can be used to encode a load and a dependent operation that uses the loaded value. Figure 8(c) shows how a short immediate can be encoded where a dependent operation can follow a load. The figure also shows it is possible to update the register being dereferenced in a load or a store, which means a postincrement of this register could be performed in parallel with the memory operation. Note that a postincrement for a load requires that either a second write port would be needed for the integer register file or a buffer would have to be utilized to store one of the write operations until the integer register write port is free.

6	5	5	16
opcode	rs	rt	immediate

ex: rt=M[rs+immed]; # load

(a) Original MIPS I Format Used for Loads and Stores

6	5	5	5		6
opcode	rs	rt	rd		funct

ex: rd=M[rs]+rt; # load+addreg

(b) MIPS R Format Used with Loads

6	5	5	10	6
opcode	rs	rt	immediate	funct

```
ex: rt=M[rs]+immed;            # load+addimmed
ex: rt=M[rs]; rs=rs+immed;     # load+postincr
ex: M[rs]=rt; rs=rs+immed;     # store+postincr
```

(c) New Short Immediate Format Used with Loads and Stores

Figure 8. Encoding Loads and Stores with an ALU Operation

We schedule the *pam* instruction so that it can immediately follow the memory access when possible. Consider the loop in Figure 9(a) where the *pam* instruction is in the loop header at L2. Figure 9(b) shows the revised loop where the *pam* instruction is moved to both the preheader and the predecessor block within the loop. Because the *pam* instruction can immediately follow the memory reference that references the same register, the compiler is able to coalesce the *pam* instruction with the store instruction. Note the distance in instructions from the *pam* instruction to the memory reference that dereferences the *pam* register is increased. Scheduling *pam* instructions earlier has multiple advantages. (1) The number of instructions executed is decreased when the *pam* instruction can be coalesced with a memory reference. (2) The L1 DC tag check is more likely to be completed before the data access in the memory reference occurs. (3) If the L1 DC tag check does not find a matching tag, then the access to the next level of the memory hierarchy can be initiated earlier, which can reduce the effective L1 DC miss penalty.

```
    ...
    PC=L2;
L1: ...
    M[r7]=r3;
    ...
L2: ...
    r7=r7+4; [pam]
    PC=r7!=r8,L1;
```

(a) Original Loop

```
    ...
    r7=r7+4; [pam]
    PC=L2;
L1: ...
    M[r7]=r3; r7=r7+4; [pam]
    ...
L2: ...
    PC=r7!=r8,L1;
```

(b) After Transformation

Figure 9. Scheduling *pam* Instructions

6 Evaluation Framework

In this section we describe the experimental environment. We use 17 benchmarks shown in Table 5 from the MiBench benchmark suite [6], which is a representative set of embedded applications. All benchmarks are simulated using the large dataset option.

Table 5. Benchmarks Used

Category	Benchmarks
automotive	bitcount, qsort, susan
consumer	jpeg, tiff
network	dijkstra, patricia
office	ispell, stringsearch
security	blowfish, rijndael, pgp, sha
telecom	adpcm, CRC32, FFT, GSM

We used the VPO compiler [5] to annotate *pam* instructions and to perform the optimizations described in the paper. We generated code for a modified version of the MIPS instruction set that supports the ability to annotate *pam* instructions shown in Table 1. We used the ADL simulator [11] to execute both a baseline MIPS ISA and the ISA that supports both *pam* annotations and loads and stores that can be coalesced with ALU operations. We modified the ADL simulator to estimate the performance of a single issue in-order pipeline as described in the paper. Table 6 shows other processor configuration details that we utilized in our simulations.

Table 6. Processor Configuration

page size	8KB
L1 DC	32KB size, 4 way associative, 1 cycle hit, 10 cycle miss penalty
DTLB	32 entries, fully associative

We used CACTI to estimate L1 DC and DTLB energy usage assuming 22-nm CMOS process technology with low standby power (LSTP). Table 7 shows the energy required for accessing various components of the L1 DC and DTLB. We estimated the energy usage for a one-way L1 DC data array read to be one fourth of the energy required to simultaneously read four L1 DC data arrays.

Table 7. Energy for L1 DC and DTLB Components

Component	**Energy**
Read L1 DC Tags - All Ways	0.782 pJ
Read L1 DC Data - All Ways	8.236 pJ
Write L1 DC Data - One Way	1.645 pJ
Read L1 DC Data - One Way	2.059 pJ
Read DTLB - Fully Associative	0.823 pJ
Read DTLB - One Way	0.215 pJ
Write AGS - 1 Entry	0.320 pJ
Read AGS - 1 Entry	0.147 pJ
Write AGV - 1 Bit in All 4 Entries	0.240 pJ
Read AGV - 32 Bits in All 4 Entries	0.500 pJ

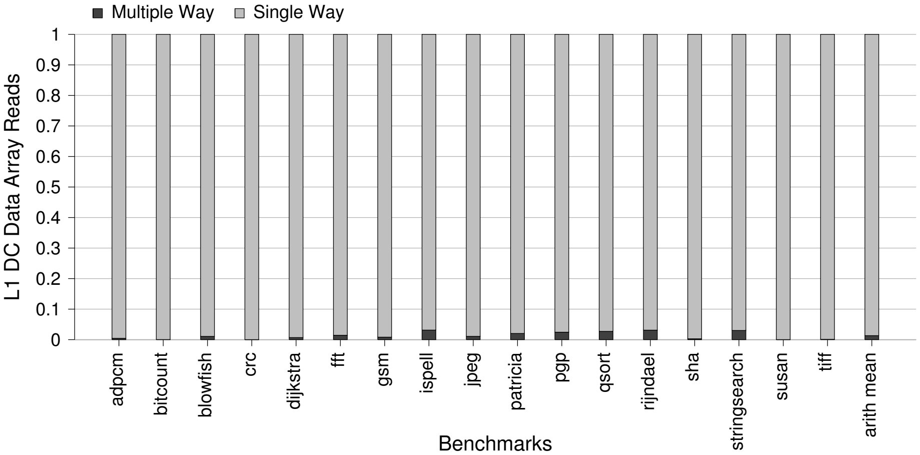


Figure 10. Load L1 DC Single and Multiple Way Data Array Accesses

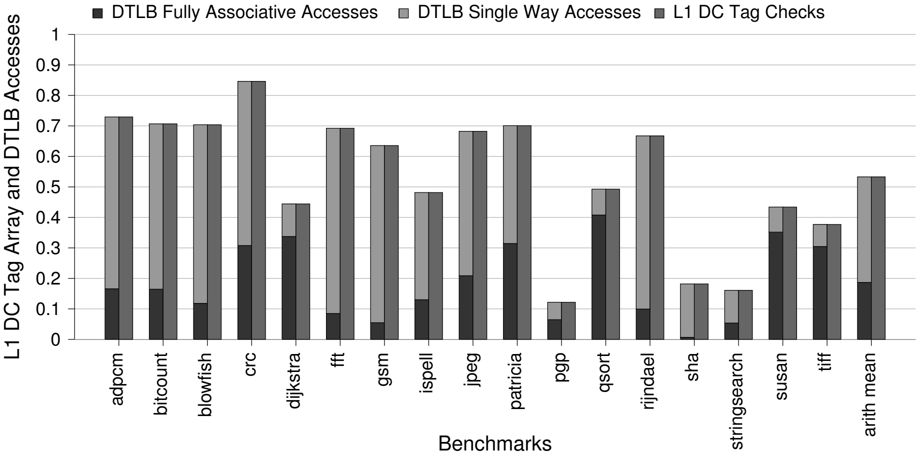


Figure 11. L1 DC Tag Checks and DTLB Access Ratio

7 Results

Figure 10 shows the ratio of accessing all L1 DC data arrays to a single L1 DC data array for load instructions. All data arrays are only accessed when the AGS entry is not marked as valid, which could occur for two reasons. First, the compiler sometimes cannot identify the *pam* instruction, which can occur when the last instruction that sets the register being dereferenced is passed as a parameter or returned from a function. Second, the entry could be invalidated due to an L1 DC line eviction. Figure 5(b) shows that for each way in the L1 DC there is a bit for each integer register that is set when the AGS entry is associated with that L1 DC way. Whenever a line is replaced, all AGS entries having that same way are invalidated. Only 1.3% of the loads on average performed an associative data array access. These results show that the compiler is able to typically identify a *pam* instruction and L1 DC evictions do not cause many associative data array accesses for loads. Note that stores always access only a single L1 DC data array.

Figure 11 shows the ratio of L1 DC tag checks and fully associative DTLB accesses that are performed in DAGDA compared to a conventional processor. Only 53.3% of the memory references require an L1 DC tag check on average. Likewise, only 34.6% of the memory references require a fully associative DTLB access on average. These results illustrate that our memoization techniques are very effective at reducing the number of L1 DC tag checks and fully associative DTLB accesses.

Figure 12 shows the energy of accessing the DTLB, L1 DC, AGS, and AGV structures in DAGDA versus a conventional DTLB and L1 DC. The left bar for each benchmark shows the energy usage breakdown for the baseline, which always totals to 100%. The right bar for each benchmark shows the energy usage breakdown for DAGDA relative to the baseline. Static energy for all of the structures comprises less than 0.5% of the total energy on average for both the baseline and DAGDA. The biggest energy usage reduction comes from L1 DC data array reads in DAGDA, dropping from 73.6% to 19.0% on average. The L1 DC data array write energy usage is unchanged since stores always access only a

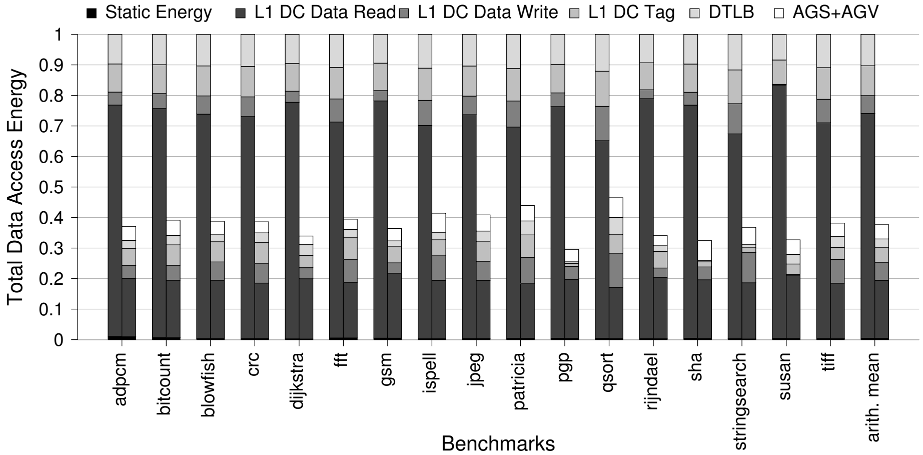


Figure 12. Data Access Energy Usage Ratio

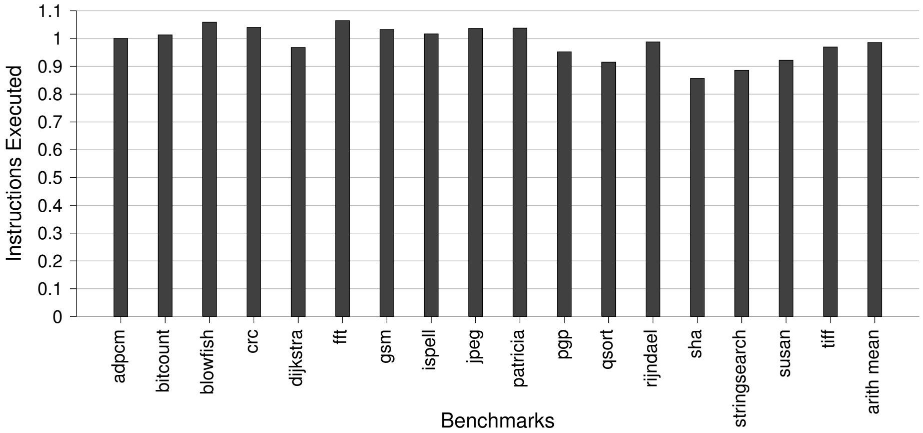


Figure 13. Instructions Executed Ratio

single L1 DC data array. The L1 DC tag array energy usage dropped from 9.8% to 5.0% on average as about 47% of the L1 DC tag checks are eliminated due to memoizing the L1 DC way in the AGS structure. Likewise, the DTLB energy usage dropped from 10.3% to 2.7% on average as 47% of the DTLB accesses are completely eliminated and 19% required accessing only a single DTLB way on average. The AGS and AGV structures required 4.7% additional energy usage on average as compared to the baseline. Overall, DAGDA provides on average a 62.4% reduction in total data access energy usage!

Figure 13 shows the ratio of instructions executed in DAGDA versus the baseline. Some additional instructions were executed in 9 of the 17 benchmarks due to decoupling the address generation and data access, which requires an additional calculation when the displacement of the data access was not zero. However, this increase in instructions executed was offset by being able to coalesce data access operations with ALU instructions. In addition, some of the additional instructions were loop invariant and were hoisted out of loops. On average the number of instructions executed decreased by over 1.4%.

Figure 14 shows the estimated cycles. The left bar for each benchmark shows the baseline and always totals to 100%. The baseline includes stalls between loads and the first instruction that references the loaded register, which only occurs when a reference is immediately after the load. The right bar for each benchmark shows the cycles using the DAGDA technique. Stalls between a *pam* instruction and loads and stores that reference the *pam* instruction destination register are depicted. Note that conventional load hazard stalls on L1 DC hits are not possible since the data access (DA) stage is performed earlier in the pipeline. Only 5 of the benchmarks with DAGDA increased the number of cycles executed. DAGDA provides on average about a 7.6% reduction in estimated cycles.

8 Related Work

Many techniques have been proposed to reduce energy usage in set-associative L1 DCs. Unlike our DAGDA approach, way-prediction techniques have a relatively high performance penalty of several percent [7, 12]. Nicolaescu et al. propose to save the way information of the last 16 cache accesses in

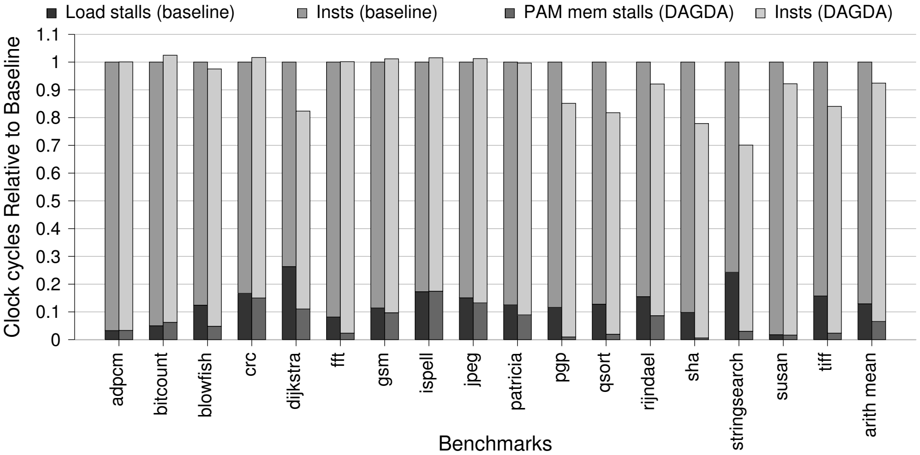


Figure 14. Estimated Performance Ratio

a table, and each memory access speculatively performs a fully associative tag search on this table. If there is a match, then the way information is used to activate only the corresponding way [10]. In contrast, our AGS structure is much smaller and only a single AGS entry is accessed for each memory reference. Way halting is another method for reducing the number of tag comparisons [16], where partial tags are stored in a fully associative memory (the halt tag array) with as many ways as there are sets in the cache. In parallel with decoding the word line address the partial tag is searched in the halt tag array. Only for the set where a partial tag match is detected can the word line be enabled by the word line decoder. This halts access to ways that cannot contain the data as determined by the partial tag comparison. Way halting requires a specialized SRAM implementation that might have a negative impact on the maximum operational frequency. An approach has been recently developed that allows way halting to be speculatively applied, but this technique only works when the displacement value in the memory reference is small and there is no carry out into the set index field of the address [9]. Way halting could be combined with our DAGDA approach to reduce energy usage even further.

There have also been some techniques proposed to avoid DTLB accesses. For example, opportunistic virtual caching is a technique to allow some blocks in the L1 caches to be cached with virtual addresses by changing the operating system to indicate which pages can use virtual caching [4]. In contrast, DAGDA can avoid many DTLB accesses by detecting that the physical page has not changed while requiring no operating system changes.

L1 DC tag checks for memory references are eliminated when the cache line to be accessed is identified by the compiler using direct address registers (DARs) [15]. The compiler annotates a memory reference that sets a DAR identifying the accessed L1 DC line and subsequent memory references that are guaranteed to access the same line reference the same DAR to avoid the tag check. Unlike DAGDA, several compiler transformations are required, such as loop unrolling and alignment of variables on cache line boundaries, to make these guarantees, which can result in both code and data size increases.

A tagless cache (TLC) design has been proposed that uses an extended TLB (ETLB) to avoid tag checks [13]. While the TLC approach can 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 access the L1 DC. The DAGDA approach could be used in conjunction with the TLC approach as the DTLB is accessed during the *pam* instruction and the L1 DC data array is accessed at least one cycle later. Unlike DAGDA, the TLC approach does not avoid TLB accesses. 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 proposed to reduce L1 DC energy usage. A line buffer can hold the last line accessed in the L1 DC [14]. The buffer must be checked before accessing the L1 DC, placing it on the critical path, which can degrade performance. 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. A small filter cache accessed before the L1 DC has been proposed to reduce the power dissipation of data accesses [8]. However, filter caches reduce energy usage at the expense of a significant performance penalty due to their high miss rate. This performance penalty mitigates some of the energy benefits of using a filter cache and has likely discouraged its use.

Like our AGS Method, the Tag Check Elision (TCE) approach stores an L1 DC way with each integer register [17]. Unlike TCE, DAGDA 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, which in their evaluation was a 29-bit value. TCE also does not schedule memory operations using *pam* instructions. In contrast, DAGDA requires no immediate value with AGS entries, which should require 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. DAGDA's check for a carry out of an addition into the *set index* field and *VPN* fields is much simpler. Unlike the TCE approach, DAGDA avoids accessing *n*-1 L1 DC data array accesses in an *n*-way set associative L1 DC even when the L1 DC way is unknown before the L1 DC tag check is performed by a *pam* instruction. Finally, TCE's invalidation scheme requires much more space than DAGDA's invalidation method.

There have also been techniques proposed to avoid associative L1 DC data array accesses. The speculative tag access (STA) approach speculatively performs an L1 DC tag check during the address generation stage when the displacement is small [1]. This approach fails when the addition of the displacement causes the *index* field of the effective address to change as compared to the same field in the base register value. Early load data dependence detection (ELD^3) has been proposed to allow the L1 DC tag check and the L1 DC data access to be sequentially performed when it is detected that the distance in instructions between the load and the first use of the loaded value is great enough avoid a stall [2]. A similar approach was also applied at compile time by using context-aware loads and stores [3]. DAGDA is able to avoid more associative L1 DC data array accesses as well as avoiding L1 DC tag checks and DTLB accesses.

9 Future Work

There are several configuration changes we can investigate using DAGDA. First, larger L1 DC cache lines should lead to fewer L1 DC tag checks due to the *L1 DC set index* field being less frequently updated each time a *pam* increment is executed. Likewise, a higher miss penalty for larger L1 DC lines may be offset by *pam* instructions initiating the L1 DC line prefetch before the load data access is performed. Second, a higher associative L1 DC with DAGDA should likely decrease energy usage as the power to access a single L1 DC data array should be reduced. As shown in Section 7, the common case in DAGDA is that only a single L1 DC data array is accessed on loads. Third, the DAGDA approach could be evaluated in the context of an out-of-order (OoO) processor. Reducing data access energy usage is still likely in an OoO processor since the DAGDA approach should still result in fewer L1 DC data array accesses, fewer L1 DC tag checks, and fewer DTLB accesses.

10 Conclusions

DAGDA reduces energy usage for memory accesses by decoupling the address generation and the data access into separate instructions. By associating the DTLB access and L1 DC tag check with address generation instructions, we are able to typically access a single L1 DC data array for loads. We are also able to avoid many DTLB accesses and L1 DC tag checks by associating the DTLB way and L1 DC way with the register that holds the memory address to be dereferenced. Finally, we show that performance is improved due to merging the address generation with another instruction when the displacement is zero, applying conventional compiler optimizations to eliminate redundant address generation instructions, coalescing ALU operations with loads and stores, and prefetching L1 DC cache lines when a *pam* instruction detects an L1 DC miss.

Acknowledgments

We thank the anonymous reviewers for their comments that helped to improve this paper. This work was supported in part by the US National Science Foundation (NSF) under grants DUE-1241525, DUE-1259462, IIA-1358147, CCF-1533846, CCF-1533828, DGE-1565215, and DRL-1640039. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

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 *Proceedings of the IEEE International Conference on Computer Design (ICCD 2013)*, Oct. 2013.

[2] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. Reducing set-associative l1 data cache energy by early load data dependence detection (eld3). In *IEEE/ACM Design Automation and Test in Europe Conference*, March 2014.

[3] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors. Improving data access efficiency by using context-aware loads and stores. In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2015.

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

[5] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *Proc. 34th ACM/IEEE Int. Symp. on Microarchitecture (MICRO)*, pages 124–133, Dec. 2001.

[16] 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.

[17] 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.