Improving Data Access Efficiency by Using a Tagless Access Buffer (TAB)

Alen Bardizbanyan
Chalmers University of Technology
alenb@chalmers.se

Peter Gavin David Whalley
Magnus Själander
Florida State University
[gavin/whalley/sjalande]@cs.fsu.edu

Per Larsson-Edefors
Sally McKee Per Stenström
Chalmers University of Technology
[perla/mckee/pers]@chalmers.se

Abstract

The need for energy efficiency continues to grow for many classes of processors, including those for which performance remains vital. Data cache is crucial for good performance, but it also represents a significant portion of the processor's energy expenditure. We describe the implementation and use of a *tagless access buffer* (TAB) that greatly improves data access energy efficiency while slightly improving performance. The compiler recognizes memory reference patterns within loops and allocates these references to a TAB. This combined hardware/software approach reduces energy usage by (1) replacing many level-one data cache (L1D) accesses with accesses to the smaller, more power-efficient TAB; (2) removing the need to perform tag checks or data translation lookaside buffer (DTLB) lookups for TAB accesses; and (3) reducing DTLB lookups when transferring data between the L1D and the TAB. Accesses to the TAB occur earlier in the pipeline, and data lines are prefetched from lower memory levels, which result in a small performance improvement. In addition, we can avoid many unnecessary block transfers between other memory hierarchy levels by characterizing how data in the TAB are used. With a combined size equal to that of a conventional 32-entry register file, a four-entry TAB eliminates 40% of L1D accesses and 42% of DTLB accesses, on average. This configuration reduces data-access related energy by 35% while simultaneously decreasing execution time by 3%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Hardware/Software co-design, Energy efficiency

Keywords energy, memory hierarchy, strided access

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

1. Introduction

Energy efficiency has become a first-order design constraint for everything from battery-powered, mobile devices that execute digital signal processing applications or audio/video codecs up to exascale systems that run large-data scientific computing applications. Multicore scaling has been used to improve energy efficiency, but a recent study estimates that multicore scaling will soon be power-limited [7]. We therefore need other ways to address energy efficiency.

A significant portion of a processor's power is dissipated in the data caches. The level-one data cache (L1D) uses up to 25% of the total power of a processor [6, 9]. The processor's overall power is increased by inefficiencies in the data memory hierarchy such as unnecessary cache line transfers and stall cycles caused by cache misses. Even the way the data cache is accessed can cause many stall cycles, resulting in decreased performance and increased power. Finally, the cost of accessing data includes the power to access the data translation lookaside buffer (DTLB) for converting virtual addresses to physical addresses.

In order to reduce the power dissipation of data caches and DTLBs without degrading execution time, we propose an approach that requires few and simple enhancements to conventional processor architectures and exposes control of these enhancements to the compiler. The compiler recognizes memory references accessed with constant strides or loop-invariant addresses and explicitly redirects these accesses to a small, power-efficient memory structure called a *tagless access buffer* (TAB).

This paper makes the following contributions. (1) We significantly reduce L1D energy by capturing many data memory references in the TAB. Furthermore, the compiler's explicit allocation of data accesses to the TAB obviates the need for tag checks or DTLB accesses. (2) We slightly improve performance by prefetching cache lines into the TAB, partially offsetting the L1D miss penalty, and by accessing the TAB earlier in the pipeline, avoiding load hazards. (3) We exploit amenable access patterns of the TAB-allocated memory references to eliminate unnecessary data transfers between memory hierarchy levels. We are able to make these

contributions with small instruction set architecture (ISA) changes that exploit a structure the size of a conventional 32-entry register file.

2. The TAB Approach

Fig. 1 shows the TAB's position in the memory hierarchy. Load and store instructions transfer conventional data access sizes between the L1D or the TAB and the register file, while entire cache lines are transferred between the L1D and the TAB. For each line transferred to the TAB, multiple L1D references are replaced with references to the more energy-efficient TAB. Data cache lines stored in the TAB are inclusive to the L1D, which simplifies writeback operations and TAB integration in a cache-coherent system.

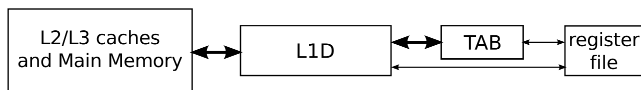


Figure 1. Memory hierarchy organization with TAB.

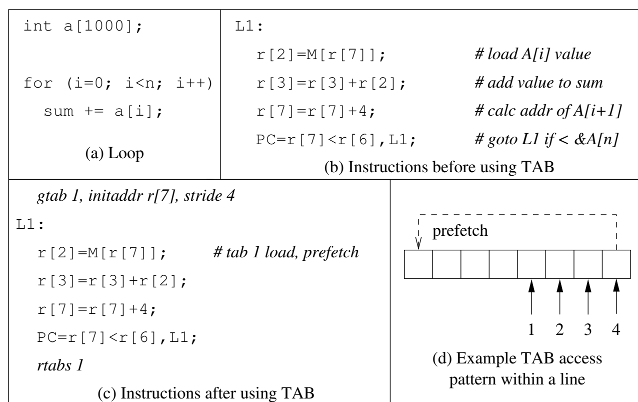


Figure 2. TAB allocation for a constant stride reference.

Fig. 2 gives a high-level illustration of how a TAB is used. When the compiler detects a reference with a constant stride or an invariant address within a loop, it changes it to access the value from the TAB instead of the L1D. Fig. 2(a) depicts a loop that sums the elements of an array, and Fig. 2(b) depicts the instructions generated from that loop. Our compiler detects that the memory reference (`r[2] = M[r[7]];`) is accessed with a constant stride of four (`r[7] = r[7] + 4;`) and replaces the memory access with a TAB access. In addition, the compiler explicitly allocates a TAB entry before the loop by inserting a "get TAB" (*gtab*) instruction. This instruction prefetches the L1D cache line referenced by the initial value of `r[7]` into the TAB. The compiler deallocates the TAB entry after the loop by inserting a "release TABs" (*rtabs*) instruction. The resulting code is depicted in Fig. 2(c). Fig. 2(d) shows an example access pattern for a cache line residing in the TAB. The initial address does not need to be aligned to the line boundary. During the TAB access, the hardware checks whether the next access crosses the cache line boundary. After the fourth access a prefetch is initiated, the next L1D line is loaded into the TAB, and the next TAB reference accesses the next sequential line of data. If dirty, the current line is written back before the new line is prefetched. A writeback also happens when a TAB entry containing a dirty line is released by the *rtabs* instruction.

3. Compiler Analysis

Performing compiler analysis to allocate memory references to the TAB is relatively straightforward, as there is no need for interprocedural analysis or additional code transformations. The compiler needs to know the minimum number of TAB line buffers supported by an architecture family to limit the allocation of TAB entries within a loop nest. Apart from that, the compiler needs to know the minimum L1D line size to ensure the correctness of TAB operations when multiple distinct references are associated with a single TAB entry.

The compiler detects memory references with a constant stride or a loop-invariant address and allocates a distinct TAB entry for each such reference by inserting a *gtab* instruction in the loop preheader. It deallocates all TAB entries for the loop by inserting a single *rtabs* instruction in the loop postheader. The compiler creates preheader and postheader blocks for the loop associated with a TAB if they do not already exist, and it inserts additional instructions to calculate the memory address of the first TAB access before the *gtab* instruction, if needed. Since these instructions are only added before and after the loop, they have little impact on the execution time. Load/store instructions to locations allocated to the TAB are annotated to reference a specific TAB entry and to indicate whether or not accesses should prefetch the next line. Fig. 3 gives a high-level description of the compiler analysis algorithm to allocate TAB entries.

```
FOR each loop in function in order of innermost first DO
   FOR each load and store in the loop DO
      IF reference has a constant stride OR
         reference is a loop−invariant address THEN
         IF reference has offset so that it can be added
            to an existing TAB for the loop THEN
               Merge the reference with the existing TAB;
         ELSE
               Put the reference in a new TAB;
   IF too many TABs THEN
      Select the TABs with the most estimated references;
FOR each TAB in the function DO
   Generate a gtab inst in the loop preheader for the TAB;
   Annotate each load or store associated with the TAB;
FOR each loop in the function DO
   IF TABs are associated with the loop THEN
      Generate a rtabs inst in the loop postheader;
```

Figure 3. Compiler analysis algorithm for TAB allocation.

3.1 References with Constant Strides

Most compilers perform loop strength reduction, which converts memory references to the form *M[reg]*, where *reg* is a register holding a basic induction variable that is only updated by assignments of the form *reg* = *reg* ± *constant*. Note that sometimes a memory reference could be of the form *M[reg+disp]*, where *disp* represents a displacement from *reg*. Detecting a memory reference with a constant stride should thus be straightforward, since the compiler only needs to detect that the register used in a memory reference is a basic induction variable. There is one small complication, as loop strength reduction requires allocating a register, and too few registers may be available to perform this transformation on every strided reference. Nonetheless, our compiler still performs analysis to detect that the reference address changes by a constant value on each iteration so that the memory reference can be allocated to the TAB.

The requirements for the compiler to allocate a single strided reference to the TAB are as follows. (1) Each reference must be within a loop and must have a constant stride. (2) Each reference must be in a basic block that is executed exactly once per iteration (due to the prefetches initiated by TAB references). (3) The stride must be smaller than the TAB line buffer so that multiple references are accessed before another L1D line is prefetched (this is important for energy-efficiency). Strides can be positive (increasing addresses) or negative (decreasing addresses).

A single TAB entry can also be used by a group of strided references having more than one distinct address. This allows the compiler to allocate more memory references that are accessed through the TAB, even though the number of distinct TAB entries is limited. The basic requirements for allocating a group of references to a single TAB entry are as follows. (1) All references in the group must be in the same loop. (2) Each reference must have the same constant stride within that loop. (3) The reference causing the prefetch must be executed exactly once per loop iteration. (4) The absolute value of the stride should be no larger than the L1D line size. (5) The maximum distance between the addresses of any two distinct references must be less than the L1D line size. If the multiple references to a particular strided access pattern can span two consecutive L1D cache lines from memory, we allocate an extra TAB line buffer. The compiler recognizes this situation and appropriately allocates two TAB line buffers. The TAB then prefetches two lines from the L1D upon executing the associated *gtab* instruction.

Sometimes multiple memory references with distinct addresses can be allocated to the TAB without requiring an extra line buffer. If the references are accessed in order with respect to the direction of the stride, the distance between each reference is the same, and the distance between the last reference in one loop iteration and the first reference in the next is the same, then we can use a single TAB line buffer. Fig. 4 illustrates an example of multiple memory references accessed with a single TAB line buffer. Fig. 4(a) depicts code that sums the elements of array *A*. Fig. 4(b) depicts the loop after it has been unrolled to improve performance. Fig. 4(c) shows the instructions generated. In this example, the alignment of array *A* is unknown, so each TAB access checks if a prefetch operation is required.

```
int A[n];

for(i=0; i < n; i++)
    sum+=A[i];
```

(a) Original loop

```
int A[n];

for(i=0; i < n; i++){
    sum+=A[i];
    sum+=A[i+1];
    sum+=A[i+2];
    sum+=A[i+3];
}
```

(b) Loop after unrolling

```
      gtab 1, initaddr r[6]-12, stride 4
L1:
      r[2]=M[r[6]-12];#tab 1 load, prefetch
      r[2]=M[r[6]-8]; #tab 1 load, prefetch
      r[2]=M[r[6]-4]; #tab 1 load, prefetch
      r[2]=M[r[6]];   #tab 1 load, prefetch
      r[7]=r[7]+r[2];
      r[7]=r[7]+r[3];
      r[7]=r[7]+r[4];
      r[7]=r[7]+r[5];
      r[6]=r[6]+16;
      PC=r[6]<r[8],L1;
      rtabs 1
```

(c) Instructions after using TAB

Figure 4. Grouping references into a single TAB line buffer.

3.2 References with Loop-Invariant Addresses

Memory references with an invariant address are also good candidates for accessing via the TAB. Although they are not as frequent as strided accesses, the benefit of accessing them in the TAB is high, as they require at most one prefetch from the L1D before the loop and one writeback after the loop. Fig. 5 illustrates an example of allocating a memory reference with a loop-invariant address to the TAB. Fig. 5(a) shows a loop in which a global variable *sum* cannot be allocated to a register due to the *scanf()* function call. Fig. 5(b) shows the corresponding loop in which the memory operations for *sum* become TAB references. The reference to the variable *n* is not allocated to the TAB because its address is passed to another function. To improve performance, the compiler attempts to minimize the number of such regular load/store operations that interfere with TAB references.

```
/* sum is a global variable */
sum = 0;
while (scanf("%d", &n))
   sum += n;
```

(a) Original loop

```
    r[14]=sum;
    gtab 1, initaddr r[14], stride 0,
            writefirst
    M[r[14]]=0;         # tab 1 store
    PC=L4;
L2: r[3]=M[r[29]+n];
    r[5]=M[r[14]];      # tab 1 load
    r[5]=r[5]+r[3];
    M[r[14]]=r[5];      # tab 1 store
L4: • • •
    r[5]=r[29]+n;
    ST=scanf;
    PC=r[2]!=r[0],L2;
    rtabs 1
```

(b) Instructions after using TAB

Figure 5. TAB usage with a loop-invariant memory address.

3.3 Minimizing Interferences with Loads and Stores

Very few regular load/store operations are redirected to the TAB. However, sometimes false interferences occur due to a

variable sharing the same cache line with a different variable assigned to the TAB. In order to minimize the frequency of false interferences, the compiler detects when the range of memory references allocated to a TAB cannot possibly be accessed by regular loads/stores in the same loop. When the compiler can guarantee that the TAB accesses are distinct from regular memory references, or both the TAB accesses and the regular memory references are loads, then the *gtab* instruction is annotated to indicate that the TAB entry is guaranteed not to interfere with regular memory references.

3.4 TAB Allocation Heuristics

The compiler initially detects all the possible memory references within a loop that can be allocated to the TAB. For many loop nests, the number of TAB entries that could potentially be allocated exceeds the number of physical TAB entries. In these situations the compiler must determine which TAB entries are the most profitable to retain. This calculation is performed by estimating the average number of L1D accesses that will be avoided on each iteration rather than for the entire loop, as the number of loop iterations is often unknown at compile time. Initially each TAB reference is assigned a reference value of one. If the TAB reference is in conditionally executed code, then the compiler divides the number of references by two. If the TAB reference occurs within an inner loop, then its value is increased to reflect that it will be accessed multiple times for each iteration of the current loop. The compiler then sums the estimated number of references for all memory accesses associated with each TAB entry and subtracts the fraction of times that the reference will cause an L1D access due to a prefetch or a write back (determined by dividing the stride by the L1D line size). The compiler also divides the result by two if an extra TAB line buffer needs to be allocated.

Fig. 6 illustrates how the compiler selects which potential TAB entries to retain. The code in Fig. 6(a) has different types of arrays, and the base type affects the size of the stride between elements. Fig. 6(b) shows how the number of avoided L1D accesses per loop iteration is calculated. Assume for this example that the L1D line size is 32 bytes. The variable *d* is referenced twice in each loop iteration. The estimated number of references per loop iteration for the variable *a* is 1.5, since the second reference to *a* is in conditionally executed code. The number of TAB line buffers required for *a* is two because the last reference is in conditionally executed code and prefetches have to be associated with references that are executed on every loop iteration. The stride for *g* is zero because the scalar is accessed with a loop-invariant address. Note that the value of the global variable *g* could not be moved out of the loop due to the call to *f()*. Since there is no prefetching of *g*, there are no L1D accesses for *g* during loop execution. The variable *s* requires no L1D loads since it is only written. Avoiding unnecessary L1D loads is described later in the paper. The variable *b* requires no L1D writebacks because it is never updated. The number of saved L1D accesses per iteration can thus be calculated as:

$$(est_refs - (L1D_loads + L1D_writes))/\#TAB_lines \quad (1)$$

If there are only four TAB line buffers available, then the compiler will not allocate a TAB entry for variable *a*, since it avoids the fewest L1D accesses per loop iteration.

```
/* global declaration */
double g;
...
/* local declarations */
char s[100];
short b[100];
int a[100], i, sum, t;
double d[100];
...
/* loop referencing vars */
for (i=0; i < n; i++) {
   s[i] = 0;
   sum += b[i];
   if ((t = a[i]) < 0)
      a[i] = -t;
   d[i] = d[i] + g;
   f();
}
```

(a) Code

TAB var	estim refs	# TAB lines	stride	L1D loads	L1D writes	avoided L1D accesses
d	2.0	1	8	8/32	8/32	1.5 = (2.0-(0.25+0.25))/1
g	1.0	1	0	0	0	1.0 = (1.0-(0+0))/1
s	1.0	1	1	0	1/32	0.96875 = (1.0-(0+0.03125))/1
b	1.0	1	2	2/32	0	0.9375 = (1.0-(0.0625+0))/1
a	1.5	2	4	4/32	4/32	0.625 = (1.5-(0.125+0.125))/2

(b) Calculation of avoided L1D accesses

Figure 6. Example of estimating saved L1D accesses.

4. Hardware Support for TAB Operations

This section describes the necessary hardware and proposed ISA changes to support TAB operations.

4.1 TAB Organization

The TAB itself consists of a *TAB top* pointer, an *index* array, *line buffers*, and metadata, as illustrated in Fig. 7. Each *line buffer* holds the data of a TAB entry and is the same size as an L1D line. The metadata of a TAB entry is associated either with the TAB *index* or with the TAB *line buffer*.

The metadata associated with the *index* includes the *stride*, a few bits of *type info*, and *extra line* and *TAB valid* bits. The *stride* determines if a prefetch needs to be performed after a TAB load/store operation. The *type info* controls how data are transferred to and from the TAB and is described in Sec. 6.1. The *extra line* bit indicates if the TAB entry is associated with two line buffers instead of one. If set, then the least significant bit of the L1D set index field within the data address is used to select which of the two line buffers to access. If the *TAB valid* bit for a corresponding TAB entry is clear, then the TAB access is treated as a conventional load/store instruction accessing the L1D.

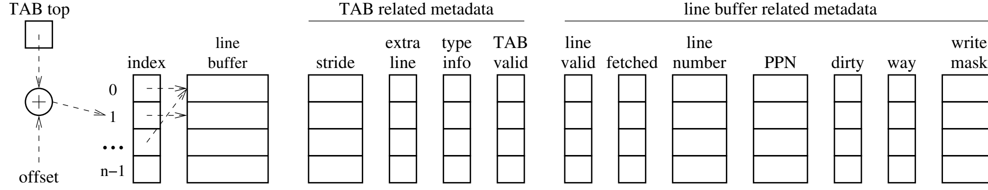


Figure 7. TAB organization.

The metadata associated with the *line buffer* includes *fetched*, *line valid*, and *dirty* bits; a *line number*; the L1D *way*; the physical page number (*PPN*); and a *write mask*. The *fetched* bit indicates whether or not the cache line has been transferred from the L1D. Since an L1D line can become invalid due to an eviction, the *line valid* bit indicates if valid data still reside in the TAB line buffer. The *line number* is used to calculate the next prefetch address. The *line number* combined with the *way* information indicate the exact position of the TAB line buffer's contents inside the L1D so that the TAB can intercept regular load/store operations to this data. This position information is also used during TAB writebacks to the L1D, allowing the operation to occur without checking the L1D tags and reducing writeback energy overhead. The *PPN* is concatenated with the next sequential line number when prefetching a line from the L1D. A DTLB access is only needed when the TAB entry is acquired (for the initial L1D line prefetch) and when prefetches cross a page boundary, which infrequently occurs. The *write mask* makes it possible to transfer only the dirty bytes of the line back to the L1D. The *dirty* bit is set whenever one or more of the write mask bits is set.

4.2 ISA Modifications

Fig. 8 illustrates instruction formats on a MIPS-like 32-bit instruction set. Fig. 8(a) shows the original format of MIPS load/store instructions, and Fig. 8(b) shows the proposed modifications to the immediate field in order to support TAB accesses. The bit fields used to control TAB accesses replace the most significant bits of the immediate field of the modified instructions, which limits the maximum positive and the minimum negative immediate values that can be represented. As a result, the compiler must insert additional instructions to use very high positive or very low negative address offset, but such values rarely occur. An advantage of this encoding is that a TAB operation can be forwarded to the L1D as a regular load/store instruction when the TAB is marked as invalid. The *T* bit identifies whether the instruction is a TAB operation or a regular load/store operation. The *P* bit indicates if the TAB access can trigger a prefetch operation. The *TAB offset* is subtracted from the *TAB top* to indicate which *index* is associated with a particular TAB access.

Fig. 8(c) and Fig. 8(d) show the instruction formats used to acquire and release TAB entries, respectively. Fig. 8(c) shows the proposed get TAB (*gtab*) instruction to allocate a TAB index and prefetch the first L1D line. The initial ad-

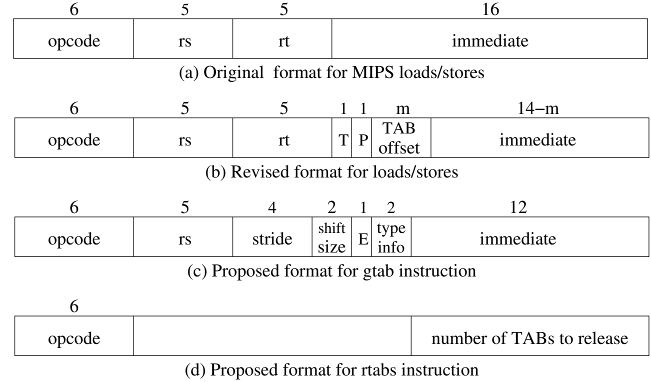


Figure 8. ISA modifications to support TAB accesses.

dress is calculated similar to regular load/store operations by adding the contents of the register *rs* and the *immediate* offset. The stride is encoded by two separate fields, the base of the *stride* (a signed integer) and the *shift size*. The actual stride is calculated as *stride* $<<$ *shift size*. This encoding enables more strides to be represented, that is, those that are an integer multiple of the data access size. The *E* bit indicates that an extra line buffer is to be allocated because that TAB data is being accessed by multiple references with distinct addresses. The *type info* bits are used to avoid unnecessary data transfers between memory hierarchy levels and to avoid false interferences with regular load/store operations. Fig. 8(d) shows the proposed release TABs (*rtabs*) instruction, which indicates the number of TAB entries to be released.

5. TAB Operations

This section describes the main TAB operations: allocation, deallocation, and prefetching. In addition, the operation to keep the coherency between L1D and TAB is explained.

5.1 TAB Allocation and Deallocation

The *TAB top* pointer initially points to the first TAB entry. TAB entries are allocated and deallocated in LIFO order. Each *gtab* operation increments the *TAB top* pointer by one or two — depending on whether the *E* (extra line) bit is set — and allocates the corresponding line buffer(s). During load/store operations from the TAB, the *TAB offset* (shown in Fig. 8(b)) is subtracted from the *TAB top*, and the resulting *index* value is used to access the correct line buffer. If the number of allocated TAB entries exceeds the total num-

ber available, then some entries will be overwritten. Each TAB entry is marked invalid when deallocated. With this approach, the compiler need not keep track of TAB allocations between function calls, which eliminates the need for interprocedural analysis. If a TAB entry is overwritten (or marked invalid due to a context switch), then the entry will be invalid when it is next referenced. As a result, subsequent accesses to this TAB entry access the L1D instead.

Fig. 9 gives an example of TAB allocation and deallocation. Assume that there are four physical TAB entries. Before entering the loop in *func1*, two TAB entries (*a*,*b*) are allocated. The first reference is to TAB *a*, and then *func2* is called. Before entering the loop in this function, three TAB entries (*c*,*d*,*e*) are allocated. The allocation of TAB *e* overwrites TAB *a*. If the *line buffer* associated with TAB *a* is dirty, then the dirty bytes are written back before the new line is fetched for TAB *e*. After exiting the loop, the three TAB entries (*c*,*d*,*e*) are deallocated. After returning from *func2*, the TAB *b* reference is still valid, but the TAB *a* reference is invalid, as the TAB *valid* bit was cleared when TAB *e* was released. Subsequent references to TAB *a* for the remaining loop iterations instead access the L1D. This example also illustrates why invalid TAB misses should occur infrequently. The loop inside *func1* repeats *m* times, and the loop inside *func2* repeats *m*n* times. All the references inside the *func2* loop will access the TAB, and only *m-1* references of TAB *a* will access the L1D.

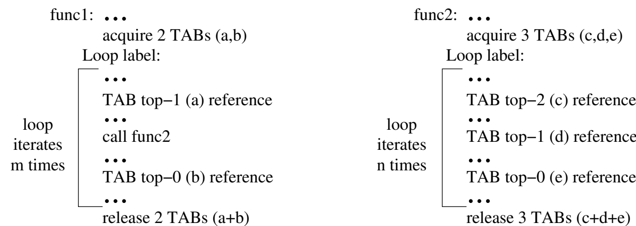


Figure 9. Example of using LIFO TAB allocation.

5.2 Prefetching into the TAB

The first prefetch for a TAB occurs when a *gtab* instruction allocates a TAB entry. When the prefetch bit of a TAB load/store instruction is set, a prefetch operation is initiated whenever there is a carry out from the sum of the *stride* and the line offset of the memory address (indicating that the next TAB reference will cross the line boundary). If dirty, the current line is written back to L1D, and the next line is prefetched by adding or subtracting one from the *line number*, depending on the sign of the stride. The *PPN* returned from the DTLB is set during this initial prefetch. DTLB accesses are unneeded during prefetches as long as the line to be fetched does not cross a page boundary. If a TAB entry uses two line buffers (extra line), the prefetch operation is initiated for the line not currently being accessed.

5.3 Intercepting Regular Loads and Stores

Sometimes regular load/store instructions access an L1D line that resides in the TAB. For example, a global variable that is accessed through the TAB in a loop may also be accessed as a regular load/store operation by another function called within that loop. In order to keep the TAB and L1D coherent, regular load/store instructions referencing such a line must be redirected to the TAB (because updates in the TAB line buffers are not reflected in the L1D until the line buffer is released). We therefore add an intercept (*I*) bit to each L1D line, and we set this bit when the line is loaded into the TAB. If this bit is found to be set during the L1D tag check of a regular load/store then the data is read from that TAB line buffer on the next cycle. To prevent false interferences, the compiler detects when the *I* bit need not be set.

Apart from maintaining consistency between the TAB and L1D, the inclusion property must be enforced during L1D line evictions. Since the *I* bit may be clear to avoid false interferences, we add a *T* bit to each L1D line that is always set when the line resides in the TAB. Consistency must also be maintained across TABs in a multicore system. Invalidation requests from other cores access the TAB only when a block is present in the L1D and the *T* bit is set. The TAB is small, and so it is feasible to check all *line numbers* and *ways* of the TAB entries with manageable overhead during L1D evictions or invalidation requests.

6. Avoiding Unnecessary Data Transfers

A conventional data memory hierarchy will always transfer a block of data from the next lower level on a miss, yet many of these transfers are actually unnecessary. Sometimes each byte of a cache line is written before ever being read. Since the TAB gives the compiler some level of explicit control over the memory hierarchy, these specific conditions can often be recognized and exploited. Unnecessary memory traffic can be reduced between the TAB, the L1D, and the next level in the memory hierarchy.

Fig. 10 shows potential data transfer scenarios in the data memory hierarchy using the TAB. Fig. 10(a) and Fig. 10(b) show conventional usage of the TAB with a dirty bit that applies to any memory hierarchy level with a writeback policy. Fig. 10(c), and Fig. 10(d) illustrate the memory hierarchy optimizations that the TAB enables. These optimizations are explained in more detail in the following sections.

6.1 Write-First and Write-First Contiguous

A compiler can detect memory access sequences in which a variable is written before it is read. In Fig. 11(a), for the given loop constraints, every second element of array *A* is written. In Fig. 11(b), all elements of array *A* are written. For both examples, when a line containing part of array *A* is associated with the TAB, the L1D line need not be read, as the TAB references are guaranteed not to interfere with

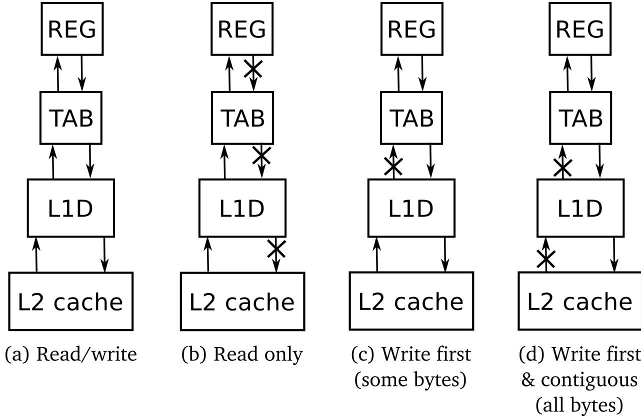


Figure 10. Exploiting the TAB to avoid L1D and L2 accesses.

regular loads or stores, and the memory locations referenced through the TAB accesses will be written first. Avoiding reading an entire line from L1D and writing it to the TAB improves energy efficiency.

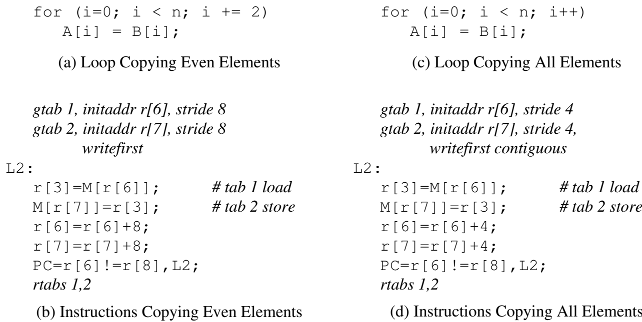


Figure 11. Examples where an array is written first.

When all bytes in the TAB line buffer are not overwritten (see array *A* in Fig. 11(a)), the cache line is prefetched into the L1D if not already present; upon release, the line buffer's *write mask* will be used to merge the TAB data with the remaining bytes in the line (shown in Fig. 10(c)). The overhead of keeping this mask for each line is negligible, and we save power during writebacks by only transferring the dirty bytes.

When all bytes of a line are written, which can occur for array *A* in Fig. 11(b), not only do we avoid prefetching the L1D line to the TAB, but we need not load the L1D line from the next level of the memory hierarchy if it is not already present, as shown in Fig. 10(d). Only a tag allocation is required in the L1D since the entire cache line will be overwritten by TAB stores. This reduces the number of data transfers from the next memory hierarchy level and further improves energy efficiency and performance by avoiding L1D misses. Note that when a write-first contiguous line in the TAB is only partially overwritten, then when released by an *rtabs* instruction that line must first be loaded into the L1D from the next memory hierarchy level so that its data are merged in the TAB writeback.

Sometimes a loop-invariant address can also be determined to be write-first. Consider again the example in Fig. 5(b). By performing the *gtab* before the store preceding the loop, the compiler guarantees that *sum* is write-first in the TAB, which allows us to avoid transferring the L1D line containing *sum*.

7. Evaluation Framework

We evaluate the proposed TAB approach using 20 benchmarks spanning six categories from the MiBench benchmark suite [8], as shown in Table 1. We use the large dataset inputs and compile the applications with the VPO compiler [4] to generate MIPS/PISA target code. The compiler automatically performs all analysis and transformations on the original application code. The simulation environment consists of the SimpleScalar simulator [2] with Wattch extensions [5] for power estimation based on CACTI [16]. We model a 100-nm technology node, the most recent one in Wattch for which the technology parameters are defined explicitly. We assume a low standby power (LSTP) design.

We modify SimpleScalar to model a time-accurate, five-stage, in-order pipeline. Table 2 shows the processor configuration. The pipeline blocks during L1D misses caused by load operations and is lock-free during one outstanding miss caused by a store or a prefetch from the TAB. Table 3 presents the energy for each access to the units in the data memory. We estimate the TAB energy values by using CACTI to provide only the data array cache power for an 8-line cache. We scale the 4-entry and 2-entry values by examining the ratio between 16-entry and 8-entry caches (due to the lack of information for the smaller cache sizes). Our evaluations include the energy usage during L1D misses.

Table 1. MiBench benchmarks.

Category	Applications
Automotive	Basicmath, Bitcount, Qsort, Susan
Consumer	JPEG, Lame, TIFF
Network	Dijkstra, Patricia
Office	Ispell, Rsynth, Stringsearch
Security	Blowfish, Rijndael, SHA, PGP
Telecomm	ADPCM, CRC32, FFT, GSM

Table 2. Processor configuration.

BPB, BTB	Bimodal, 128 entries
Branch Penalty	2 cycles
Integer&FP ALUs, MUL/DIV	1
Fetch, Decode, Issue Width	1
L1D & L1I	16 kB, 4-way, 32B line, 1 cycle hit
L2U	64 kB, 8-way, 32B line, 8 cycle hit
DTLB & ITLB	32-entry fully assoc, 1 cycle hit
Memory Latency	120 cycles
TAB (when present)	128 B, 32 B line, 4 lines

Table 3. Energy values estimated in Wattch.

Access Type	Energy
L1D (word) / (line)	903 pJ / 2,260 pJ
DTLB	100 pJ
2-TAB (word) / (line)	20.52 pJ / 51.3 pJ
4-TAB (word) / (line)	34.2 pJ / 85.5 pJ
8-TAB (word) / (line)	57 pJ / 142 pJ

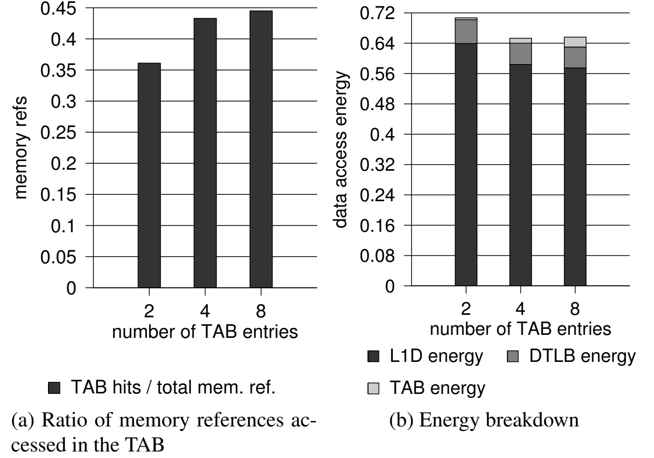
8. Results

We first analyze the impact of the total number of TAB entries on energy efficiency. Fewer entries require less power per TAB access, but they avoid fewer L1D and DTLB accesses. Fig. 12(a) shows the average ratio of the data memory references captured by the TAB for all benchmarks in Table 1. We can access 43.3% of the memory references through the TAB when using only four TAB entries. Using more TAB entries only slightly increases the number of accesses captured by the TAB. Fig. 12(b) shows the energy breakdown of the L1D/DTLB/TAB normalized to the total L1D/DTLB energy without the TAB. Total data-access energy usage is reduced by 34.7% with four TAB entries. In addition, using only four TAB line buffers makes the integration of the structure in the execute stage of the pipeline easier due to a faster access time. The four-entry TAB configuration reduces energy in the L1D and DTLB by 35.4% and 41.9%, respectively.

To evaluate the energy overhead of the additional instructions to acquire and release TABs, we use an L1 instruction cache (L1I) and an instruction TLB (ITLB) equal to the sizes of the L1D and DTLB, respectively. The overhead is only 2.2% of the total L1D/DTLB energy without the TAB.

We now present a more detailed evaluation using the most energy-efficient TAB configuration with four entries. On average, the ratio of the load/store operations captured in the TAB is 43.3% (Fig. 13), with the percentage for each benchmark falling between 10% and 96%. Only 0.5% of the TAB load/store instructions miss in the TAB due to overwritten elements in the *index* buffer, which shows that using LIFO circular buffer allocation works well with very low overhead.

Strided accesses constitute 67.8% of the TAB accesses, and loop-invariant address references constitute the remainder. Several applications have a high percentage of loop-invariant address references. For instance, the source code of *dijkstra* uses all global variables. Its kernel loop includes a function call, which prevents our compiler from detecting strided references because it assumes that counter variables could be updated in the called function. Nonetheless, it can detect that these counters have loop-invariant addresses, and counter updates comprise a large fraction of the memory references. The kernel loop of *crc* also contains pointers having loop-invariant addresses that are repeatedly dereferenced. A function call in this loop prevents the compiler from assuming the dereferenced pointer values are loop-invariant, hence it cannot apply loop-invariant code motion.



(a) Ratio of memory references accessed in the TAB

(b) Energy breakdown

Figure 12. TAB utilization and energy breakdown of data accesses for different numbers of TAB entries.

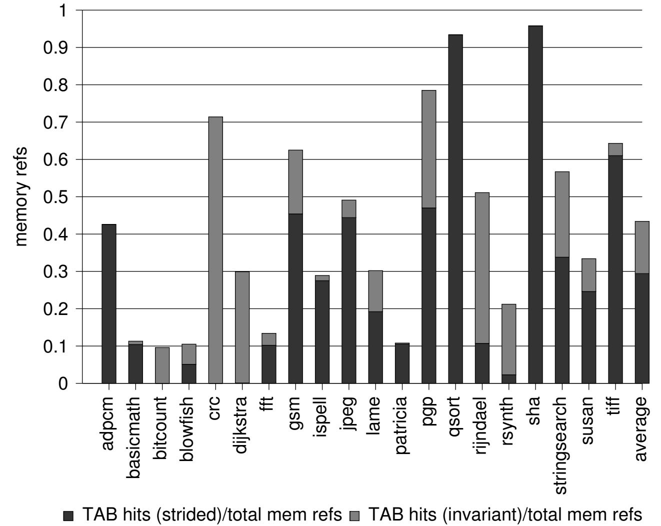


Figure 13. TAB utilization for different benchmarks.

Fig. 14 shows the breakdown of the L1D accesses with and without TAB accesses. The L1D accesses are reduced by 40.1%, on average. With compiler-controlled allocation and deallocation of the line buffers, the numbers of L1D prefetches and writebacks are only 2.4% and 0.9%, respectively, of the total number of loads and stores without a TAB. The write-first optimization reduces TAB prefetches from the L1D by 20.6%, on average.

Fig. 15 shows the reduction in DTLB accesses. The number of TAB-related DTLB accesses is very small, and these accesses are mainly caused by *gtab* instructions (initial prefetches). On average DTLB accesses are reduced by 41.9% using the TAB.

The TAB eliminates, on average, 12.1% of data accesses to the L2 cache. This benefit comes mainly from write-first contiguous TAB entries for which all data are overwrit-

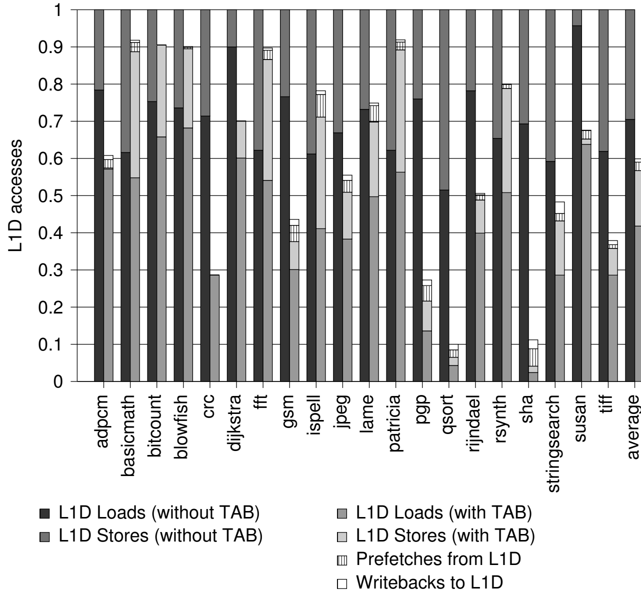


Figure 14. L1D access breakdown.

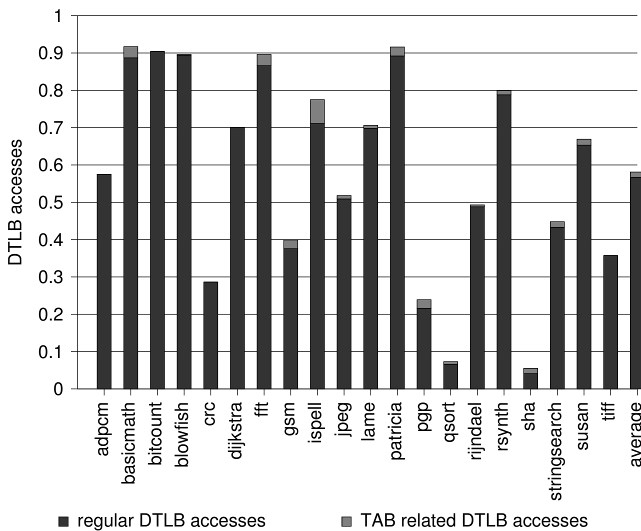


Figure 15. DTLB statistics with the TAB.

ten before being read. Apart from providing performance benefits, decreasing access counts to the next memory hierarchy level further improves energy efficiency. Note that for some applications the number of next-level accesses increases very slightly. Depending on the variable's alignment, the last TAB prefetch initiated inside a loop may be unnecessary if it occurs just before the loop exit, and such prefetches can trigger additional cache misses compared to an architecture with no TAB. As this situation only occurs just before TAB entries are released, the impact is negligible.

Fig. 16 shows how the execution time is affected by using the TAB. The execution time is improved on average by 3.1%. Avoiding load hazards by accessing the TAB in the execute stage of the pipeline delivers a 3.3% improvement. Avoiding some L1D miss stalls by prefetching L1D lines into the TAB and avoiding unnecessary data transfers between memory hierarchy levels deliver an additional execution time benefit of 0.2%. The *gtab* and *rtabs* instructions cause extra execution cycles, as do instructions to calculate the initial TAB access address in the loop preamble. Likewise, intercepting regular load/store instructions in the TAB incurs some execution time overhead. These extra cycles degrade performance by 0.4%. Clearly, the overheads are very limited when compared to the execution time benefits of the TAB approach.

On average, the *gtab* and *rtabs* instructions instructions incur 1.5% code size expansion. In addition, 0.4% of the regular load/store references are intercepted in the TAB. Without the optimizations 1.9% of the references are intercepted; our false-interference optimizations reduce this interception count by 73.1%.

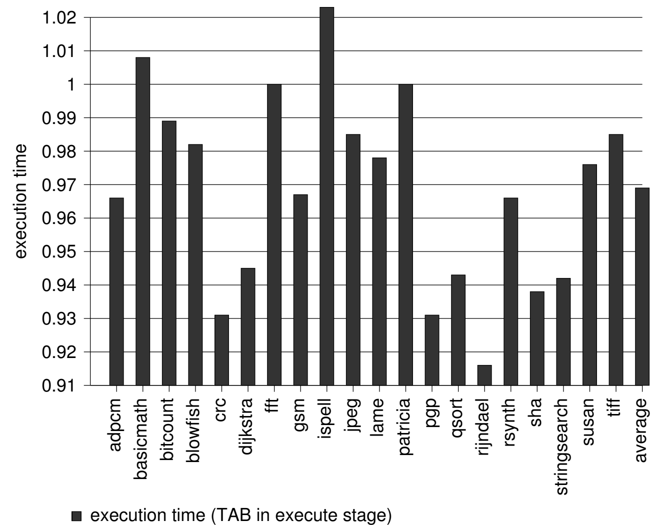


Figure 16. Impact of using the TAB on execution time.

To better understand the costs of using a TAB, we examine *ispell*, the application with the highest overhead. Much of its increased execution time is due to executing extra instructions to acquire and release TABs before and after loops with few iterations (processing characters in words). *ispell* also often requires additional instructions to calculate the initial address for the *gtab* instruction because the compiler does not have the required registers available to perform loop strength reduction.

9. Related Work

Witchel *et al.* propose a hardware-software design for data caches called direct address caches (DAR) [17]. The DAR eliminates L1D tag checks for memory references when the cache line to be accessed can be identified by the compiler as being known. The compiler annotates a memory reference that sets a register identifying the accessed L1D line and subsequent memory references that are guaranteed to access the same line reference that register to avoid the tag

check. The compiler uses some of the immediate bits of the load/store operations for control purposes in a manner similar to the TAB approach. The DAR reduces energy usage by avoiding accesses to the tag array and by activating only a single way of the L1D data array for the memory references that are guaranteed to access a specified L1D line. The TAB approach also avoids tag checks, but, in contrast, it accesses a much smaller and more power-efficient TAB structure, as opposed to a much larger single way of the L1D data array. In addition, the DAR approach requires code transformations, such as loop unrolling, to make alignment guarantees for strided accesses. Many loops cannot be unrolled, as the number of loop iterations must be known at the point the original loop is entered. When the alignment of a variable cannot be identified statically, a pre-loop is inserted to guarantee alignment in the loop body, which can be complex to align references to multiple variables. These transformations can also result in code size increases. The TAB approach does not require such extensive code transformations.

Kadayif *et al.* propose a compiler-directed physical address generation scheme to avoid DTLB accesses [10]. Several translation registers (TRs) are used to hold PPNs. The compiler determines the variables that reside on the same virtual page, and a special load instruction stores the translated PPN in one of the TRs. Subsequent memory references that access this page avoid the DTLB, getting the PPN from the specified TR register. The compiler uses some of the most significant bits of the 64-bit virtual address to identify whether the access must get the physical address from a particular TR. If the virtual address cannot be determined statically, additional runtime instructions modify the virtual address dynamically. Several code transformations, including loop strip mining, are used to avoid additional DTLB accesses, but these transformations increase code size. This approach reduces the number of DTLB accesses and thus DTLB energy usage, but L1D accesses occur as normal. Zhou *et al.* propose a heterogenous tagged cache scheme, where the DTLB accesses for private data can be eliminated [18]. It requires to add some logic to the most significant bits of the calculated memory address which will go through the critical path of the DTLB. In both of these schemes, only DTLB accesses are avoided, while the TAB approach avoids both DTLB and L1D accesses, which saves more data access energy.

Other small structures have been suggested to reduce L1D energy. A line buffer can be used to hold the last line accessed in the L1D [15]. The buffer must be checked before accessing the L1D, placing it on the critical path. Our evaluations with the 20 Mibench benchmarks showed that the miss rate for a 32 byte last line buffer used for the data cache is 73.8%, which will increase the L1D energy usage instead of reducing due to continuously fetching full lines from the L1D cache memory. Small filter caches sitting between the processor and the L1D have been proposed to reduce the power dissipation of the data cache [13]. However, filter caches save energy at the expense of a significant performance penalty due to their high miss rate. This performance penalty can mitigate some of the energy benefits of using the filter cache. In any case, line buffers and filter caches are complementary to the TAB approach.

Nicolaescu *et al.* propose a power saving scheme for associative data caches [14]. The way information of the last *N* cache accesses are saved in a table, and each access makes a tag search on this table. If there is a match the way information is used to activate only the corresponding way. It would be possible to use this and similar techniques in combination with the TAB to reduce L1D access power for loads and stores that are not captured by the TAB.

There has also been some research on using scratchpads in which variables are explicitly allocated or copied [3, 11, 12]. Scratchpads can save energy since they are typically small structures, and there is no tag check or virtual-to-physical address translation. While much smaller than main memory, scratchpads are typically much larger than the TAB structure. Furthermore, unlike the TAB, scratchpads are exclusive of the rest of the memory hierarchy and require extra code to copy data to and from the main memory system, which is a challenge for the compiler writer or the application developer. Since data must be explicitly copied to/from scratchpads, they cannot be used to reduce the energy used for strided accesses that are not repeatedly referenced.

The POWER family instruction set uses a *data cache line set to zero (dclz)* instruction [1] that works much like write-first contiguous TAB accesses. If the line resides in the cache, all the elements are set to zero. If it is not resident, then a tag allocation occurs, and all line elements are set to zero, but the data line is not fetched from the next level in the hierarchy. In contrast, a write-first contiguous access in TAB is not limited to just setting a line to zero.

10. Future Work

Enhanced compiler analysis and code transformations can be applied to increase the percentage of memory references accessed through the TAB. Analysis can determine the locations of globals so that multiple global scalar variables accessed with loop-invariant addresses can be allocated to a single TAB line buffer. Likewise, the sizes of activation records could be changed to always be an integer multiple of the L1D line size, which would allow multiple local scalars to be accessed in a single TAB line buffer. In both of these cases, the scalars could be located to require fewer TAB line buffers. In addition, write-first arrays could be aligned on L1D line boundaries to avoid additional line transfers between memory levels.

Another area to explore is how the TAB can reduce energy usage in an out-of-order processor. However, some interesting challenges must then be addressed. First, the memory references in a loop associated with a specific TAB entry

cannot be reordered, as the TAB relies on preserving the order of the strided memory references detected statically by the compiler. The load/store queue can enforce in-order processing of each TAB entry's references when selecting the next memory reference to perform, since the TAB entry is known after instruction decode. Alternatively, in-order processing of all memory references could be enforced whenever there are one or more TABs in use. This can be detected by *gtab*/*rtabs* instructions or TAB valid bits, which eliminates the need to use memory-fence instructions. Likewise, prefetching an L1D line into the TAB along a misspeculated path can cause problems because there is no tag check for TAB accesses. One simple solution would be to invalidate all TAB line buffers after each branch misprediction, as such events infrequently occur.

11. Conclusions

In this paper we describe an approach to access a large percentage of the data references in a more energy-efficient manner. We can replace a significant percentage of L1D accesses with references to a much smaller and more power-efficient TAB, which requires no tag checks or DTLB accesses because TAB entries are explicitly allocated. Furthermore, we provide small performance improvements by accessing values earlier in the pipeline and by avoiding some L1D miss stall cycles by prefetching data into the TAB. In addition, we avoid a number of unnecessary data transfers between memory hierarchy levels by classifying the access patterns of the memory references allocated to the TAB. We are able to make these enhancements with small ISA changes, relatively simple compiler transformations, and a TAB as small as a conventional 32-entry register file.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants CNS-0964413 and CNS-0915926, and in part by grant 2009-4566 of the Swedish Research Council.

References

[1] *Programming for AIX, assembler language reference, instruction set (AIX 5L Version 5.3)*, 3 edition, July 2006. URL `http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.aixassem/doc/alangref/alangref.pdf`.

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

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *10th Int. Symp. on Hardware/Software Codesign*, pages 73–78, 2002.

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

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual Int. Symp. on Computer Architecture*, pages 83–94, 2000.

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

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th Annual Int. Symp. on Computer Architecture*, pages 365–376, 2011.

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

[9] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *37th Annual Int. Symp. on Computer Architecture*, pages 37–47, 2010.

[10] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-directed physical address generation for reducing dTLB power. In *IEEE Int. Symp. on Performance Analysis of Systems and Software*, pages 161–168, 2004.

[11] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishman, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. pages 690–695, June 2001.

[12] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):281–287, Mar. 2004.

[13] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *30th ACM/IEEE Int. Symp. on Microarchitecture*, pages 184–193, Dec. 1997.

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

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

[16] S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid State Circuits*, 31(5): 677–688, May 1996.

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

[18] X. Zhou and P. Petrov. Heterogeneously tagged caches for low-power embedded systems with virtual memory support. *ACM Trans. Des. Autom. Electron. Syst.*, pages 32:1–32:24, 2008.