Redesigning a Tagless Access Buffer to Require Minimal ISA Changes

Carlos Sanchez,
Peter Gavin
Florida State University, USA
[sanchez,gavin]@cs.fsu.edu

Daniel Moreau
Chalmers University of
Technology, Sweden
moreaud@chalmers.se

Magnus Själander
NTNU, Norway
Uppsala University, Sweden
magnus.sjalander@idi.ntnu.no

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

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

Sally A. McKee
Chalmers University of
Technology, Sweden
mckee@chalmers.se

ABSTRACT

Energy efficiency is a first-order design goal for nearly all classes of processors, but it is particularly important in mobile and embedded systems. Data caches in such systems account for a large portion of the processor's energy usage, and thus techniques to improve the energy efficiency of the cache hierarchy are likely to have high impact. Our prior work reduced data cache energy via a tagless access buffer (TAB) that sits at the top of the cache hierarchy. Strided memory references are redirected from the level-one data cache (L1D) to the smaller, more energy-efficient TAB. These references need not access the data translation lookaside buffer (DTLB), and they can avoid unnecessary transfers from lower levels of the memory hierarchy. The original TAB implementation requires changing the immediate field of load and store instructions, necessitating substantial ISA modifications. Here we present a new TAB design that requires minimal instruction set changes, gives software more explicit control over TAB resource management, and remains compatible with legacy (non-TAB) code. With a line size of 32 bytes, a four-line TAB can eliminate 31% of L1D accesses, on average. Together, the new TAB, L1D, and DTLB use 22% less energy than a TAB-less hierarchy, and the TAB system decreases execution time by 1.7%.

CCS Concepts

•**Computer systems organization → Architectures;** •**Hardware → Power and energy;** •**Software and its engineering → Compilers;**

Keywords

energy efficiency, memory hierarchy, strided access

CASES '16, October 01-07 2016, Pittsburgh, PA, USA

ISBN 978-1-4503-4482-1/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2968455.2968504

1. INTRODUCTION

Mobile and embedded devices must operate under strict energy-usage constraints. Power expenditure affects not only battery lifetime and temperature but also performance: for a given power ceiling, processors using less power can be made to run faster. In many systems, the level-one data cache (L1D) accounts for up to 25% of a processor's total power draw [6, 8]. Decreasing the number of L1D accesses saves energy in the cache as well as in the data translation lookaside buffer (DTLB). Avoiding unnecessary transfers between levels of the cache hierarchy further reduces energy usage.

Our prior work introduces a small, power-efficient structure — the tagless access buffer (TAB) — into the cache hierarchy to reduce L1D and DTLB energy dissipation [3]. The compiler recognizes memory references that are accessed with a constant stride or whose addresses are loop-invariant, and it generates instructions to redirect such references to the TAB. Many applications spend most of their time in loops, and so capturing these references in the smaller, more power-efficient TAB yields significant energy savings.

The drawback to the original TAB framework [3] is that it necessitates substantial changes to the ISA: all load and store instructions must be modified, and two additional opcodes are required. We present a new TAB system in which the only ISA requirement is *one free opcode for the two instructions that allocate and deallocate TAB entries*. These two instructions — `gtab` (get TAB entry) and `rtabs` (release TAB entries) — explicitly manage the TAB, and thus load and store instructions need not be modified.

With 32-byte lines, a four-line TAB eliminates 31% of the L1D accesses, on average. Our new design achieves 72% of the original TAB's energy benefits while reducing ISA changes to the addition of a single opcode. Furthermore, our new TAB system can execute legacy non-TAB code, which was not possible with the original TAB.

2. THE NEW TAB SYSTEM

The TAB holds cache lines inclusive to the L1D. Figure 1 illustrates its position in the memory hierarchy. Rather than using hardware to predict the best lines to move, the compiler generates special instructions to explicitly move lines from the L1D to the TAB. We use "TAB" to refer to the whole buffer and "TAB entry" to refer to an individual entry and its associated line within the buffer.

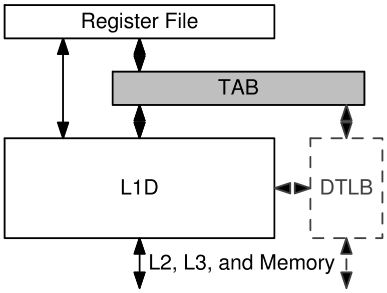


Figure 1: References that hit in the TAB (which sits between the L1D cache/DTLB and the register file) save energy by removing the need to access the cache and DTLB.

```
int a[1000];

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

(a) original loop

```
L1:
    r[2]=M[r[7]];     # load a[i] value
    r[3]=r[3]+r[2];   # add value to sum
    r[7]=r[7]+4;      # calc addr of a[i+1]
    PC=r[7]<r[6],L1;  # goto L1 if &a[i+1]<&a[n]
```

(b) compiler-generated RTL instructions

```
    gtab T[1],r[7],4
L1:
    r[2]=M[r[7]];
    r[3]=r[3]+r[2];
    r[7]=r[7]+4;
    PC=r[7]<r[6],L1;
    rtabs T[1]
```

(c) RTL with compiler-generated TAB instructions

Figure 2: This example (adapted from our previous work [3]) shows how the compiler generates code for a constant-stride loop both without and with TAB references.

The compiler detects loop memory references that have invariant addresses or constant strides, and it generates one or more `gtab` instructions to capture these in the TAB. The `gtab` instruction associates the base register of the captured memory references with the specified TAB entry. Memory references with this register are directed to the TAB instead of the L1D, and only references associated with the TAB entry are allowed to use this register within the loop. We track register associations in hardware, and since the base register is already a field in memory references, we need not alter the ISA's existing instructions.

Figure 2(a) gives a high-level example of how TAB instructions are generated for a simple loop iterating over the elements of the integer array `a[]`. Figure 2(b) shows the generated instructions in *RTL* (register transfer list) format, which has a one-to-one correspondence with MIPS assembly instructions. The compiler detects that memory reference `r[2]=M[r[7]]` is accessed with a constant stride due to the addition `r[7]=r[7]+4`. Figure 2(c) shows the `gtab` instruction the compiler generates before the loop and the `rtabs` instruction it generates after. The `gtab` tells the processor to associate register `r[7]` with TAB entry `T[1]`. Each memory reference using `r[7]` will now access this TAB entry. Addresses to prefetch are calculated with stride four. The `rtabs` instruction releases `T[1]`, which disassociates register `r[7]` from the TAB entry.

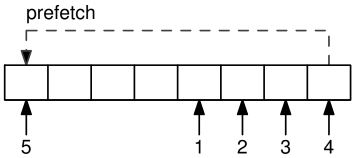


Figure 3: This example shows sequential accesses that hit in the TAB. Reference #4 accesses the last slot in the TAB line, which invokes a prefetch from the L1D.

Figure 3 gives an example of a TAB line access pattern. To prepare for the first TAB reference, the `gtab` instruction prefetches the line from the L1D. TAB references with constant strides cause a prefetch when the next reference address (calculated from the stride) crosses the line boundary. Since the next TAB reference address is always known, L1D tag checks and DTLB accesses for each reference become unnecessary. The sequential references continue to access the TAB until an `rtabs` instruction deallocates the TAB entry and removes the base register association.

The `gtab` instruction also communicates other access pattern information (*type info*) that can be used to further reduce energy. For instance, for access patterns overwriting all bytes in a line, we need not fetch data from the L1D. Section 3.1 describes the type info and how it is used.

3. SUPPORT FOR TAB OPERATION

We next describe the necessary hardware for TAB support. Section 3.1 describes the hardware structures in the new TAB design. Section 3.2 describes the fields of the two new instructions, and Section 3.3 describes the few, small L1D changes required for TAB support.

3.1 TAB Organization

Figure 4 gives an overview of our TAB organization. The *register array* stores the base register number for each TAB entry. The *TAB valid window* array is a circular buffer indicating which TAB entries are valid within the current function's context. The register array and TAB valid window are used to determine which TAB entry, if any, is associated with a given access. The base register of the memory reference is compared in parallel with all register array entries. The results are ANDed with the bits in the current TAB valid window. If the base register matches a valid TAB entry, that entry is accessed on the next cycle. Section 4.2 describes how the TAB valid window supports function calls.

Each TAB entry includes several metadata fields. The *index* associates a TAB entry with a data line in the buffer. Each line in the buffer also has metadata specific to the data it holds. A single TAB entry may be linked to two lines, each with its own metadata, and a line may be shared by multiple TAB entries. Figure 5 shows an expanded view of both sets of metadata. Field widths in our implementation are given on top (these numbers may differ in other systems).

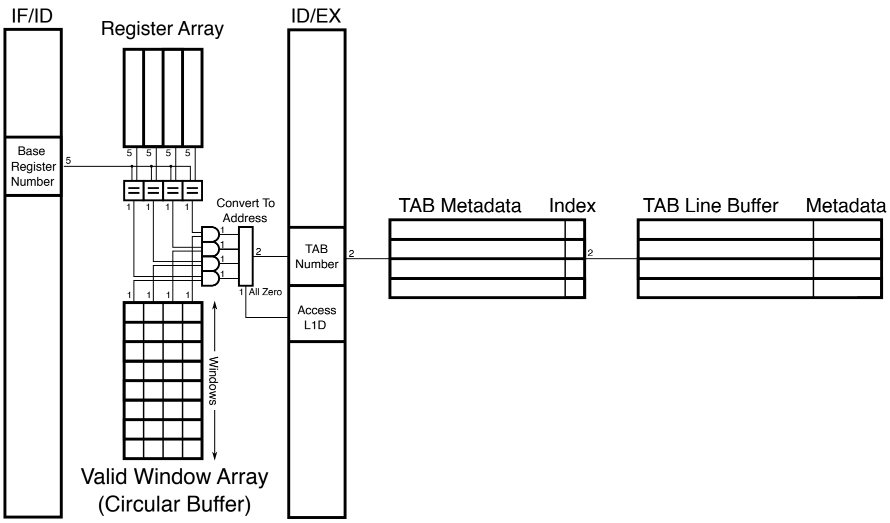
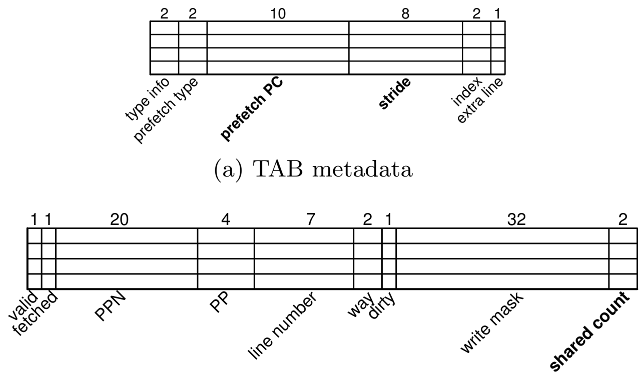


Figure 4: In this TAB hardware overview, the pillars represent pipeline registers between stages, and the small numbers indicate bit widths of the lines. The "=" gates are comparators, and the "Convert to Address" demultiplexer converts a field with a single bit set to a proper two-bit address. If all bits are zero, the "L1D Access" control signal directs the memory reference to cache.



(a) TAB metadata

(b) line buffer metadata

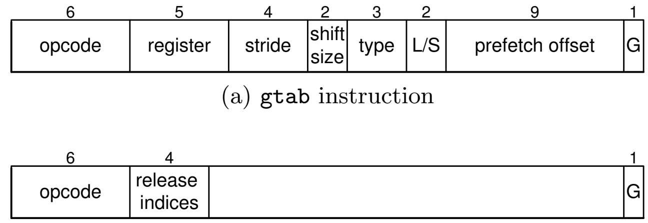
Figure 5: The TAB tracks metadata to ensure correct, energy-efficient operation (fields that differ from the original TAB [3] are labeled in bold).

Figure 5(a) shows the TAB metadata, which include the *type info*, *prefetch type*, *prefetch PC*, *stride*, *index*, and *extra line* fields. Fields that are new compared to the original TAB metadata [3] are depicted in bold. The type info bits control how data are transferred from the L1D to the TAB [3] (to save energy). The two-bit **prefetch type** indicates whether all loads, all stores, no references within the loop, or a single reference should trigger a prefetch. For instance, if the "all loads" case is indicated, any load directed to this TAB entry performs the prefetch check (but will not necessarily trigger a prefetch). If "no references within the loop" is set, the address is loop invariant, and the `gtab` instruction performs the only prefetch. If a single instruction causes a prefetch and the **prefetch PC** field matches the least significant bits of the current program counter, then a prefetch check is done. If the prefetch type is met and the stride plus the current reference's address crosses the line boundary, the TAB prefetches the next line. The index field determines which of the four line buffers is associated with this TAB entry. This lets multiple TAB entries share the same line. Finally, if the extra line bit is set, the current TAB line and the next line are allocated to a single TAB entry.

The compiler allocates an extra line when multiple references are directed to a TAB entry, are accessed out of order, and span two lines. Instead of prefetching back and forth, which wastes energy, we use two lines and prefetch only the line needed next. The TAB register array associates base registers with the first TAB line of the pair, and the least significant bit of the L1D set field determines which line to use.

Figure 5(b) shows the line buffer metadata, which includes the *valid*, *fetched*, *PPN* (physical page number), *PP* (page protection), *line number*, *way*, *dirty*, *write mask*, and *shared count* fields. The valid bit for the TAB entry is separate from the buffer valid bit. If the line is evicted from the L1D or the page table is updated, the buffer's valid bit gets cleared. The next access to the TAB initiates a line fetch that causes both the L1D and DTLB to be accessed. The fetched bit gets set when a line arrives from the L1D (so prefetches must first clear this bit). Any access to a TAB entry for which the fetched bit is not set stalls until the data arrive. The PPN and line number make up the high-order bits of addresses being directed to a TAB line. On a prefetch, the PPN field is prepended to the next sequential line number and used to directly access the L1D (removing the need for a DTLB lookup). The DTLB must only be accessed when the PPN field is updated, as with a `gtab` instruction or a prefetch that crosses a page boundary, both of which infrequently occur. The PP bits are used to enforce the proper page permissions (again avoiding DTLB accesses).

On a page table update, the mechanism that updates the DTLB also invalidates all TAB entries. This policy negligibly impacts performance, since page table updates infrequently happen. The line number and way fields together give the L1D location of the line being used in the TAB.



(a) `gtab` instruction

(b) `rtabs` instruction

Figure 6: The two new TAB instructions share an opcode, but the `gtab` instruction requires more fields than the `rtabs` instruction.

This enables memory references not directed to the TAB but sharing the same line to determine which TAB entry holds the line (Section 3.3 discusses how we handle such interfering accesses). We use these fields to reduce energy overheads when the TAB performs an L1D writeback, since we need not perform an L1D tag check. The dirty bit and write mask control how data are written back to the L1D. The dirty bit simply states that the data in the TAB line differ from those in the L1D and must be written back when the TAB line is invalidated or evicted. The write mask indicates which bytes have been altered; only dirty bytes are written back to the L1D. Finally, the **shared count** records the number of TAB entries mapping to the associated line.

3.2 ISA Modifications

The ISA must be updated to support the TAB, but the two needed TAB operations can be represented using a single opcode that has a one-bit field to differentiate between `gtab` and `rtabs`. Figure 6 shows the instruction formats for a MIPS-like 32-bit ISA.

The `gtab` instruction format depicted in Figure 6(a) has no immediate field. In order to properly perform the first prefetch on the execution of a `gtab` (pulling the first L1D line into the TAB), the specified register must contain the address for the first reference. This requirement sometimes causes the compiler to generate a few additional instructions to add an offset to the register before the `gtab` instruction, but these instructions are outside the loop and negligibly impact performance.

In order to support a wider range of values, the access stride is calculated as the `gtab` field's *stride* $\ll$ *shift size*. For instance, assuming a four-byte integer, the stride when accessing an integer array is always a multiple of four. We can thus set the shift size to two and use the four bits of the stride field for the upper bits of the actual stride.

The *type* information is a combination of the TAB metadata's two-bit type-info field and one-bit extra-line field. The *L/S* field specifies the prefetch scheme described above, and the *prefetch offset* is multiplied by the instruction width and added to the current PC to generate the prefetch PC in the TAB metadata. The signed nine-bit prefetch offset limits the distance between the `gtab` and the prefetching reference instruction: this is why the prefetch PC field only needs ten bits. The *G* bit differentiates `gtab` and `rtabs` instructions. Figure 6(b) depicts the `rtabs` instruction format, which only has a bitfield to indicate which TABs to deallocate.

3.3 L1D Changes

Inclusion requires that a TAB line be evicted when its corresponding line is evicted from the L1D. To avoid checking all TAB line numbers against the L1D line number on every eviction, we extend each L1D line with T and I bits. The T bit specifies whether this line resides in the TAB, and the I (intercept) bit specifies whether non-TAB accesses to this line should be directed to the TAB. The T bit is used to maintain inclusion: only evicted L1D lines with their T bits set will trigger an invalidation of the TAB line buffer. The L1D line number and way are compared to each TAB entry to determine which line buffer to invalidate. Since the TAB is small, evictions requiring an invalidation are infrequent, and the overhead of performing these checks is manageable.

If a normal (non-TAB) load or store accesses an L1D line that also resides in the TAB, it may need to be redirected to the TAB line instead (because the TAB has the most recent version). Such interference infrequently occurs, but when it does the data must be read on the following cycle, as we must now wait for the TAB access. Normally we would be able to use the T bit to indiscriminately redirect these references, but some TAB lines are guaranteed not to interfere with regular loads and stores when the compiler can determine that no other memory references access the same bytes within the line (i.e., to avoid false interferences). Thus, we use a separate I bit to specify the lines for which regular references should be directed to the TAB. The TAB sets the I bits based on the TAB entry's type info metadata.

4. TAB OPERATIONS

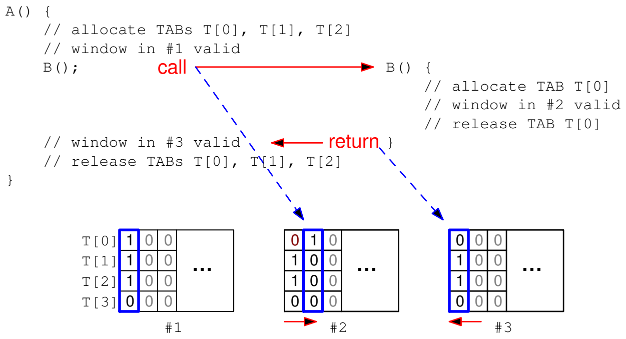
We next describe TAB allocation and deallocation, actions required on function calls, and prefetching.

4.1 TAB Allocation and Deallocation

All TAB entries are initially invalid across all windows. When a `gtab` instruction is executed, it allocates the specified TAB entry by prefetching the first line to be accessed and marking the TAB entry as valid. The register array is updated to associate the register given in the `gtab` instruction with the specified TAB entry. If the TAB entry to be allocated is already marked valid, the existing TAB entry must first be deallocated. Deallocation normally requires that associated dirty line(s) in the buffer be flushed back to the L1D and that the TAB entry be marked invalid. In the case of a deallocation triggered by a `gtab`, the TAB entry stays valid. When flushing the line, the write mask metadata field tells the TAB which bytes to update in the L1D. The `rtabs` instruction is used to explicitly deallocate one or more TAB entries. No action is required for invalid TAB entries specified in the `rtabs` instruction.

4.2 Function Call Support

Since TAB entries are associated with registers, they cannot remain live across function calls. The base register may be the same, but the actual address will almost certainly be different, and thus memory references would access the wrong line if directed to a TAB entry from a previous function. To avoid this problem, the TAB *valid window* circular buffer behaves similarly to a set of register windows. The buffer has a number of windows (our implementation uses eight), each having a bit per TAB entry to indicate which entries are valid for that window. Function calls shift the current window pointer forward; returns shift it backward.



```
A() {
    // allocate TABs T[0], T[1], T[2]
    // window in #1 valid
    B();                                      B() {
                                                 // allocate TAB T[0]
                                                 // window in #2 valid
                                                 // release TAB T[0]
    // window in #3 valid                       }
    // release TABs T[0], T[1], T[2]
}
```

	#1			#2			#3		
T[0]	1	0	0	0	1	0	0	0	0
T[1]	1	0	0	1	0	0	1	0	0
T[2]	1	0	0	1	0	0	1	0	0
T[3]	0	0	0	0	0	0	0	0	0

Figure 7: This figure shows the circular buffer that implements the TAB valid window. The current window (highlighted) advances on a function call and moves back on the return.

With this windowing system, a new set of valid bits is used per function call. Only the valid bits are unique per call — TAB metadata and line buffers are unaltered. A TAB entry can only be valid in one window because allocating that entry in a different window overwrites the metadata and line buffer. The valid window enables TAB entries *not* used in new function calls to remain active upon return, whereas those used in the function are invalidated.

Figure 7 gives an example of window system operation. Figure 7(a) shows the TAB operations of two functions, `A()` and `B()`. `A()` requests TAB entries `T[0]`, `T[1]`, and `T[2]` and then calls `B()`, which only requests TAB entry `T[0]`. Figure 7(b) depicts the TAB valid window structure at different points during execution. Just before the call to `B()`, the highlighted window shown in #1 is current, and TAB entries zero, one, and two are valid for that window. After `B()` allocates TAB entry `T[0]`, the valid bit is cleared for that TAB entry across all windows and then set in the current window (#2 in the figure). Recall that deallocating a TAB entry clears the valid bit for that entry. This ensures that TAB entries are only valid in at most one window. After `B()` returns, memory references in `A()` may continue to use TAB entries `T[1]` and `T[2]`.

When the window advances, all entries are invalidated in the new window. This avoids conflicts when function call stacks are deep enough to wrap and reuse a window.

4.3 Prefetches

The TAB prefetches a line from the L1D upon the execution of a `gtab` instruction or when a TAB entry's access pattern is about to cross the line boundary. This means that the data will usually be ready when the next TAB memory reference needs it. After the `gtab` instruction executes, subsequent prefetches are automatically performed based on the pattern specified in the prefetch type metadata field (Figure 5(a)). If there is a dirty line already associated with a TAB entry at the point of a prefetch, the original line is flushed before the new line is loaded. Recall that each successive address is calculated by adding the stride to the current address. If the line number portion of the calculated address differs from that of the current TAB line, a prefetch is triggered. Note that this address calculation requires only a small adder, as we need only check the final carry out from the line offset. This calculation is only performed for TAB references that fulfill the prefetch condition specified by the prefetch type.

As noted in Section 3.1, the prefetch type field indicates whether the TAB entry should perform a prefetch check on all loads, all stores, all of both, or a single reference. Note that if the compiler cannot determine that the access pattern fits one of these prefetch modes, it will not generate TAB instructions, as it cannot ensure correct execution.

If the TAB entry is set to perform the prefetch check for a single reference, the TAB uses the prefetch PC metadata field to determine whether to do the prefetch check for the current reference. Executing a `gtab` instruction calculates this field by multiplying the given offset (which the compiler calculates as the number of machine instructions between the `gtab` and the prefetch reference) by the instruction width (here, four bytes) and adding that to the current PC.

Using the TAB saves energy because accesses to it not only avoid L1D accesses, but also trigger fewer DTLB accesses than accesses to the L1D. Executing a `gtab` instruction causes the TAB to access the DTLB to get the PPN to store in the TAB metadata. Thereafter, only TAB references whose prefetches cross a page boundary need to access the DTLB, which rarely happens.

TAB entries use their index metadata field to access the line buffer. This indirection enables multiple TAB entries to map to the same line buffer. Prefetches thus first check the line number to fetch against valid lines already in the TAB. On a match, the TAB sets the index to point to the matching line. Otherwise, the TAB allocates a new line and sets the index accordingly. Even though line buffer metadata are shared among all TAB entries that map to a given line, TAB metadata remain unique to each entry. The TAB tracks the number of entries that map to each line. If multiple TAB entries point to the same line buffer and one entry requests a prefetch, that TAB entry decrements the line buffer's counter and prefetches into a new TAB line. A dirty line is not flushed back to the L1D until its counter is zero.

5. COMPILER ANALYSIS

The compiler needs no additional code transformations or interprocedural analyses to support the TAB; it just needs the number of TAB entries and the L1D line size to know how many TAB entries it can use within a loop nest and whether the stride fits within a line. The compiler detects memory references with loop-invariant addresses or constant strides. It inserts a `gtab` before a loop containing such references and an `rtabs` after. If they are not already present, the compiler generates loop preheader and postheader blocks for these instructions. Additional arithmetic instructions may be needed in the preheader to store the calculated initial address in the base register used by the `gtab`. Since these instructions are outside the loop, they have negligible performance impact. Figure 8 gives the algorithm for generating TAB instructions in both the old and new TAB systems.

5.1 References with Constant Strides

Constant-stride memory references have the form `M[reg]` or `M[reg+disp]`, where updates to `reg` follow the pattern `reg = reg` ± `constant`, and `disp` is a displacement off the address in `reg`. The latter pattern arises from applying induction variable analysis to regular data access patterns. For instance, sequentially accessing every element in an integer

```
FOR (each loop in function sorted by innermost first) DO
  FOR (each load/store in the loop) DO
    IF (reference has constant stride OR
        has a loop-invariant address) THEN
      IF (reference offset and base register allow it
          to be added to existing TAB) THEN
        Merge reference with existing TAB;
      ELSE
        Assign new TAB to reference;
  FOR (each TAB created) DO
    IF (TAB base register used elsewhere) THEN
      Remove TAB;
  IF (too many TABs) THEN
    Select TABs with most estimated references;

FOR (each TAB in function) DO
  Generate GTAB instruction in loop preheader;
FOR (each loop in function) DO
  IF (TABs associated with loop) THEN
    Generate RTABS instruction in loop postheader;
```

Figure 8: This pseudocode outlines the compiler algorithm for deciding when to allocate/deallocate TAB entries.

array generates the constant-stride pattern `reg = reg + 4` (assuming a four-byte integer). When the compiler detects a memory reference with an appropriate access pattern, it tries to allocate it to a TAB entry.

The requirements to allocate a single strided reference to a TAB entry are as follows:

1. The reference must be in a loop and must have a constant stride.
2. The reference must be in a basic block that is executed exactly once per loop iteration (due to the prefetch system). If the TAB line is not accessed on every iteration, the TAB may miss a prefetch and use the wrong line for the next access.
3. The stride must be less than or equal to half the L1D line size (otherwise using the TAB yields no energy benefits).
4. The base register of the reference must not be used as a base register in any other memory references not associated with the TAB entry within the loop.

Multiple references can be directed to a single TAB entry, provided they follow a few rules:

1. All references must be in the same loop.
2. All references must have the same constant stride.
3. All references must share the same base register (so that they are directed to the same TAB entry).
4. The reference(s) causing the prefetch must occur exactly once per loop iteration (much like TAB entries with a single reference).
5. The absolute value of the stride must be no larger than the L1D line size.
6. The maximum distance between any two references cannot be larger than the L1D line size.

This happens, e.g., in loop unrolling, which causes a single memory reference in the original loop to occur k times in the unrolled loop (so k is the unrolling factor). They all access the same structure, and assuming the original loop memory references have a constant stride, the multiple references also have constant strides. Sometimes a set of references can span two cache lines (e.g., due to an out-of-order access pattern), in which case the compiler sets the *extra line* field of the **gtab** instruction to indicate that this TAB entry requires two lines instead of one. The lowest bit of the index portion of the reference address is then used to direct references to the proper line within the buffer.

There are cases in which an extra line is unnecessary for a TAB entry with multiple references. If the references are accessed in order and in the same direction as the stride, and if the distance between each reference is the same (including the distance between the last reference in one iteration and the first in the next), a single TAB line buffer can be used — this case is no different from a single reference with a constant stride. Figure 9 illustrates an example of loop unrolling and of how the TAB can capture multiple references. Figure 9(a) shows code that sums the elements of an integer array, Figure 9(b) shows the loop after unrolling, and Figure 9(c) shows the generated instructions. Note that because the first reference has an offset, we must generate instructions to store that address in **r[6]** before emitting the **gtab** instruction.

```
int a[1000];

for (i=0; i<n; i++) {
    sum += a[i];
}
```

(a) original summation loop

```
int a[1000];

for (i=0; i<n; i+=4) {
    sum += a[i];
    sum += a[i+1];
    sum += a[i+2];
    sum += a[i+3];
}
```

(b) after loop unrolling

```
     r[6]=r[6]-12;     #calculate base register
     gtab T[1],r[6],4; #allocate T[1], stride=4
     r[6]=r[6]+12;
L1:  r[2]=M[r[6]-12];  #T[1] load+prefetch
     r[3]=M[r[6]-8];   #T[1] load+prefetch
     r[4]=M[r[6]-4];   #T[1] load+prefetch
     r[5]=M[r[6]];     #T[1] load+prefetch
     r[7]=r[7]+r[2];   #update sum variable r[7]
     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;  #loop condition
     rtabs T[1];
```

(c) unrolled loop references directed to TAB

Figure 9: This figure shows how a loop (a) is unrolled by a factor of four (b) and gives the RTL representation of the instructions the compiler generates to capture the loop references in the TAB (c) (example adapted from our previous work [3]).

5.2 References with Loop-Invariant Addresses

Although loops containing references with loop-invariant addresses are uncommon, the potential energy benefits from directing them to the TAB are large. TAB entries allocated for these types of references trigger at most one prefetch and one writeback, requiring only one DTLB and at most two L1D accesses. Depending on the access type stored in the type info metadata, only one L1D access may be needed. Figure 10 gives example code that allocates a TAB entry for a reference with a loop-invariant address. Figure 10(a) shows a loop with a function call to `scanf()`, which prevents the global variable `sum` from being stored in a register. Figure 10(b) shows the generated instructions, where `sum` has been allocated to a TAB entry. Notice that no reference causes a prefetch. The variable `n` is not allocated a TAB entry because its address is passed to another function.

```
sum = 0; //global variable

while (scanf("%d", &n)) {
    sum += n;
}
```

(a) original loop

```
     r[14]=sum;
     gtab T[1],r[14],0;
     M[r[14]=0;       #TAB T[1] store
     PC=L4;
L2:  r[3]=M[r[20]+n];
     r[5]=M[r[14]];   #TAB T[1] load
     r[5]=r[5]+r[3];
     M[r[14]]=r[5];   #TAB T[1] store
L4:  . . .
     r[5]=r[29]+n;
     ST=scanf;
     PC=r[2]!=r[0],L2;
     rtabs T[1];
```

(b) instructions when using the TAB

Figure 10: This example shows how the TAB is used for a loop-invariant memory addresses. The code at the top shows a sample loop, and code at the bottom shows the RTL representation of the instructions the compiler generates for the TAB system (figure adapted from our previous work [3]).

5.3 TAB Allocation Heuristics

Loop nests often have the potential to allocate more TAB entries than the hardware supports. In such cases, the compiler chooses the subset of memory references that are likely to be most profitable. As in the original TAB system [3], we identify these references by estimating the number of L1D accesses that could be avoided per loop iteration. Since the number of iterations is usually unknown at compile time, we exclude it from the calculation. This metric is calculated by the following equation:

$$\frac{estimated\_saved\text{-}references - (L1D\_loads + L1D\_writes)}{\#TAB\_lines} \qquad (1)$$

6. EVALUATION FRAMEWORK

In order to compare our results to those of our original TAB system, we again use the VPO compiler [5] and SimpleScalar simulator [1]. VPO produces MIPS/PISA target code and performs the analysis to generate the additional `gtab` and `rtabs` instructions. We use a SimpleScalar configuration that models a timing-accurate, five-stage, in-order pipeline. Table 1 gives details for this processor configuration. We choose to use a four-line TAB based on simulation results from our previous TAB implementation [3].

We estimate energy values for the TAB by synthesizing its structures with the Synopsys Design Compiler [15] using a 65nm 1.2-V low-power CMOS cell library. We use Synopsys PrimeTime [16] to estimate the power consumption of the entire structure by setting the design netlist's inputs to approximate the cycle-switching probabilities of a real workload. Since we use the same 65nm process technology, we take the L1D and DTLB energy figures from our recent work [4]. These values (shown in Table 2) are multiplied by event counters in SimpleScalar to produce overall energy projections for a simulation. We run the same 20 MiBench applications [7] used to study the original TAB system, and we use large input datasets. These benchmarks span the six application categories shown in Table 3.

Table 1: Processor configuration

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

Table 2: Energy values

Access Type	Energy
L1D (load) / (store)	170.0 pJ / 91.2 pJ
L1D (byte) / (line)	28.2 / 367.4 pJ
DTLB	17.5 pJ
TAB (word) / (line)	8.2 pJ / 10.6 pJ
Metadata (TAB) / (line buffer)	1.4 pJ / 4.5 pJ
Extra Structures (register array) / (valid window)	0.7 pJ / 2.3 pJ

Table 3: MiBench benchmarks

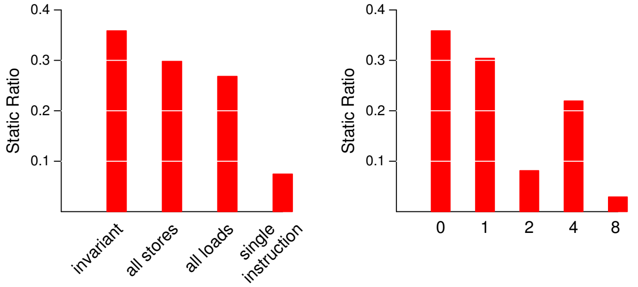
Category	Applications
Automotive	basicmath, bitcount, qsort, susan
Consumer	jpeg, lame, tiff
Network	dijkstra, patricia
Office	ispell, rsynth, stringsearch
Security	blowfish, rijndael, sha, pgp
Telecomm	adpcm, crc32, fft, gsm

7. EXPERIMENTAL EVALUATION

We first present performance and power results with respect to a TAB-less system. We then compare these results to those for the original TAB implementation.

7.1 Current Results

The compiler allocates one or more TAB entries to 63.3% of all loops, and the average number of TAB entries it allocates to these loops is 1.59. Figure 11(a) shows the static ratios of the prefetch schemes assigned to TAB entries by the `gtab` instructions allocating them. Over a third of the `gtab` instructions allocate TAB entries to data with invariant addresses: a prefetch occurs only once for the `gtab` instruction because the data remain in the TAB for the duration of the loop's execution. This happens because global variable



(a) prefetch types

(b) strides

Figure 11: The left graph shows ratios of static memory reference instructions generating different prefetch types. The right graph shows the portions of the statically allocated TAB entries accessed with a given stride.

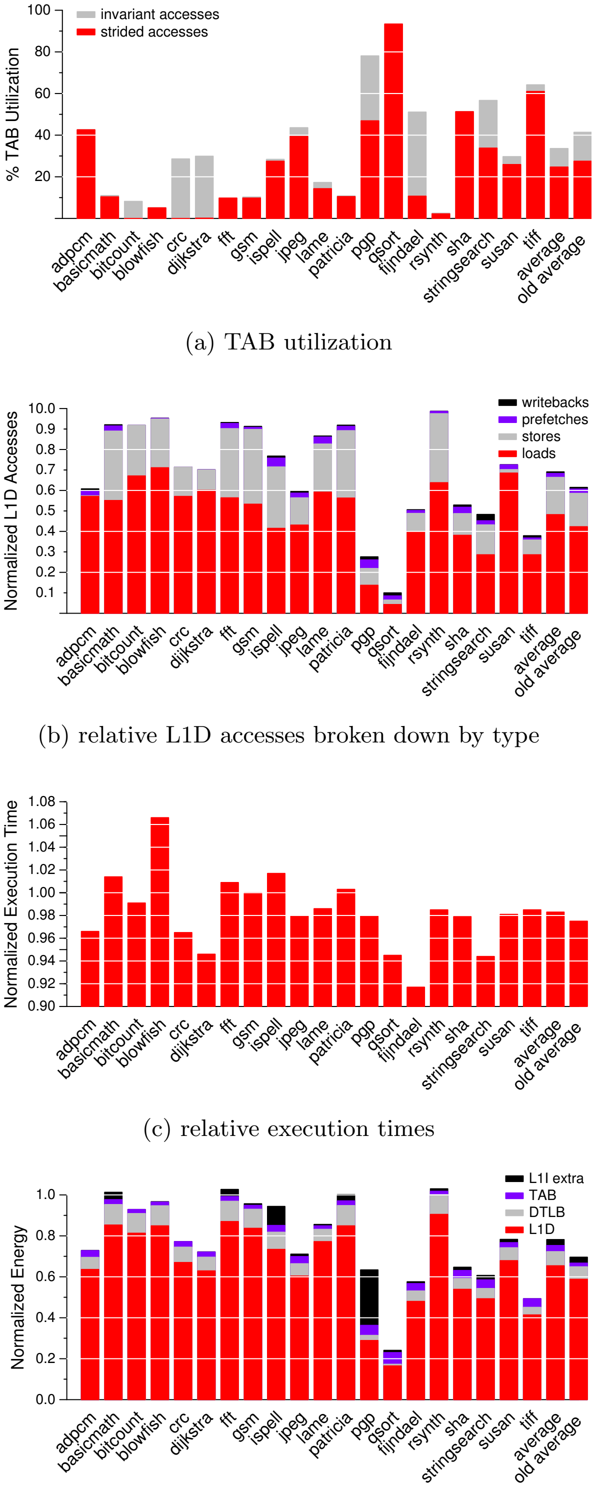
references cannot be hoisted from loops containing function calls. Most of the remaining **gtab**s allocate TAB entries set to prefetch on all stores or all loads. Fewer than 1% of the allocated TAB entries hold data that are referenced multiple times within a loop; these necessitate prefetching on only one of those memory reference instructions.

Figure 11(b) shows the distribution of strides with which the TAB entries are accessed. This breakdown is normalized to the number of TAB entries allocated by (static) **gtab** instructions generated by the compiler. Non-power-of-two strides are used by fewer than 1% of allocated TAB entries, and so the figure only shows power-of-two strides. TAB entries holding character (one-byte) and integer (four-byte) data are most common. TAB entries associated with invariant addresses have zero-stride accesses.

Figure 12(a) shows the percentages of memory references directed to the TAB and what portion of those accesses are strided or invariant. TAB references account for 33.4% of the total references, on average, but some applications have few TAB accesses. For instance, many functions in **rsynth** take large structures as pass-by-value parameters addressed through the stack pointer instead of through registers. These accesses cannot be directed to the TAB because they do not fit within a TAB entry, and they cannot be split between the L1D and the TAB because the hardware requires the base register to be exclusive to the TAB references. Passing large structures by value is not a standard programming practice, so it has little impact on results.

Figure 12(b) shows the breakdowns of L1D accesses on our TAB system versus the TAB-less baseline system. On average, the L1D is accessed 30.9% fewer times. TAB overheads — additional accesses for prefetches and writebacks — account for just 2.7% of normal L1D accesses.

Figure 12(c) gives executed cycles as a percentage of total baseline cycles. We access the TAB in the execute stage: it is small and fast, and the base register plus displacement calculation is only used for the offset into the TAB line buffer. Despite the extra instructions generated for the TAB, applications execute 1.7% fewer cycles, on average, largely because accessing the TAB in the execute stage avoids some load hazards.



(a) TAB utilization

(b) relative L1D accesses broken down by type

(c) relative execution times

(d) relative energy dissipated

Figure 12: These graphs show TAB performance and energy results for our 20 MiBench applications as compared to a TAB-less system. The second bars from the right show arithmetic means for all applications, and the right-most bars show the same means for the old implementation [3].

Figure 12(d) shows TAB system energy usage compared to total L1D and DTLB baseline energy. We include the extra energy dissipated in the L1 instruction cache (L1I) by the additional instructions required for TAB operation. These contribute just 2.9% of the baseline L1D and DTLB energy. Note that *ispell* and *pgp* use more L1I energy due to loops that execute few iterations. This means they cannot amortize the overheads of fetching *gtab* and *rtabs* instructions.

The extra TAB hardware has an average energy overhead of 3.0%, bringing the TAB's total operational overhead to 5.9%. Total average L1D/DTLB energy savings is 21.8%. Note that even when data are not being accessed in the TAB, loads and stores must still check the TAB register array and valid window to ensure that accesses are directed to the proper structure. Some applications use slightly more energy with the TAB because of such per-reference overheads, higher instruction counts, and/or low TAB hit rates. Such increases are small and infrequent, and we deem them reasonable tradeoffs for an implementation that requires only one ISA change and maintains backward compatibility.

7.2 Comparing New and Old TAB Systems

The old implementation takes bits from the immediate fields of load and store instructions to direct them to a TAB entry or to the L1D. The new implementation modifies no existing instructions, trading flexibility for compatibility with legacy code. In contrast to our old implementation, we now must restrict TAB allocation to references using a base register, and no non-TAB references may use that register. The two rightmost bars in Figure 12(a) show that these restrictions reduce the percentage of TAB accesses from 41.4% to 33.6%. Those in Figure 12(b) show that the percentage of L1D accesses grows from 61.6% to 69.1%, an increase of 7.5%. These restrictions have little effect on strided and invariant-address TAB accesses.

The rightmost bars in Figure 12(c) show an increase from 97.5% to 98.3% in relative execution times. This is due to having fewer TAB accesses and more generated instructions. The old implementation uses a `gtab` immediate field as to calculate the starting address, whereas this implementation uses these bits to encode other information. The starting address must instead be stored in the register specified in the `gtab`. As shown in Figure 5, sometimes the compiler must generate extra instructions to add an offset to the base register. This has little impact: execution time is still lower than when not using the TAB.

The rightmost bars in Figure 12(d) show an 8.6% decrease in energy savings (from 30.4% to 21.8%) from having fewer TAB accesses compared to old implementation and from the overhead of the extra hardware structures. Decreased TAB utilization accounts for 86.04% of the lower energy savings, and the extra TAB hardware accounts for 13.95%. The new implementation's increased instruction count has negligible impact, accounting for only 0.01% of the energy differences.

8. RELATED WORK

Witchel et al. [17] propose a hardware/software design that uses direct address registers (DARs) to hold information on lines loaded into the L1D. The compiler places values in the DARs as an optional side-effect of performing a load or store, and subsequent accesses to the cache line bypass tag checks, directly addressing the data array. The compiler uses some immediate bits of the load/store operations for control purposes, much like the original TAB approach. The TAB approach presented here avoids such invasive ISA changes. 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 memory references guaranteed to access a specified L1D line. The TAB also avoids tag checks, but it accesses smaller and more power-efficient structures compared to a single way of the much larger L1D data array.

In addition, the DAR approach requires code transformations like loop unrolling to make alignment guarantees for strided accesses. Many loops cannot be unrolled because the number of loop iterations is not known at the entry point the loop. When the compiler cannot identify a variable's alignment, it inserts a preloop to guarantee alignment in the loop body (and multiple variables can make this alignment complex) The TAB requires no extensive code transformations.

Kadayif et al. [9] propose a compiler-directed physical address generation scheme to avoid DTLB accesses. Several translation registers (TRs) hold PPNs. The compiler identifies variables residing in the same virtual page and generates a special load instruction to store the PPN in a TR. Subsequent references accessing this page bypass the DTLB, getting the PPN from the specified TR. 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 dynamically modify the virtual address. 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 it does not reduce the number of L1D accesses.

Others have suggested adding small structures to reduce L1D energy. For instance, Su and Despain [14] add a buffer to hold the last L1D line accessed. This buffer resides on the critical path: it must be checked before each L1D access. Our evaluations with our MiBench applications find that a 32-byte last-line buffer incurs a 73.8% miss rate. This architecture continuously fetches full lines from the L1D, increasing rather than decreasing energy usage. Kin et al. [12] propose small filter caches between the processor and the L1D to reduce power dissipation in the data cache. Filter caches have historically suffered high miss rates that unacceptably lower performance. In other work, we present a filter cache implementation that reduces both energy and execution time [4]. The TAB can be used with our filter cache to further reduce energy: TAB accesses exploit sequential locality detected at compile time, while the filter cache automatically detects other locality.

Nicolaescu et al. [13] propose a power-saving scheme for associative data caches. A table stores way information for the last N cache accesses, and all cache accesses cause tag searches in the table. On a match, the corresponding information is used to activate only that way. This approach still requires an L1D access on every data reference, but it (and similar techniques) can be combined with the TAB to reduce L1D access power for accesses the TAB does not capture.

Scratchpads [2,10,11] can reduce energy usage, since they require no tag checks or virtual-to-physical address translation. Variables must be explicitly allocated to these small memory structures, which are typically much larger than the TAB. Unlike the TAB, scratchpads are exclusive with respect to the rest of the memory hierarchy, and they require extra code to copy data to and from main memory. This presents a challenge for compiler writers or application developers. Since data must be explicitly copied, there is no implicit support for strided access patterns.

9. CONCLUSION

Our original TAB system reduces L1D and DTLB energy usage by almost a third, but it requires changes to load and store instructions. The alternative implementation that we present here saves less energy (by over a fifth), but it is backwards-compatible with codes compiled for a TAB-less system, and it requires only one additional opcode. Both the old and new implementations reduce energy by replacing L1D accesses with those to a smaller, more power-efficient structure that requires no tag checks. As in the original TAB, this implementation reduces execution time by accessing the TAB earlier in the pipeline and by using prefetching to offset stalls incurred by L1D misses.

10. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-0964413, DUE-1241525, DUE-1259462, IIA-1358147, and CCF-1533846, and by the Swedish Research Council grant 2015-5159. The authors thank Alen Bardizbanyan for sharing his expertise from previous collaborations.

11. REFERENCES

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

[2] 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 *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 73–78, May 2002.

[3] A. Bardizbanyan, P. Gavin, D. Whalley, M. Själander, P. Larsson-Edefors, S. A. McKee, and P. Stenström. Improving data access efficiency by using a tagless access buffer (TAB). In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 269–279, Feb. 2013.

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

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

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

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

[8] 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 *Proceedings of the International Symposium on Computer Architecture*, pages 37–47, June 2010.

[9] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-directed physical address generation for reducing dTLB power. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 161–168, Mar. 2004.

[10] 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.

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

[12] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the International Symposium on Microarchitecture*, pages 184–193, Dec. 1997.

[13] 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 the International Conference on Computer Design*, pages 101–107, Oct. 2006.

[14] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: A case study. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 63–68, Apr. 1995.

[15] Synopsys, Inc. *Design Compiler®, v. 2010.03*, 2010.

[16] Synopsys, Inc. *PrimeTime® PX, v. 2011.06*, 2011.

[17] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the International Symposium on Microarchitecture*, pages 124–133, Dec. 2001.