FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

THE TAGLESS ACCESS BUFFER: IMPROVING DATA ACCESS EFFICIENCY WITH MINIMAL ISA CHANGES

By

CARLOS R SANCHEZ

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
2015

Carlos R Sanchez defended this thesis on November ?, 2015.
The members of the supervisory committee were:

Professor

David Whalley

Committee Member 1

Gary Tyson

Committee Member 2

Xin Yuan

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

TAB - **T**agless **A**ccess **B**uffer
L1D - **L**evel **1**(One) **D**atacache
GTAB - **G**et **TAB**
RTABS - **R**elease **TABS**
DTLB - **D**ata **T**ranslation **L**ookaside **B**uffer
PPN - **P**hysical **P**age **N**umber

ABSTRACT

Energy efficiency is an important design consideration in nearly all classes of processors, but is of particular importance to mobile and embedded systems. The data cache accounts for a significant portion of processor power. An approach has been previously presented to reduce cache energy by introducing an explicitly controlled *tagless access buffer* (TAB) at the top of the cache hierarchy. The TAB reduces energy usage by redirecting loop memory references 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 can sometimes avoid unnecessary transfers from lower levels of the memory hierarchy. In this thesis we improve upon the previous TAB design to create a system that requires fewer instruction set changes, gives more explicit control over the allocation and deallocation of TAB resources, and is backwards compatible with existing code. We show that with a cache line size of 32 bytes, a four-line TAB can eliminate on average 31% of L1D accesses, which reduces L1D/DTLB energy usage by 22% with TAB accesses included.

CHAPTER 1

INTRODUCTION

Mobile device and embedded processors have strict energy usage constraints placed on them by their system's design. Even in large-scale systems like supercomputing clusters, energy usage for individual processors can be the determining factor for a system's maximum processing capability [15]. As such, given a strict energy usage ceiling, a processor model that uses less energy can be made to run faster than those that use more energy. Energy usage can be reduced by decreasing accesses to the level-one data cache (L1D), a system which accounts for up to 25% of a processor's total power draw [7, 9]. This subsequently reduces energy usage for the data translation lookaside buffer (DTLB), as each cache access must check this buffer to convert virtual addresses to physical addresses. Inefficiencies in the data transfers between memory hierarchies can also be improved to reduce energy usage.

In order to reduce the power dissipation caused by accesses to the data cache and the DTLB without degrading execution time or requiring significant instruction set architecture (ISA) changes, we introduce a *tagless access buffer* (TAB) into the cache hierarchy and expose control of this structure to the compiler. The compiler can recognize memory references within loops whose addresses are invariant or accessed with a constant stride and generate instructions to redirect these references to the TAB. It is often the case that most of an application's execution time is spent in loops, so significant energy savings come from capturing these references in the smaller, more power efficient TAB.

Using a TAB provides the following advantages. (1) We significantly reduce energy usage by replacing L1D accesses with TAB accesses. A DTLB lookup or tag check is not required when accessing the TAB, further reducing energy expenditure. (2) Execution time is slightly improved by prefetching lines from the L1D, which offsets some of the stalls incurred when a line is not present in the L1D. This improvement comes despite a slight increase in instruction count. (3) Energy expenditure is further reduced by avoiding unnecessary data transfers within the memory hierarchy, made possible by the compiler's explicit control over the TAB structure.

This thesis presents an updated version of our original TAB system [4] and makes the following contributions. Our new implementation requires fewer ISA changes, and the compiler has explicit control over which TABs are allocated and deallocated. The only ISA requirement is one free opcode for the two instructions, TAB allocation and deallocation, necessary to operate the TAB. Furthermore, load and store instructions do not have to sacrifice bits to control access to the TAB, as all actions can be performed from the two TAB instructions. Unaltered loads and stores makes the new TAB system backwards compatible, allowing old code to execute on the new architecture. This new implementation achieves 72% of the original energy benefits while reducing the ISA changes to just the addition of one opcode.

CHAPTER 2

THE TAB SYSTEM

The TAB is a buffer that holds a small number of cache lines which are inclusive to the L1D. It is placed at the top of the memory hierarchy, as seen in Fig. 2.1. Rather than using hardware to predict the best lines to move up the hierarchy, the compiler performs analysis and generates special instructions to explicitly move lines from the L1D into the TAB. This makes the TAB able to capture more memory references than could be recognized with only hardware analysis. Two instructions control the TAB: *gtab* (get TAB entry) and *rtabs* (release TAB entries). For the remainder of this paper, "TAB" will refer to the whole buffer, whereas "TAB entry" will refer to individual entries and the associated line within the buffer.

The compiler detects loop memory references that are invariant or have a constant stride. It then generates one or more *gtab* instructions to capture these references. The *gtab* instruction associates the base register of the captured memory references with the TAB entry specified in the *gtab*. Memory references with this base register are directed to the associated TAB entry instead of the L1D, so only references associated with the TAB entry are allowed to use this base register within the loop. This also makes it unnecessary to alter preexisting instructions, as a hardware structure is used to store base register associations, and the base register is already a field in memory references.

Fig. 2.2 gives an example of the access pattern for a TAB line. To prepare for the first TAB reference, the *gtab* prefetches the first line to be accessed from the L1D. TAB references with constant strides can also cause a prefetch when the next reference address (calculated from the stride) will cross the line boundary. This makes it unnecessary to align the initial TAB address to the

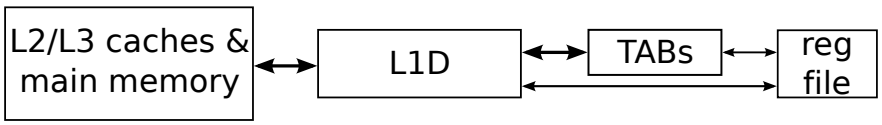


Figure 2.1: Memory hierarchy organization with TAB

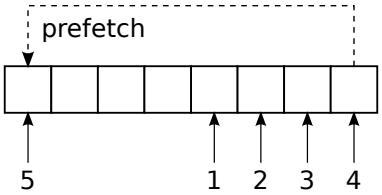


Figure 2.2: Example acccess pattern invoking a prefetch

```
int a[1000];

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

(a) Original summation 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) Generated RTL instructions

```
   gtab 1,r[7],4     #get TAB 1, stride 4
L1:
   r[2]=M[r[7]];     #redirect to tab 1
   r[3]=r[3]+r[2];
   r[7]=r[7]+4;
   PC=r[7]<r[6],L1;
   rtabs 1
```

(c) After compiler inserted TAB instructions

Figure 2.3: TAB allocation for a constant stride reference

line boundary. As the next TAB reference address is always known, tag checks and DTLB accesses for each reference are also unnecessary. References continue to access the TAB in this manner until an *rtabs* instruction deallocates the TAB entry and removes the base register association.

Fig. 2.3 gives a high-level example of how TAB instructions would be generated for a simple loop. Fig. 2.3(a) shows an example loop that iterates over the elements of the integer array *a*. Fig. 2.3(b) shows the instructions generated from that loop. The instructions are represented in an *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`. The compiler then generates a *gtab* instruction before the loop, and an *rtabs* instruction afterwards, as shown in Fig. 2.3(c). The gtab instruction indicates to the processor to associate the register `r[7]` with TAB entry one. Each memory reference using `r[7]` will instead access this TAB entry. A stride of four will be used to calculate addresses for prefetching. The *rtabs* instruction disassociates the register from the TAB entry.

The *gtab* instruction indicates extra information about the access patterns of TAB references which can further reduce energy. For instance, given a certain memory write pattern, we do not need to fetch lines from the L1D into the TAB, as the bytes will be overwritten anyway. This is made possible because of the level of control the TAB gives to the compiler. This encoded information is referred to later as the *type info* in Sec. 6. The *gtab* instruction also indicates when a TAB entry needs two lines instead of one. Called the *extra line* field, this field and its uses are described in Sec. 3.1.

In order to direct the hardware appropriately for prefetching lines from the L1D, *gtab* instructions also indicate information about which memory references cause a prefetch, and is explained in further detail in Sec. 4.3.

CHAPTER 3

HARDWARE SUPPORT FOR TAB OPERATIONS

This section describes the necessary hardware for TAB support. Sec. 3.1 describes the additional hardware structures required to support the TAB. Sec. 3.2 gives the structure and describes the fields of the two new instructions required to operate the TAB. Sec. 3.3 describes the small, necessary changes to the L1D for TAB support.

3.1 TAB Organization

The TAB consists of a number of hardware structures, including the data buffer itself. Fig. 3.1 gives an overview of the TAB organization. The *register array* stores the base register number for each TAB entry. The *TAB valid window* is a circular buffer which indicates if a TAB entry is valid. The circular buffer can also indicate if the TAB entry is valid for the current function call. The *register array* and *TAB valid window* are used together to determine which TAB entry, if any, is associated with a memory access. The base register number of the memory reference is compared against all register numbers in the *register array* in parallel. These results are ANDed with the bits in the current window of the *TAB valid window*. If the base register matches the register of a TAB entry and that TAB entry is valid, then this TAB entry will be accessed on the next cycle. The *TAB valid window* and how it supports function calls is described in more detail in Sec. 4.2.

Each TAB entry has an assortment of associated metadata. One field, the *index*, is used to associate a TAB entry with one of the data lines in the buffer. Each line in the buffer also has associated metadata. The line metadata is logically separate from the TAB metadata because the line metadata changes for each line pulled into the TAB. Furthermore, a single TAB entry may be linked to multiple lines. Since each line has unique metadata, a single TAB entry may have two associated line metadata sets. Although the hardware may implement the metadata as a unified strucuture, it is best to think of the metadata as separate. Fig. 3.2 shows an expanded view of the TAB and line buffer metadata structure. The width of the fields are listed on top; depending on the system, these numbers may vary. The numbers given are the widths used in our implementation.

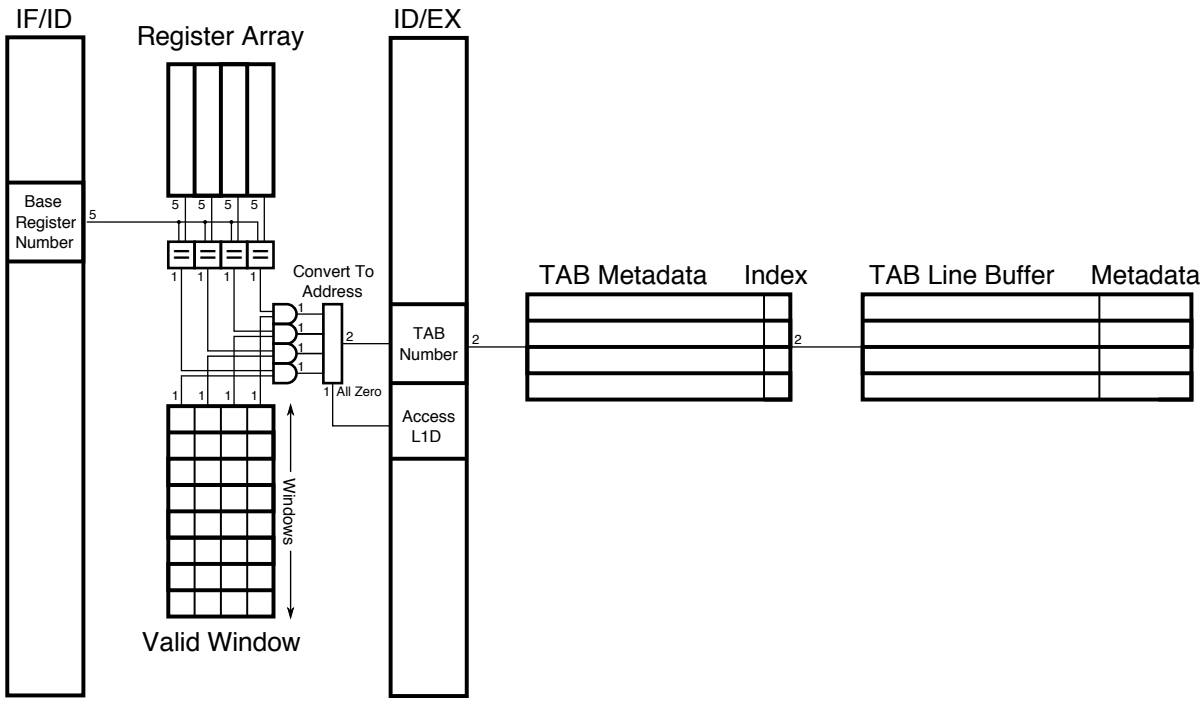
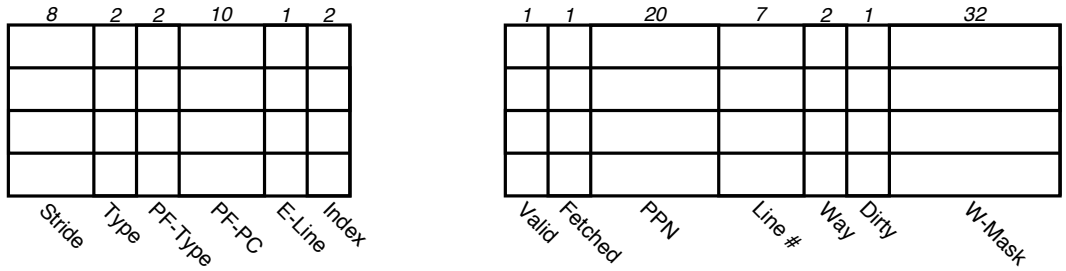


Figure 3.1: TAB hardware overview. *The pillars shown represent the set of pipeline registers between the indicated stages. The small numbers indicate the bit width of the lines. The "=" gates are compare gates. The "Convert to address" gate converts a 4 bit field with a single bit set to a proper 2 bit proper address. If all bits are 0, the "L1D Access" control signal is set, which will direct the memory reference to the cache.*



(a) TAB metadata

(b) Line buffer metadata

Figure 3.2: TAB metadata structure overview.

As seen in Fig. 3.2(a), the TAB metadata includes the *stride*, *type info*, *prefetch type*, *prefetch PC*, *extra line*, and *index* fields. The *stride* information is used to determine if a prefetch is required for a given access. As described earlier, the stride is added to the current reference's address to determine if the next reference will cross the line boundary. The *type info* bits control how data is transferred from the L1D to the TAB (for energy saving purposes), and is described further in Sec. 6. The *prefetch type* is a two bit field which indicates if all loads, all stores, all loads and all stores, or a single reference cause a prefetch. For instance, if the "all loads" bit is set, any load directed to this TAB entry will perform the prefetch check (it will not necessarily prefetch a line). If a single instruction causes a prefetch, the *prefetch PC* field is checked against the least significant bits of the current PC to determine if the current instruction needs to perform the prefetch check.

The *extra line* bit indicates if this TAB entry needs to use two lines instead of one. If an extra line is needed, both the current TAB line and the one immediately after will be treated as one. The *register array* will associate registers with the first TAB line of the pair, and the least significant bit of the L1D *set* field will determine which of the two TAB lines to use. An *extra line* may be needed when multiple references are directed to a TAB entry, but are accessed out of order and span two lines. Instead of prefetching back and forth and wasting energy, we use two lines and prefetch only the line needed next.

Finally, the *index* field determines which of the four line buffers are associated with this TAB entry. This is done so that multiple TAB entries can share the same line, which may be necessary to avoid cache conflicts. If two TAB entries each had their own copy of the same line, changes to each line may not be merged correctly, and the TAB entry may not contain the most up-to-date data. For instance, if lines are not shared, when one TAB entry writes to a line while another TAB entry reads from that same line, the second TAB entry will be reading old data from the L1D.

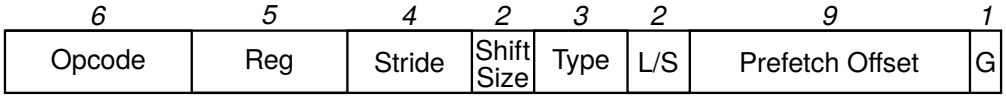
Fig. 3.2(b) shows the line buffer metadata, which includes the *valid*, *fetched*, *PPN* (Physical Page Number), *line number*, *way*, *dirty*, and *write mask* fields. The *valid* bit determines if the line in the TAB entry is still valid, and is separate from the TAB valid bit. If the line is evicted from the L1D, this bit is updated to reflect that the data in the TAB entry is invalid. L1D line evictions require each TAB line's *line number* field to be checked in parallel against the evicted line number. Since the check is made in parallel and only performed when the L1D line resides in the TAB (see Sec. 3.3), the overhead for evictions is extremely minimal.

The *fetched* bit simply states whether or not the line has been fetched from the L1D. For instance, if a TAB entry performs a prefetch, this bit is cleared until the line has been pulled in from the L1D to avoid TAB references accessing the wrong data. Without the fetched bit, if the next memory reference tries to access the fetching TAB entry before the next line is pulled in, it will be reading the previous line's data. The *PPN* and *line number* make up the current high-order bits of the address of the TAB line. When a prefetch occurs, the PPN is appended to the next sequential line number to directly access the L1D without going to the DTLB, which greatly reduces the amount of DTLB lookups. Storing the *PPN* requires a DTLB lookup, but this only occurs on a *gtab* instruction or when any prefetch crosses a page boundary, which infrequently occurs. The *line number* and *way* fields together give the exact location within the L1D of the line being used in the TAB. This allows memory references which were not directed to the TAB but which share the same line to determine which TAB entry holds the line. These *interferences* are discussed further in Sec. 3.3. This combination also removes the need to perform tag checks within the L1D when the TAB entry performs a writeback, which reduces energy overhead. The *dirty* bit and *write mask* bit-mask control how and when data is written back to the L1D. The *dirty* bit simply states that the data in the TAB line is different than the L1D and should be flushed if the line is invalidated or evicted. The *write mask* indicates which bytes have been altered so that only the bytes which have been modified in the TAB line are written back to the L1D. For instance, if we are iterating over an array of one byte characters and we only write every other character, the write mask will decrease the number of L1D bytes written back per line used in this TAB entry by 50%.

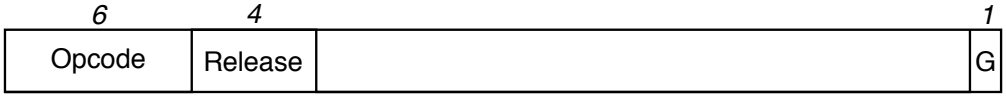
3.2 ISA Modifications

In order to support the TAB, the instruction set architecture (ISA) must be changed to include the new *gtab* and *rtabs* instructions. However, these can be implemented with a single opcode, as a single bit field within the instruction can differentiate a *gtab* from an *rtabs* instruction. The instruction formats for a MIPS-like 32-bit instruction set can be seen in Fig. 3.3

The *gtab* instruction format depicted in Fig. 3.3 (a), does not have an immediate field. Thus, in order to properly perform the first prefetch on the execution of a *gtab* instruction (pulling in the first L1D line into the TAB), the register given must contain the actual address for the first reference. This may require a few additional instructions to add an offset to the register before the



(a) TAB metadata



(b) Line buffer metadata

Figure 3.3: ISA modifications to support TAB accesses

gtab instruction, but these additional instructions have a negligible performance impact as they are performed outside of loops. In our original implementation, the gtab instruction had a proper immediate field instead of the *prefetch PC* field. This was because the original implementation used bits from loads and stores to indicate prefetches, thus they didn't need to follow a pattern. In this way, the gtab instruction could construct the proper starting address without altering the register, just like a load or store. Luckily, the need for such initial address construction is very rare, as strided references almost always use just the base register.

The *stride* and *shift size* are used to compute the actual stride stored in the TAB metadata. In order to support a wider range of values, the actual stride is calculated as *stride* << *shift size*. For instance, when accessing an integer array, the stride will always be a multiple of four (assuming a four byte integer). As such, we can simply set the *shift size* to two and use the four bits of *stride* information as the upper bits of the stride. Allocating a TAB entry with an overall stride that be represented with this system is unlikely, as nearly all common types have a size which is a power of two. It is even more unlikely given an L1D line size of 32 bytes, as strided references must fit within the line to save energy. Only arrays of structures with an odd assortment of data may have issues; for instance, most data types won't have a strange size like 15 bytes. Arrays of structures with an assortment of four or eight byte variables are more common and can easily fit into this system. Arrays of common types such as one byte characters, two byte shorts, four byte integers, four or eight byte pointers, and eight byte floats can always fit given a reasonable 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* gives the prefetch scheme (see previous section), and the *offset* is used to calculate the TAB metadata *prefetch PC* field, which is calculated by multiplying the *offset* by

the instruction width and adding it to the current PC. The signed nine bit *offset* field limits the distance between the *gtab* and prefetching reference instruction; this is why the *prefetch PC* TAB metadata field only needs to be ten bits. Using two's compliment, this nine bit field limits the distance between the gtab instruction and the prefetch reference to a range of -256 to 255 machine instructions. This is more than enough for most purposes; in fact, we never encountered an instance in our benchmark suite where this range was not sufficient. We use a signed value because loop pre-headers can be either above or below the loop itself; this is dependent on the compiler and the particulars of the code generation. The *G* bit simply indicates whether this is a *gtab* or *rtabs* instruction. Fig. 3.3 (b) depicts the *rtabs* instruction format, which only has a bitfield to indicate which TABs to deallocate. The *G* bit is set to zero for *rtabs* instructions.

3.3 L1D Changes

If a TAB line is evicted in the L1D, it must also be evicted in the TAB to preserve the inclusion property. However, the TAB is much smaller than the L1D, so L1D evictions have a low chance of evicting TAB lines. Instead of checking all TAB line numbers against the evicted L1D line number for every eviction, we extend each L1D line with two bits: the T and I bits. The T bit specifies that this line resides in the TAB, while the I bit (intercept) specifies whether non-TAB accesses to this line should be directed to the TAB. The T bit is used to maintain the inclusive property of the TAB: only evicted L1D lines with the T bit set will perform a TAB entry deallocation. Upon deallocation, the L1D *line number* and *way* are compared against each TAB entry to determine which one to deallocate. Since the TAB is small, evictions requiring a TAB entry invalidation are infrequent and the overhead of performing these checks is manageable.

If a regular (non-TAB) load or store accesses an L1D line which also resides in the TAB, it may need to be redirected to access the TAB line instead (because the TAB has the most recent version of the line). This is called an *interference*, and infrequently occurs. When an interference occurs, 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, however some TAB lines are guaranteed not to interfere with regular loads and stores. Thus, we use a separate bit (the I bit) to specify the lines for which regular references should be directed to the TAB. The TAB sets the I bit on lines based on the TAB entry's *type info* metadata. Now that the new TAB

system associates base registers to TAB entries, interferences are almost non-existent. It is difficult to naturally construct a situation where an interference occurs, however there are still edge cases. For instance, if a pointer is used to access data in an array separately from the strided references, and the pointer so happens to point to a memory location which currently resides in a TAB entry, it will cause an interference. The compiler is able to detect when situations like this might occur (even if they do not actually happen), and sets the I bit accordingly.

CHAPTER 4

TAB OPERATIONS

The following sections describe the main TAB operations: allocation, deallocation, and prefetching. Special actions must be performed on function calls for proper TAB operation, which is discussed in Sec. 4.2.

4.1 TAB Allocation and Deallocation

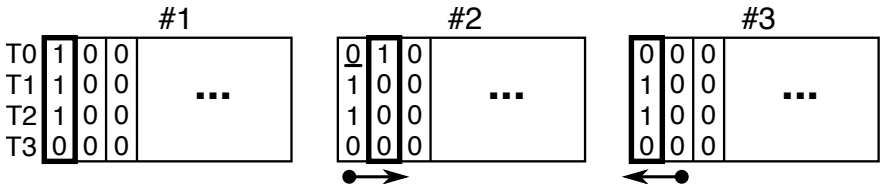
At the start, all TAB entries are invalid across all windows. When a *gtab* instruction is encountered, it allocates the TAB entry specified in the instruction by prefetching the first line to be accessed and marking the TAB entry as valid. Upon allocation, the register given in the *gtab* instruction is used to update the *register array* to associate the given register to the given TAB entry. If the TAB entry to be allocated is already valid at the point of allocation, the existing TAB entry is deallocated first. To deallocate a TAB entry, the associated lines in the buffer are flushed back to the L1D if the lines are dirty, and the TAB entry is marked invalid. In the case of a *gtab* performing a deallocation, the TAB entry is kept valid. When flushing the line, the TAB uses the *write mask* metadata field to write only the bytes which were altered from the L1D. If a TAB entry is already invalid when it is deallocated, the *dirty bit* will not be set, and nothing will happen. The *rtabs* instruction is used to explicitly deallocate one or more TAB entries.

4.2 Supporting Function Calls

Due to the association of registers to TAB entries, the TAB entries cannot stay active across function calls. The base register may be the same, but the actual address will almost certainly be different, and thus memory references would be accessing the wrong line if directed to a TAB entry from a previous function. For instance, assume an array is being accessed with a base register `r[0]`, and that `r[0]` is associated with a TAB. After a function call, `r[0]` is now being used to access a completely different array. If the register association is kept, accesses to this new array would be incorrectly directed to the TAB associated with `r[0]`, which does not contain the new array.

```
Function A()                    Function B()
{                               {
   Allocate TAB 0,1,2              Allocate TAB 0
#1 ←•      ...                  #2 ←•      ...
   B();                            Release TAB 0
#3 ←•      ...                  }
   Release TAB 0,1,2
}
```

(a) Example pseudo-code for valid window. Example windows in part (b) are labeled as bold numbers in code.



(b) Valid window at various points from part (a). Bold sections are the current window; empty space is unused windows.

Figure 4.1: Valid window code example

To remedy this, the *TAB valid window* acts as a circular buffer in a manner similar to register windows. Each window of the buffer has a bit for each TAB entry to indicate which TAB entries are valid for that window. The buffer has a number of windows (in our implementation, we use eight); function calls will shift the current window pointer forward, while function returns will shift the current window pointer back. Because it is a circular buffer, shifting forward when at the end of the buffer will wrap around to the beginning, and vice-versa for shifting back at the front of the buffer. A pointer is kept which points at the current window. The TAB entry is valid if a valid bit is set in any window, but the entry is only valid for the current function call if the set bit is in the current window. Memory references check the current window validity, while TAB allocation and deallocation check the general validity (valid in any window). Memory references use current window validity because they should not use a TAB entry unless it is valid in the current function call, but TAB allocation and deallocation use global validity because they deal with an associated TAB line, whose validity is completely distinct from the current window's validity.

With the window system, a new set of valid bits is used per function call. Only the valid bits are unique per call; the TAB metadata and line buffers are not altered. A TAB entry can only be valid

in one window, as allocating a TAB entry in a different window will overwrite the metadata and line buffer. The valid window enables TAB entries *not* used in new function calls to remain active upon return, while invalidating those that *are*. Since the TAB metadata and the current window validity is kept intact, any TAB entries that are not used in a function call can continue to be used after the function returns. With inter-procedural analysis and a heuristic for allocating TAB entries that are least likely to interfere with allocations in future function calls, TAB allocation collisions can be minimized. Invalidating the TAB entries in a new window (only for that window) means that even if a base register matches a TAB associated register, the TAB entry will appear invalid in the current context, which forces the reference to correctly go to the L1D. Fig. 4.1 gives an example of the window system in use. Fig. 4.1(a) depicts the TAB operations of two functions: A() and B(). Function A() requests TAB entries zero, one, and two, then calls function B(). Function B() requests only TAB entry zero. Fig. 4.1(b) depicts the *valid window* structure at various points during execution. Just before the call to B(), the current window is zero, and TAB entries zero, one, and two are valid for window zero. After B() allocates TAB entry zero, the valid bit shifts to window one, which is now the current window, while the other TAB entries remain valid in window zero. This is because B() requested a TAB entry that was already valid, thus TAB entry zero was deallocated before being allocated again in window one. When a TAB entry is deallocated, it clears the valid bits for that TAB entry across all windows, as there is no more usable data within the buffer. This also ensures that TAB entries are only valid for one window. Finally, only TAB entries one and two are valid after the call to B(), as TAB entry zero was used for window one. For the rest of A(), memory references can continue to use TAB entries one and two.

When the window is advanced, valid TAB entries in the new window are deallocated. This is done to avoid conflicts arising from function call depths being deep enough to reuse an old window. Were this not to happen, old valid bits from a previous function call could cause memory references in the current call to use old TAB lines, which would result in errors. For instance, assume TAB entry zero is allocated in some function. This function then calls further functions until the window wraps back around to our current position. Assume no intermediate functions used TAB entry zero. If we do not clear out the old TAB entries in our new window, it will appear as though this entry is still valid. This could incorrectly direct references to TAB entry zero, causing data errors.

The original TAB implementation did not have a special system for maintaining TAB entries across function calls. TAB entries were not tied to a register, so they could be safely preserved on a function call. Instead of specifying the TAB to allocate in the *gtab* instruction, the original system treated the TAB as a circular buffer and allocated TAB entries in a "last in, first out" order. If there were too many TAB entries (for instance, allocating after a function call), the TAB would deallocate the oldest TAB entry and allocate the new TAB in its place. Now that the current implementation ties the base register to the TAB entry, TAB entries can absolutely not be used in a different stack frame than the one it was created in. This requires either deallocating all TAB entries on a function call, or preserving some form of past state. Since the former would render the TAB useless in any loop with a function call, the latter was chosen. During the development of the current implementation, we worked with a window system which saved the full set of register associations per function call. This structure had too much energy usage overhead, so it was restructured into the current valid window system. Although the current system requires a four-way parallel compare to check the register association, the comparisons are small and the overall energy usage is much lower than the full register association window.

4.3 Prefetches

Prefetching is the act of copying a line from the L1D into the TAB when necessary. Unlike the L1D, which may or may not have the requested line and which pulls lines from the L2D as the lines are accessed, the TAB assumes it will have the proper line at all times. This is accomplished with prefetching: the current TAB memory reference causes the TAB entry to pull a new line *now* if necessary so it is already available when the next TAB memory reference needs it. Prefetching means there are no conventional misses when accessing the TAB; plus, if there are enough cycles between the prefetch and the next TAB reference, the TAB may never have to wait on a line, unlike the L1D where a "miss" is inevitable. If there is a dirty line already associated with a TAB entry at the point of a prefetch, the original line is flushed in a similar manner to TAB deallocation before the new line is brought in. The first prefetch for a new TAB entry is performed by the *gtab* instruction in order to prepare for the first TAB memory reference. After this, the following prefetches are performed automatically based on the pattern specified in the *prefetch type* metadata field and whether or not it is necessary to prefetch at the given moment. Prefetches are necessary

when it is detected that the following TAB memory reference will cross the line boundary. This is done by adding the stride to the current address and checking if the line portion of the address differs from the current TAB line. This is a simple operation that requires only a small adder, as we are only checking the final carry out from the addition. However, this should not be performed for every TAB reference; some are within conditionally executed code or followed by references which should still access the same line. A TAB reference in conditional code cannot perform a prefetch because the following TAB reference will be impacted by whether or not the conditional reference was executed. If the conditional reference causes a prefetch, the following reference will access the new line, however if it does not, the following reference will access the old line. In order to let a TAB entry know which instructions cause a prefetch, the *prefetch type* field is used to indicate whether the TAB entry should perform a prefetch check on all loads, all stores, all load and stores, or a single reference. It is important to note that if the compiler cannot fit the TAB reference prefetch scheme into one of these modes, it should not generate TAB instructions, as the correct execution cannot be ensured. Nearly all possible TAB allocations fit into one of the four mentioned schemes; most of any reduced TAB usage comes from mismatching TAB base registers rather than the prefetch scheme not fitting.

If the TAB entry is set to perform the prefetch check for only a single reference, the TAB entry uses the *prefetch PC* metadata field to determine if the prefetch check should be performed for the current reference. As stated previously, this field is computed upon the execution of a *gtab* instruction by multiplying the given offset (the number of machine instructions between the *gtab* instruction and the prefetch reference) and multiplying it by the addressable width of instructions, which is then added to the current PC. If a TAB entry does not need to perform prefetches after the initial *gtab* prefetch (for instance, when the TAB references are address invariant), the offset field given in the *gtab* instruction is set to zero. This causes the *prefetch PC* field to be the least significant bits of the *gtab's* PC, which is outside of the loop. With the *prefetch type* set to a single instruction, this effectively disables prefetches, as no memory reference can have the same PC as the *gtab* instruction.

Accessing the TAB requires fewer DTLB accesses than accessing the L1D, which can reduce energy usage. The TAB accesses the DTLB on a *gtab* instruction to store the PPN in the TAB

metadata. With the PPN stored, only TAB references whose prefetch crosses a page boundary need to access the DTLB, although this is a rare occurrence.

TAB entries have a level of indirection when accessing the line buffer: the buffer is accessed using the *index* metadata field. This enables multiple TAB entries to access the same line buffer within the TAB when necessary, as there should not be two instances of the same L1D line within the buffer. When a prefetch occurs, it checks the prefetch line number against the valid lines within the TAB. In order to minimize performance loss, this check is performed in parallel. If there is no match, it simply allocates a new line and sets the index to the new line. If there is a match, it does not allocate a new line and does not perform the prefetch; instead it simply sets the index to point to the existing line. TAB metadata remains unique, as it is associated with each entry instead of each line, however the line buffer metadata is shared. If multiple TAB entries point to the same TAB line and one requests a prefetch, that TAB entry prefetches into a new TAB line, and the line is no longer shared. Since the TAB line is still being used by at least one TAB entry, the line is not flushed back to the L1D. A counter is used per line to keep track of how many TAB entries are sharing the current line; a line is only deallocated if no more TAB entries are using it. Changes made by an old TAB entry which used to share the line will remain in the TAB line until the current TAB entry associated with the line finally flushes it. This situation can cause L1D interference issues if the L1D is not properly informed: for instance, if the original TAB entry does not have interferences, but the new entry does, the interferences may not be captured in the TAB line. To remedy this, when a TAB line is shared the interference fields for the two TAB entries sharing the line are `OR` combined so that if at least one TAB entry has interferences, the associated line will too. The L1D line is updated if necessary to reflect a new I bit value when shared.

CHAPTER 5

COMPILER ANALYSIS

The compiler does not need to perform interprocedural analysis or additional code transformations in order to support the TAB, which makes it relatively straightforward to implement. The compiler needs to know how many lines are in the TAB buffer and how large the lines are. Without these, the compiler will not know how many TABs it can use within a loop nest or whether the stride will fit properly within a line.

Only two instructions need to be generated in order to support TAB operations: the *gtab* and *rtabs* instructions. The compiler detects memory references with a constant stride or that are loop-invariant and sets up the TAB environment for each reference by inserting a *gtab* before the loop and an *rtabs* afterwards. If multiple TAB entries are allocated for a given loop, they can all be deallocated at the same time with a single *rtabs* instruction. The compiler generates loop preheader and postheader blocks for these instructions if they do not already exist. Additional arithmetic instructions may need to be inserted in the preheader to store the calculated initial address in the register used by the *gtab*. These instructions simply add a constant to the register before the *gtab*, then subtract the same constant afterwards. For instance, if the first TAB reference is +160 off from the base register `r[5]`, the generated instructions would be `r[5] = r[5] + 160` before the *gtab* and `r[5] = r[5] - 160` afterwards. All of the additional instructions needed for the TAB occur before and after the loop, and have little impact on execution time. Fig. 5.1 gives a brief description of the algorithm used to generate TAB instructions.

5.1 References with Constant Strides

Memory references with constant strides have the form `M[reg]` or `M[reg+disp]`, where `reg` follows a pattern of `reg = reg` $\pm$ `constant` and `disp` is simply a displacement from the address in `reg`. This pattern arises from applying loop strength reduction to strided accesses in data structures, such as sequential access to an array. For instance, if we sequentially access every element in an integer array, the constant stride pattern would look like `reg = reg + 4` (assuming

```
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 5.1: Compiler analysis algorithm for TAB allocation

a four byte integer). If the compiler detects a memory reference with an appropriate strided access pattern, it will attempt 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 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, we may miss a prefetch and have the wrong line for the next access. (3) The stride must be less or equal to half the L1D line size in order to obtain the energy benefits from using the TAB. If the stride is greater than half the L1D line size, there will be times where only a single reference is directed to a TAB line before another line has to be pulled in. Since it takes additional energy to pull in this line, it would cost more to read the one reference from the TAB line than to read it directly from the L1D. (4) The base register of the reference must not be used as a base register in any other loop memory references. This is because the base register is used to direct references to either the TAB or the L1D; if the base register for a TAB reference is also used in non TAB references, the non TAB references will also be caught by the TAB, causing data errors.

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. If references had different strides, they would eventually span different lines, which breaks the TAB's prefetch system. (3) All references must share the same base register so they are directed to the same TAB entry. It is possible to associate more than one base register with a given TAB entry, but this would require more equality checks when determining which TAB entry (if any) to direct the reference to. Since this increases energy usage, we use just one base register per TAB entry. (4) The reference or references 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 should be no larger than the L1D line size. This differs from the previous requirement of "half or less" because we are guaranteeing that multiple references will access the line before swapping it out. Thus, even if the stride is equal to the line size and the line is swapped out on every loop iteration, we know at least two or more references will access the line before it is swapped. (6) The maximum distance between any two references cannot be larger than the L1D line size. An example situation where this might be used is loop unrolling, where a single memory reference within the original loop gets expanded into many. They all access the same structure, and assuming the loop memory reference originally had a constant stride, the multiple references will have it too. Sometimes a set of references can span two cache lines; for instance, due to an out of order access pattern. In this situation, the compiler can set the *extra line* field of the *gtab* instruction to signify that the 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 where an extra line is not required for a TAB entry with multiple references. The references must be accessed in order and in the same direction as the stride. Furthermore, the distance between each reference must be the same, including the distance between the last reference in one iteration and the first in the next. In this special case, a single TAB line buffer can be used, as it is no different than a single reference with a constant stride. Fig. 5.2 illustrates an example of loop unrolling and how the TAB can capture multiple references. Fig. 5.2(a) contains code that sums the elements of an integer array. Fig. 5.2(b) shows the loop after unrolling, and Fig. 5.2(c) shows the generated instructions. Note that because the first reference has an offset, we

```
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) Summation after loop unrolling

```
  r[6]=r[6]-12;     #precalculate base register
  gtab 1,r[6],4     #get tab 1, stride 4
  r[6]=r[6]+12;
L1:
  r[2]=M[r[6]-12]; #tab 1 load, prefetch
  r[3]=M[r[6]-8];  #tab 1 load, prefetch
  r[4]=M[r[6]-4];  #tab 1 load, prefetch
  r[5]=M[r[6]];    #tab 1 load, prefetch
  r[7]=r[7]+r[2];  #r[7] is sum variable
  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 1
```

(c) Loop unrolled references directed to TAB

Figure 5.2: Capturing loop unrolled references in the TAB

must generate instructions to store the first address within `r[6]` before the execution of the *gtab* instruction.

5.2 References with Loop-Invariant Addresses

Although loops containing references with loop-invariant addresses are not very common, the potential energy benefits from directing them to the TAB are large. TAB entries allocated for these types of references only need to perform at most one prefetch and one writeback, which means one DTLB access and at most two L1D accesses. Depending on the access type stored in the *type info* metadata, only one L1D access may be needed. Fig. 5.3 gives example code that allocates a reference with a loop-invariant address to a TAB entry. Fig. 5.3(a) is a loop with a function call to *scanf()* which prevents the global variable *sum* from being stored in a register. Fig. 5.3(b) shows the generated instructions, where *sum* has been allocated to a TAB entry. Notice that no reference

```
sum = 0;  //global variable

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

(a) Original loop

```
   r[14]=sum;
   gtab 1,r[14],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 5.3: TAB usage with a loop-invariant memory address

causes a prefetch, and references can be used even outside the loop. The variable n is not allocated to a TAB entry because the address is passed to another function.

5.3 TAB Allocation Heuristics

Loop nests often have the potential to allocate more TAB entries than the hardware can support. In these cases, the compiler must choose a subset of the memory references that would be the most profitable. The metric we use to determine the "best" memory references is the estimated number of avoided L1D accesses per loop iteration. The number of loop iterations is often unknown at compile time, therefore we do not incorporate it as part of the calculation.

We begin by assigning a value of one saved L1D access per TAB reference. This value is then increased for references in inner loops, as they should be referenced more than those in outer loops. This value increase can be set arbitrarily; use whatever you find most profitable for your specific needs. It can even be a compiler option should the need for such micro optimizations arise. If the reference is in a conditionally executed section, this value is halved. This conditional reference reduction can also be set arbitrarily and tuned to specific needs. For instance, a program may benefit from an even greater reduction if the conditional references within critical sections are almost never run. In general, halving works well. The sum of these values for references directed to a given TAB entry becomes the *estimated references* value. Even though a TAB entry may have a high number of *estimated references*, it may not save many L1D accesses due to a large stride or frequent prefetches. Thus, the *estimated references* value is adjusted by subtracting the sum of

the estimated L1D loads and stores per loop iteration. The estimated L1D loads and stores can each be a multiple of the stride divided by the L1D line size, where the multiple depends on the number of accesses per loop. For instance, assume we have a potential TAB entry with a single reference and a stride of one. If the L1D line size is 32 bytes, the TAB line is accessed 32 times before a prefetch is required. A prefetch performs both a load and a store to the L1D; thus for every 32 accesses we *would* have directed to the L1D, the L1D is actually accessed two times. Per loop iteration, this is 30/32 saved L1D accesses, or 1-(1/32+1/32), where 1/32 is the stride divided by the line size. Finally, this adjusted value is divided by the number of lines required to allocate this TAB entry (either one or two depending on whether the TAB entry needs an extra line). The following equation gives an overview of the metric calculation:

$$(estim\_refs - (L1D\_loads + L1D\_writes))/\#TAB\_lines \tag{5.1}$$

Fig. 5.4 gives an example application of the heuristic. The loop given in Fig. 5.4(a) accesses four arrays, each with a different type. The type affects the stride; although the *for* loop increments the index variable i by one, each type is a different size and thus the actual stride through memory will differ between types. Fig. 5.4(b) shows the values needed to estimate the avoided L1D accesses for each variable which can be placed in a TAB entry. Assume the L1D line size is 32 bytes. Both d and a are referenced twice per iteration, but the *estimated reference* value for a is 1.5 because one reference is inside an *if* statement. Furthermore, this conditionally executed reference cannot cause a prefetch, meaning a requires an extra line. The stride for g is zero because it is a loop-invariant global address, and the value cannot be stored in a register outside the loop due to the function call *f()*. This inability to use a register is why the reference to g is a TAB candidate. Since g does not require any prefetches due to having a loop-invariant address, it does not access the L1D. The variable s does not require any L1D loads because it is only written. Avoiding unnecessary reads is a feature of the *type info* metadata system, and is described in more detail in the next section. The variable b does not require any L1D writes because it is never updated (the dirty bit is never set and thus flushes do not occur during prefetches). If the TAB only has four entries, the compiler would not generate TAB instructions for a, as it gives us the least amount of avoided L1D accesses.

```
// global declaration
double g;
...
// local declarations
char s[100];
short b[100];
int a[100], i, sum, t;
double d[100];
...
```

```
// loop referencing all 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) Example code to be estimated

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	0.97 = (1.0 - (0 + 0.031)) / 1
b	1.0	1	2	2/32	0	0.94 = (1.0 - (0.63 + 0)) / 1
a	1.5	2	4	4/32	4/32	0.63 = (1.5 - (0.13 + 0.13)) / 2

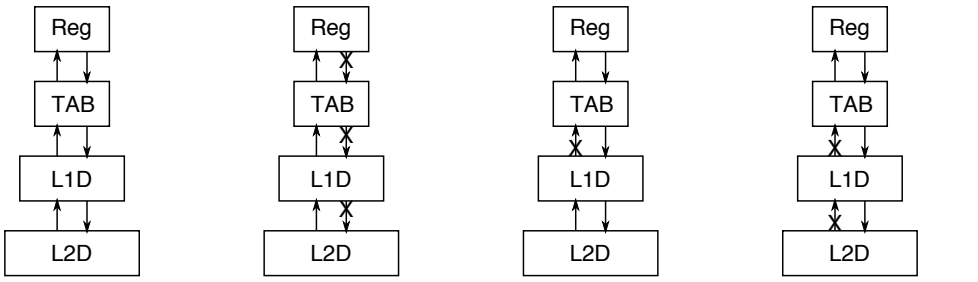
(b) Calculation of avoided L1D accesses

Figure 5.4: Example of estimating saved L1D accesses

CHAPTER 6

AVOIDING UNNECESSARY DATA TRANSFERS

The upper levels of the memory hierarchy use dirty bits to prevent costly write backs to the lower levels, reducing energy usage in the process. The TAB continues this approach with the *write mask* field, which only writes back altered bytes. This obviously doesn't impact strided references that write all bytes in a line, but it can have a significant impact if the line is read-only. For sparsely written lines, the potential benefits are offset by the fact that the line isn't being accessed as much; if the line is written sparsely, the stride is most likely large. The TAB can further reduce energy usage by using the *type info* bits to specify how data will be written to the TAB line. These bits indicate if bytes referenced in a TAB entry are always written to first (write-first), or if all bytes in the TAB entry are overwritten (write-contiguous). Write-first TAB entries do not need to pull lines in from the L1D, as reads in this TAB entry use no L1D data. The write-mask is used to write back only altered bytes, preserving the unaltered bytes in the L1D. Although it is rare to write before reading while striding through memory, write-first is particularly useful for saving energy in write only TAB entries. Write-contiguous TAB entries pull lines from neither the L1D nor the L2D, as the entire line will be overwritten anyway. Write-contiguous is a subset of write-first; were it not, the TAB would need the data from the L1D for the first read. Fig. 6.1 gives an overview of how various TAB access patterns reduce transfers in the memory hierarchy.



(a) Read/Write (b) Read only (c) Write first (some bytes) (d) Write first & contiguous (all bytes)

Figure 6.1: Exploiting the TAB to avoid L1D and L2 accesses

CHAPTER 7

EVALUATION FRAMEWORK

The TAB system is evaluated using the VPO compiler [6] and the SimpleScalar simulator [2]. VPO performs the analysis required for the TAB and generates MIPS/PISA target code, which SimpleScalar simulates in a time-accurate, five-stage, in-order pipeline. The processor configuration used in SimpleScalar is detailed in Table 7.1. The decision to use a four line TAB is motivated by the simulations performed in our previous work [4], which determined the optimal TAB configuration for energy reduction. Energy values for the TAB structures are estimated by synthesizing the structures with the Synopsys Design Compiler [17]. A 65nm 1.2-V low-power CMOS cell library is used. Probabilistic estimations on the design netlist are done with Synopsys PrimeTime [18]. The L1D and DTLB energy estimates are taken from a recent work [5] which estimated accurate energy usage for the same 65nm technology. These energy values (shown in Table 7.2) are multiplied by event counters in SimpleScalar to produce the overall energy for a simulation. Our results are produced by running 20 MiBench benchmarks [8] with large datasets through the model processor. The benchmarks come from the six categories shown in Table 7.3.

Table 7.1: Processor configuration

BPB, BTB	Bimodal, 128 entries
Branch Penalty	2 cycles
Integer&FP ALUs, MUL/DIV	1
Fetch, Decode, Issue Width	1
L1D & L1I	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

SimpleScalar comes with various simulators at varying levels of simulation, however its cycle accurate simulator only comes as an out of order super scalar pipeline. We made changes to force it into an in-order traditional five stage pipeline, which involved shrinking various queues to only

Table 7.2: 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
Meta (TAB) / (line buffer)	1.4 pJ / 4.5 pJ
Extra Structures (reg array) / (valid win)	0.7 pJ / 2.3 pJ

Table 7.3: 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

handle one item and adding a workaround for the super-scalar simulation. SimpleScalar calculated some simple energy values for the L1D; combined with power estimations from CACTI, these made up the energy estimates in the original TAB implementation. For the new implementation, we estimate more accurate energy values with a more modern fabrication for both the L1D and the TAB using Synopsys Design Compiler. Because the new estimates model more events (such as distinguishing between the energy required for a load and the energy required for a store), more events needed to be tracked during simulation.

CHAPTER 8

RESULTS

The results from simulating the 20 MiBench benchmarks are detailed here. Each graph shows the results with the TAB enabled as a percentage of the results without the TAB. The second to last bar in each graph is the current average of all the benchmarks, and the last bar shows the average of the original TAB implementation. We first present the current results, then we compare the old results against the new ones.

8.1 Current TAB Results

Fig. 8.1 details the percentage of memory references directed to the TAB, and what portion of hits were strided or invariant. The average is 33.4%, but there are outliers, like *rsynth*, which have few hits. *Rsynth* is a special case: many functions in this benchmark accept overlarge structures as value parameters. These parameters contain so much data that they must be accessed through the stack pointer instead of registers. These parameter references cannot be directed to the TAB because they cannot all fit within a single TAB entry, and they cannot be split up between the L1D and the TAB because the hardware needs the base register to be exclusive to TAB references. Passing large structures by value is not a standard programming practice and has little impact on the overall TAB hit percentage.

Fig. 8.2 details the breakdown of L1D accesses while the TAB is active. On average, the L1D is accessed 30.9% less. The overhead of prefetching and writebacks is extremely small, amounting to an extra 2.7% when compared against L1D accesses without the TAB. Fig. 8.3 gives the execution time as a percentage of cycles. Despite the additional instructions generated to access the TAB, benchmarks run on average 1.7% faster with the TAB enabled. This is mostly due to the TAB being accessed in the execute stage, thus avoiding some load hazards.

Fig. 8.4 shows the energy usage when the TAB is enabled. The values shown are a percentage of the total L1D and DTLB energy without the TAB. We include the extra energy dissipated by the Level-One Instruction Cache (L1I) to determine the impact of the extra instructions required

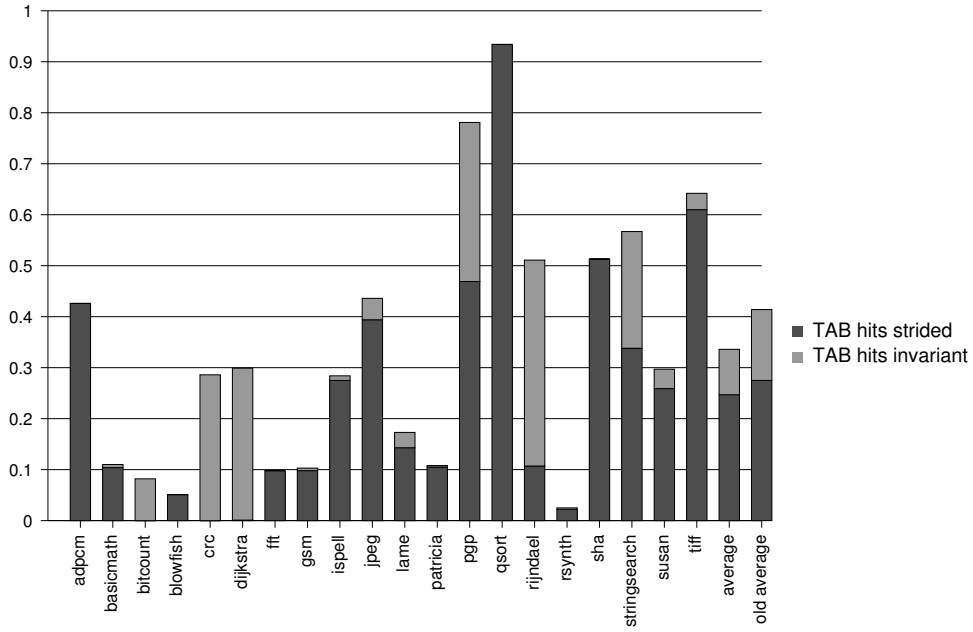


Figure 8.1: TAB utilization for all benchmarks

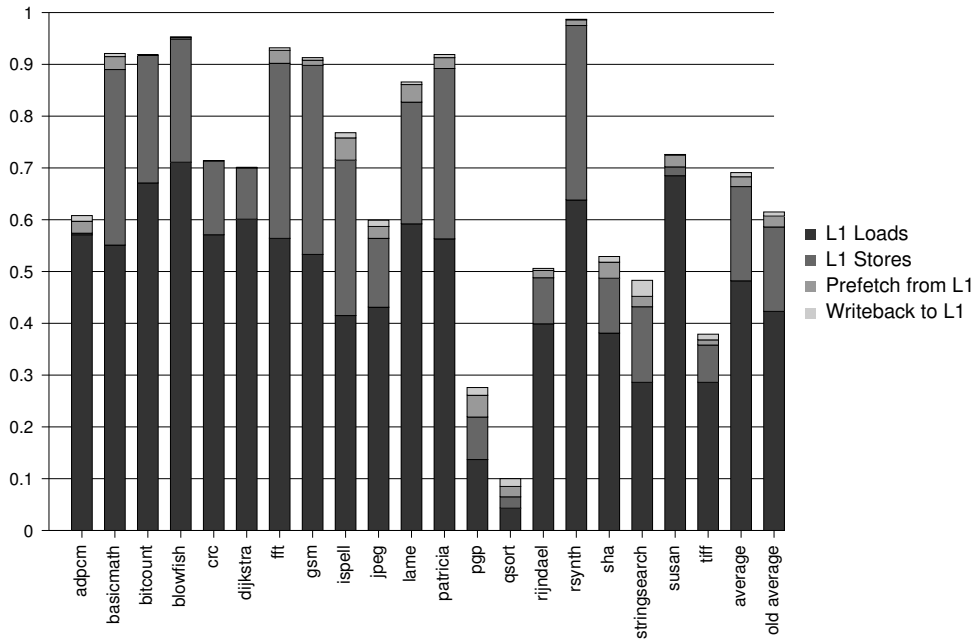


Figure 8.2: L1D access breakdown

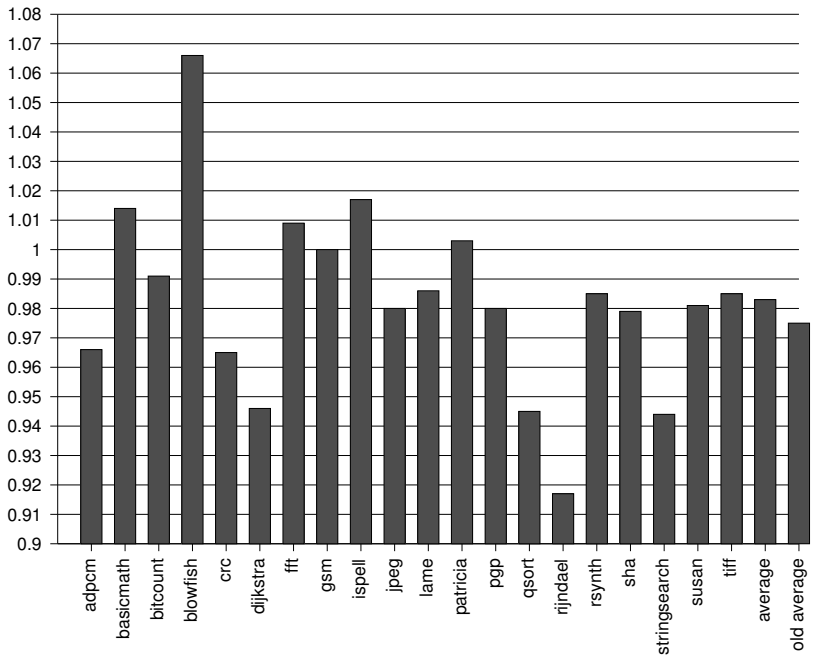


Figure 8.3: Execution time with the TAB enabled

for TAB operation. These instructions only account for an average of 2.9% of the total L1D and DTLB energy without the TAB. The extra hardware required for the TAB has an average overhead of 3.0%, giving a total TAB operation overhead of 5.9%. After overheads are applied, the average L1D/DTLB energy saved is 21.8%. Some benchmarks use slightly more energy with the TAB enabled; this is due to a combination of a forced overhead for every memory reference, increased instruction count, and a low TAB hit rate in these benchmarks. Even if the TAB is not being accessed, loads and stores must check the *register array* and *valid window* each time they are executed to ensure they are directed to the proper structure. The TAB increases energy usage infrequently however, and the average is reasonable considering the only ISA change is one extra opcode.

8.2 Comparing Results with Original Implementation

The original TAB implementation took bits from the immediate field of load and store instructions to direct them to a TAB entry or the L1D. The new implementation does not use any load

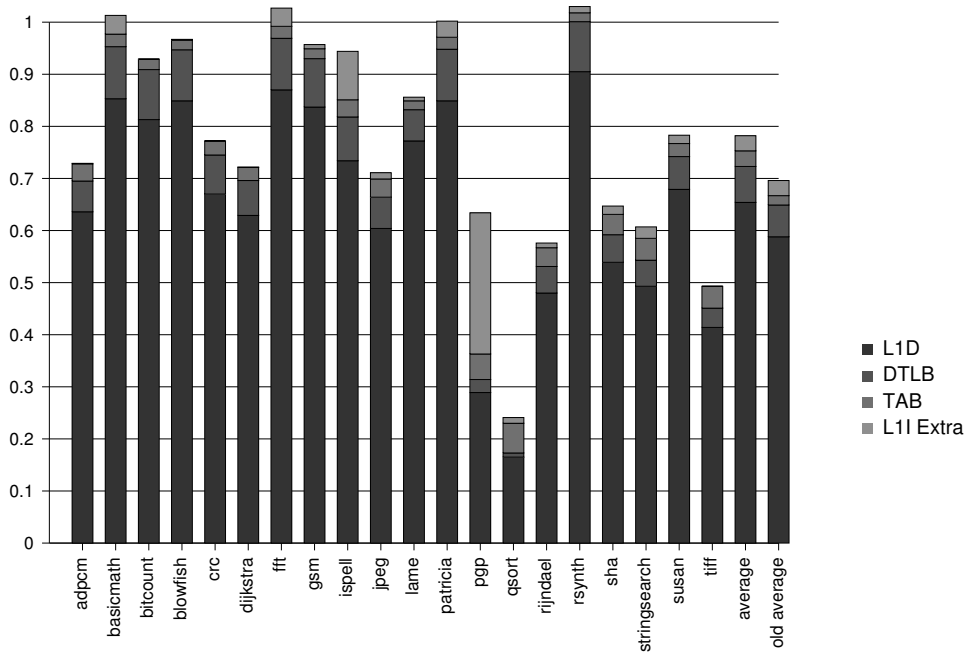


Figure 8.4: Percentage of energy dissipated with TAB enabled

or store bits, which sacrifices flexibility for simpler ISA changes. Because only the base register is used to direct references, extra restrictions must be applied to TAB allocation. For instance, a TAB entry cannot be allocated if there are memory references outside the TAB entry with the same base register. Restrictions like this reduce the percentage of TAB hits from 41.4% to 33.6%, as seen in Fig. 8.1. Subsequently, the L1D accesses have increased from 61.6% to 69.1% as seen in Fig. 8.2. Neither strided nor invariant TAB references are particularly impacted by the new restrictions: their frequency is reduced by roughly the same amount.

Fig. 8.3 shows an increase in execution time from 97.5% to 98.3%, which is due to decreased TAB hits and increased generated instructions. The original implementation used the immediate field as part of the starting address calculation, but the new implementation uses the immediate field for other data. For the new implementation, the starting address must be stored in the *gtab* register. Extra instructions must be generated in some cases to temporarily store the offset in the *gtab* register. The impact is minimal however, as the execution time is still lower than running without the TAB.

The decrease in energy savings from 30.4% to 21.8% as seen in Fig. 8.4 is caused by the decrease in TAB hits compared to the original implementation and the forced overhead of the extra hardware structures. The decreased TAB utilization accounts for 86.04% of the decreased energy savings, while the extra TAB hardware accounts for 13.95%. The new implementation's increased instruction count makes an extremely minimal impact, accounting for only 0.01% of the difference in energy savings.

CHAPTER 9

RELATED WORK

Witchel *et al.* propose a hardware-software design for data caches, called a direct address cache (DAR) [19]. 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. Subsequent memory references guaranteed to access the same line will reference that register to avoid the tag check. The compiler for the DAR uses some immediate bits of the load/store operations for control purposes, much like the original TAB approach. With the new TAB approach, such invasive ISA changes are unnecessary. The DAR reduces energy usage by avoiding accesses to the tag array and activating only a single way of the L1D data array for memory references guaranteed to access a specified L1D line. The TAB approach also avoids tag checks, but it accesses the smaller and more power-efficient TAB structure as opposed to a single way of the much larger 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. 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 statically determined, 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 L1D accesses occur as normal. The TAB approach avoids both DTLB and L1D accesses, which reduces more data access energy usage.

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 [16]. 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 continously 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]. Historically, filter caches reduce energy usage at the expense of a significant performance penalty due to their high miss rate, however we recently proposed an implementation of a filter cache which reduces both energy and execution time [5]. Because TAB accesses require less power than a filter cache access, they can be used together to further reduce energy. TAB accesses can exploit compile time detected sequential locality, while the filter cache automatically detects other locality.

Nicolaescu *et al.* propose a power saving scheme for associative data caches [14]. The way information of the last Nth 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. This technique still requires the L1D to be accessed on every data access. 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 scratch pads in which variables are explicitly allocated or copied [3, 12, 11]. Scratchpads can reduce energy usage 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 described in this paper. 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 the TAB is not limited to just setting a line to zero: any value can be written.

CHAPTER 10

CONCLUSION

In this paper, we present an alternative to the original TAB system, which reduced L1D and DTLB energy usage by almost a third, but required changes to the structure of loads and stores. The newly presented implementation reduces said energy usage by over a fifth, but is backwards compatible with preexisting code and only requires one additional opcode. Both the old and new implementations reduce energy by replacing L1D accesses with accesses to a smaller, more power efficient structure which does not require tag checks. As with the previous implementation, execution time is reduced by accessing the TAB earlier in the pipeline and using prefetching to offset the stalls incurred by L1D misses.

REFERENCES

[1] *Programming for AIX, assembler language reference, instruction set (AIX 5L Version 5.3)*, 3 edition, July 2006.

[2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[3] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter 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] Alen Bardizbanyan, Peter Gavin, David Whalley, Magnus Själander, Per Larsson-Edefors, Sally McKee, and Per Stenström. Improving data access efficiency by using a tagless access buffer (tab). In *IEEE Int. Symp. on Code Generation and Optimization*, pages 269–279, February 2013.

[5] Alen Bardizbanyan, Magnus Själander, David Whalley, and Per Larsson-Edefors. Designing a practical data filter cache to improve both energy efficiency and performance. *ACM Transactions on Architecture and Code Optimization*, 10(4), December 2013.

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

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

[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, December 2001.

[9] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark 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, 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, March 2004.

[12] 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.

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

[14] Dan Nicolaescu, Babak Salamat, Alex Veidenbaum, and Mateo Valero. Fast speculative address generation and way caching for reducing l1 data cache energy. In *IEEE Int. Conf. on Computer Design*, pages 101–107, 2006.

[15] Curtis Storlie, Joe Sexton, Scott Pakin, Michael Lang, Brian Reich, and William Rust. Modeling and predicting power consumption of high performance computing jobs. Technical report, Los Alamos National Laboratory, North Carolina State University, 2015.

[16] 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.

[17] Synopsys. Design compiler, 2013.

[18] Synopsys. Primetime, 2013.

[19] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanović. Direct addressed caches for reduced power consumption. In *34th ACM/IEEE Int. Symp. on Microarchitecture*, pages 124–133, December 2001.

BIOGRAPHICAL SKETCH

Carlos Sanchez recieved a Bachelor of Science in Computer Science from Florida State University in December of 2012. He graduated Magna Cum Laude and pursued a Master's degree in Computer Science, also at Florida State University. His research dealt with compilers and energy efficiency, and he has submitted a paper based on this manuscript in the ACM journal: *Transactions on Architecture and Code Optimization*. He has many years of experience as a teaching assistant, where he made every effort to ensure each student's success. Carlos spent the summer of 2015 on a research trip in Sweden, where he worked at Chalmers University of Technology. Here, he continued his research on processor efficiency with some of the top international minds in the field.