Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache

Stephen Hines, David Whalley, and Gary Tyson
Florida State University
Computer Science Dept.
Tallahassee, FL 32306-4530
{hines,whalley,tyson}@cs.fsu.edu

Abstract

Very small instruction caches have been shown to greatly reduce fetch energy. However, for many applications the use of a small filter cache can lead to an unacceptable increase in execution time. In this paper, we propose the Tagless Hit Instruction Cache (TH-IC), a technique for completely eliminating the performance penalty associated with filter caches, as well as a further reduction in energy consumption due to not having to access the tag array on cache hits. Using a few metadata bits per line, we are able to more efficiently track the cache contents and guarantee *when hits will occur in our small TH-IC. When a hit is not guaranteed, we can instead fetch directly from the L1 instruction cache, eliminating any additional cycles due to a TH-IC miss. Experimental results show that the overall processor energy consumption can be significantly reduced due to the faster application running time and the elimination of tag comparisons for most of the accesses.*

1 Introduction

Embedded systems are often subject to tighter power constraints due to their portable nature and thus increased dependence on batteries. Instruction fetch is a prime area to investigate since previous studies have shown that the instruction cache (IC) can be responsible for a significant portion of the energy consumption, up to 27% of the total processor power requirements on a StrongARM SA110 [17]. Although traditional caches are often found on embedded processors, many also include specialized cache structures to further reduce energy requirements. Such structures include filter caches [13, 14], loop caches [15], L-caches [2, 3], and zero-overhead loop buffers (ZOLBs) [6]. Techniques like drowsy caching [12] can also be applied to further reduce power consumption.

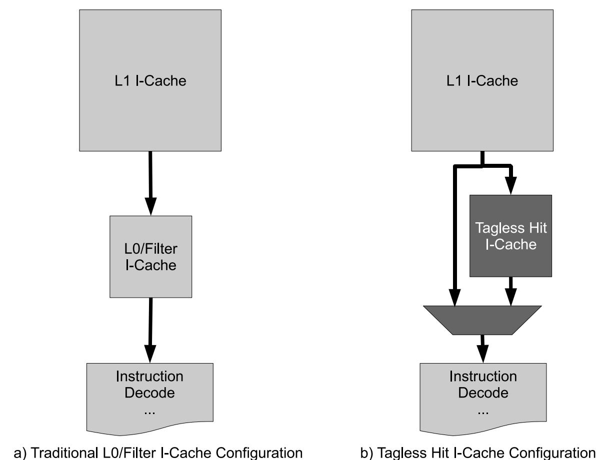


Figure 1. Traditional L0/Filter and Tagless Hit Instruction Cache Layouts

Filter or L0 instruction caches (L0-IC) are small, and typically direct-mapped caches placed before the L1 instruction cache (L1-IC) for the purpose of providing more energy-efficient access to frequently fetched instructions [13, 14]. Since the L0-IC is accessed instead of the L1-IC, any miss in the L0-IC incurs an additional 1-cycle miss penalty prior to fetching the appropriate line from the L1-IC. Figure 1a shows the traditional layout of a small L0/filter IC. Although an L0-IC reduces the requirements for fetch energy, these miss penalties can accumulate and result in significant performance degradation for some applications. It is important to note that this performance loss will indeed reduce some of the energy benefit gained by adding the L0-IC due to having to actively run the processor for a longer period of time. The inclusion of an L0-IC in a memory system design is essentially a tradeoff providing a savings in fetch energy at the expense of longer execution times.

In this paper we propose an alternative configuration for a small instruction cache to be used in conjunction

with an L1-IC. This configuration is shown in Figure 1b and is related to previous research that has sought to bypass the small cache based on *predictions* [20]. We have renamed the small IC as a Tagless Hit instruction cache or TH-IC. Using just a few specialized metadata bits, the TH-IC supplies a fetched instruction only when the instruction is *guaranteed* to reside in it. As a side effect of the way in which guarantees are implemented, we no longer require tag comparisons on hits, hence the term "Tagless Hit". The small size of the cache and its novel use of metadata is what facilitates the ability to make guarantees about future cache hits, while still retaining the ability to operate and update in an energy- and performance-conscious manner. A TH-IC of similar size to an L0-IC has nearly the same hit rate and does not suffer a miss penalty since the TH-IC is not used to fetch an instruction when a miss may occur. In essence, the TH-IC acts as a filter cache for those instructions that can be determined to be hits in the TH-IC, while all instructions that cannot be guaranteed to reside in the TH-IC access the L1-IC without delay. Additionally, the energy savings is greater than using a L0-IC due to the faster execution time (the TH-IC has no miss penalty), the reduction in ITLB accesses (the TH-IC can be accessed using bits from the portion of the virtual address that is unaffected by the translation to a physical address), as well as the elimination of tag comparisons on cache hits (since we do not need to check a tag to verify a hit).

This paper makes the following contributions. We show that a simple direct-mapped L0/filter instruction cache can be replaced with a Tagless Hit instruction cache (TH-IC) that does not require a tag comparison when an instruction can be guaranteed to be in the TH-IC. Additionally, the TH-IC can be bypassed when a hit is not guaranteed, resulting in the elimination of the L0-IC performance penalty. We also show that the hit rate for a TH-IC is only slightly lower than an L0-IC of equivalent size. This leads to a technique for instruction cache design that features both higher performance and greater energy efficiency than the current state of the art in filter cache design and implementation. It also makes the inclusion of a small, energy-conscious instruction cache feasible even for high performance processors.

The remainder of this paper is organized as follows. First, we describe the design of the Tagless Hit instruction cache in detail, including various configurable options that facilitate the effective scaling of circuit complexity. Next, we present our experimental framework along with a thorough evaluation of the TH-IC. Third, we review related research in reducing instruction fetch energy. Finally, we present our conclusions.

<table>
<tr><td colspan="3">TH–IC accesses</td></tr>
<tr><td rowspan="2">guaranteed hits (or just hits)</td><td colspan="2">potential misses</td></tr>
<tr><td>fm</td><td rowspan="2">true misses</td></tr>
<tr><td colspan="2">potential hits</td></tr>
</table>

Figure 2. Terminology Used to Describe Tagless Hit Instruction Cache Accesses

2 Tagless Hit Instruction Cache (TH-IC)

This section presents an overview of the design of the Tagless Hit instruction cache (TH-IC). First, we describe the intuition behind removing tag comparisons from our small IC, as well as the principles involved in guaranteeing when the next instruction fetch will be a hit. Second, we explore the new IC metadata and the rules required for guaranteeing tagless hits. Finally, we propose four invalidation policies for efficiently managing the cache metadata.

2.1 Guaranteeing Tagless Hits

One of the key principles in the design of the TH-IC is the idea of bypassing the TH-IC when we are *not certain* that the requested instruction/line is resident in the TH-IC. This leaves three possibilities when an instruction is fetched: 1) it is guaranteed to be a hit in the TH-IC, 2) it resides in the TH-IC, but we were not sure so we directly accessed the L1-IC, or 3) it did not reside in the TH-IC, and we avoided the miss penalty by attempting to access it directly from the L1-IC. None of these cases involve any miss processing by the TH-IC, so the execution time will be unchanged by the inclusion of a TH-IC. Figure 2 illustrates the terminology we use to describe the types of accesses that can occur when fetching from a TH-IC. The *potential hits* and *true misses* reflect the hits and misses that would occur in an L0-IC with the same cache organization. A *guaranteed hit* represents the portion of the potential hits that the TH-IC can guarantee to reside in cache. A *potential miss* means that a hit is not guaranteed, and thus the requested line will instead be fetched from the L1-IC. We may miss opportunities to retrieve data from the TH-IC when it might reside there but cannot be guaranteed; however, any reduction in hit rate will be more than offset by the ability to avoid any TH-IC miss penalty. During potential TH-IC misses, we check whether the line fetched from the L1-IC is already available in the TH-IC. We use the term *false miss* (fm in Figure 2) to describe such an event, and *true miss* to indicate that the line is not in the TH-IC. The TH-IC approach works well when a significant fraction of the instructions that reside in the TH-IC can be guaranteed to reside there prior to fetch.

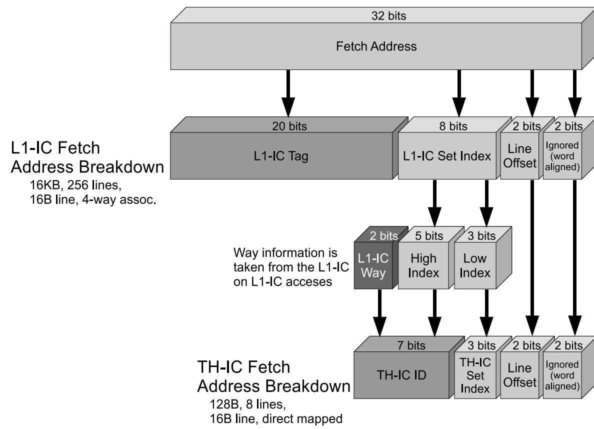


Figure 3. Fetch Address Breakdown

A breakdown of fetch addresses into their separate bitwise components for properly accessing the various instruction caches present in our memory hierarchy is shown in Figure 3. For this example, we use a 16 KB, 256 line, 16-byte line size, 4-way set associative L1-IC. We also use a 128 B, 8 line, 16-byte line size, direct-mapped TH-IC. Instructions are word-aligned, so the low-order two bits of any fetch address can be safely ignored. Two bits are used to determine the L1-IC line offset, while eight bits are necessary for the set index, leaving twenty bits for the tag. Two bits are again used to determine the TH-IC line offset, while three bits are used for the set index. In order to reduce the effective tag size of the TH-IC, we employ a subtle approach based on Ghose and Kamble's work with multiple line buffers [8]. Instead of storing a large tag for the remainder of the address in the TH-IC, it is sufficient to identify the corresponding line in the L1-IC by storing the set index and the location of the line in the set. Not storing the entire tag in the TH-IC is possible since the L1-IC is being accessed simultaneously and a true miss will occur if the L1-IC misses. The cache inclusion principle guarantees that any line in the TH-IC must also reside in the L1-IC. Thus by detecting an L1-IC hit *and* verifying the precise L1-IC line that corresponds to our TH-IC line, we can effectively determine whether we have a false miss.

The figure shows that we construct a TH-IC *ID* field that is made up of the additional high-order bits from the L1-IC set index along with two bits for specifying which line in the cache set is actually associated with this particular line address (it's "way"). When we are updating the TH-IC (on a potential miss), we are already accessing the L1-IC, so we only need to compare whether we have the appropriate set/way from the L1-IC already in the TH-IC. The miss check can be done by concatenating the two-bit way information for the currently accessed line in the L1-IC and the five high-order

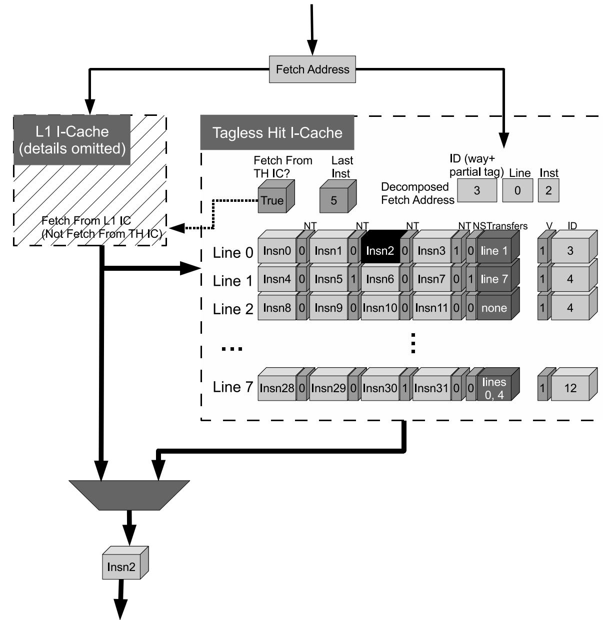


Figure 4. Tagless Hit Instruction Cache

bits of the address corresponding to the L1-IC set index, and comparing this result to the stored TH-IC ID of the given set. If these seven bits match, then the TH-IC currently contains the same line from the L1-IC and we indeed have a false miss. If these bits do not match, or the L1-IC cache access is also a miss, then we have a TH-IC true miss and must update the line data as well as the TH-IC ID with the appropriate way and high index information. The ID field can be viewed as a line pointer into the L1-IC that is made up of way information plus a small slice of what would have otherwise been the TH-IC tag. If the L1-IC were direct-mapped, the ID field would only consist of the extra bits that are part of the L1-IC set index but not the TH-IC set index. The cache inclusion property thus allows us to significantly reduce the cost of a tag/ID check even when the TH-IC cannot guarantee a "tagless" hit.

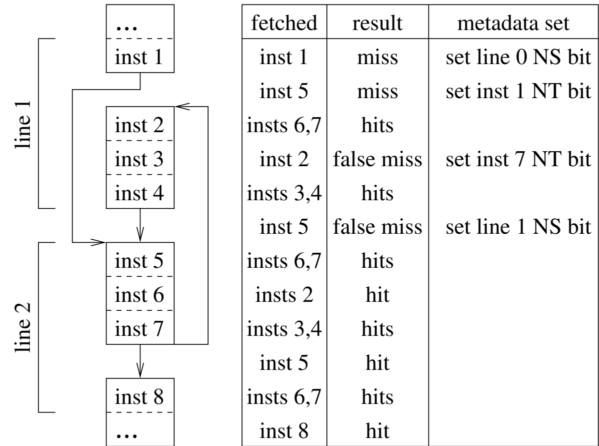
Figure 4 shows a more detailed view of an instruction fetch datapath that includes a TH-IC. The TH-IC has been extended to use additional metadata bits (approximately 110 total bits for the simplest configuration we evaluated). The first aspect to notice in the TH-IC is the presence of a single decision bit for determining where to fetch the next instruction from (*Fetch From TH-IC?*). This decision bit determines when the TH-IC will be bypassed and is updated based on the metadata bits contained in the TH-IC line for the current instruction being fetched, as well as the branch prediction status (predict taken or not taken). We also keep track of the last instruction accessed from the TH-IC (using the

pointer *Last Inst*). The last line accessed from the TH-IC (*Last Line*) can easily be extracted from the high-order bits of the last instruction pointer.

There are really two distinct types of access in the TH-IC or any other instruction cache for that matter: sequential accesses and transfers of control. If the predictor specifies a direct transfer of control (taken branch, call or jump), then the TH-IC will make use of the *Next Target* bit (NT), one of which is associated with each instruction present in the small cache. If the current instruction has its NT bit set, then the transfer target's line is guaranteed to be available and thus the next instruction should be fetched from the TH-IC. If the NT bit is not set, then the next instruction should be fetched from the L1-IC instead, and the TH-IC should be updated so that the previous instruction's target is now in the TH-IC. We discuss a variety of update policies and their associated complexity in the next subsection. In this figure, the last instruction fetched was *Insn5*, which resides in line 1. The branch predictor (which finished at the end of the last cycle's IF) specified that this was to be a taken branch and thus we will be fetching *Insn5*'s branch target, which is *Insn2*. The corresponding NT bit was set, so the TH-IC is going to be used to fetch the target instruction on this cycle. We thus fetch *Insn2* from line 0 of the TH-IC. Since the cache is direct-mapped, there is only a single line where this instruction can reside. Note that the tag/ID check is unnecessary, since the NT bit guarantees that this instruction's branch target is currently available in the TH-IC.

On a sequential fetch access (branch prediction is not taken), there are two possible scenarios to consider. If we are accessing any instruction other than the last one in the line, then we will always choose to fetch the next instruction from the TH-IC, since we know that the next sequential instruction in this same line will still be available on the subsequent access. This process is similar to the operation of a sophisticated line buffer [7]. If it is the last instruction in the line that is instead being fetched, then fetching the next instruction from the TH-IC will occur only if the *Next Sequential* bit (NS) is set. This bit signifies that the next line (modulo the number of lines) in the cache actually contains the next sequential line in memory. This is a behavior that line buffers do not support, since they only hold a single line at a time, and thus must always return to fetch from the L1-IC when they reach the end of the line.

Figure 5 shows an example that illustrates how instructions can be guaranteed to reside in the TH-IC. The example in the figure contains eight instructions spanning four basic blocks and two lines within the TH-IC. Instruction 1 is fetched and is a miss. The previous line's NS bit within the TH-IC is set since there was a sequential transition from line 0 to line 1. Instruction 5 is



fetched	result	metadata set
inst 1	miss	set line 0 NS bit
inst 5	miss	set inst 1 NT bit
insts 6,7	hits	
inst 2	false miss	set inst 7 NT bit
insts 3,4	hits	
inst 5	false miss	set line 1 NS bit
insts 6,7	hits	
insts 2	hit	
insts 3,4	hits	
inst 5	hit	
insts 6,7	hits	
inst 8	hit	

Figure 5. TH-IC Example

fetched after the transfer of control and it is also a miss. Instruction 1's NT bit is set to reflect that the target of instruction 1 resides in the TH-IC. Instructions 6 and 7 are fetched and are guaranteed to be hits since they are sequential references within the same line. Instruction 2 is fetched and it resides in the TH-IC, but it is a false miss since it was not guaranteed to hit in the TH-IC (instruction 7's NT bit is initially false). At this point, the NT bit for instruction 7 is set to indicate its target now is in the TH-IC. Instructions 3 and 4 are fetched and are hits due to the intra-line access. Instruction 5 is fetched and is a false miss (line 1's NS bit is false). Line 1's NS bit is set at this point indicating that the next sequential line now resides in the TH-IC. The instructions fetched in the remaining iterations of the loop are guaranteed to be hits since the TH-IC metadata indicates that the transitions between lines (line 1's NS bit and instruction 7's NT bit) will be hits. Finally, instruction 8 is fetched and will be a hit since it is a sequential reference within the same line.

The TH-IC exploits several nice properties of small, direct-mapped instruction caches. First of all, the nature of a direct-mapped cache allows a given fetch address to reside in only a single location. This facilitates the tracking of cache line contents and their associated interdependences. In addition to the elimination of tag/ID comparisons, the ITLB access can also be avoided on TH-IC hits. This is due to the virtual to physical address translation not affecting the page offset portion (12-bits for 4KB page size) of a fetch address. Since the indexing of the TH-IC is accomplished using bits from the page offset (and no tag comparison is required on guaranteed hits), we do not actually need to verify the translation. The update and invalidation policies of the TH-IC help to maintain these conservative principles by which hits can be guaranteed.

One special case to consider is the possibility of indirect transfers of control. If the current instruction to be fetched is a *jr* or *jalr* instruction (jump register or jump and link register in the MIPS ISA), then we cannot guarantee that the branch target is in the TH-IC since the address in the register may have been updated since the last time the instruction was executed. We instead choose to fetch from the L1-IC directly. Recognizing an indirect transfer of control is relatively simple and can be done by checking for only 2 instructions in the MIPS ISA. Fortunately, indirect transfers of control occur much less frequently than direct transfers.

2.2 Updating Tagless Hit Instruction Cache Metadata

There are really only two important steps in the operation for the TH-IC: *fetch* and *update*. Figure 6 shows a flow diagram that graphically depicts the operation of the TH-IC. The first step is the decision based on whether to fetch from the TH-IC or the L1-IC. *Fetch* is similar to traditional instruction fetch on a cache hit. *Update* replaces the concept of the traditional cache miss. The TH-IC performs an update whenever the instruction/line being fetched is not *guaranteed* to be in cache. This does not necessarily mean that the line is not available in the cache. Availability is checked by performing a tag/ID comparison within the TH-IC in parallel with the L1-IC fetch. On a false miss, the TH-IC need not write the cache line from the L1-IC, and does not need to *invalidate* any additional cache metadata either.

If the fetch is a true miss, however, we need to replace the appropriate line in the cache and update/invalidate various portions of the TH-IC. First of all, the new line needs to be written into cache from the L1-IC along with its corresponding TH-IC ID. The NS bit and the NT bits for each instruction in this line are cleared, as we cannot guarantee that any branch target or the next sequential line are available in cache. If we are replacing a line that is a known branch *target*, we need to invalidate the NT bits on all corresponding lines that may have transfers of control to this line. This requirement to manipulate metadata for multiple lines is not particularly onerous since the total number of metadata bits is extremely small. There are several possible schemes for keeping track of where these transfers originate, and four approaches are discussed in the next subsection on invalidation policies. We use the previous branch prediction's direction bit to determine whether we have fetched sequentially or taken a transfer of control. If the access was sequential, the previous line's NS bit is set since we are simply filling the cache with the next sequential line in memory. Note that the only time that we can have a sequential fetch causing a cache miss will be when we are fetching the first instruction in the new line. For transfers of control, we need to keep track of the last instruction fetched. If we are not replacing the line containing the transfer of control, we set the last instruction's NT bit to signify that its branch target is now available in the small cache.

Once the instruction is fetched, we need to update the last instruction pointer (and hence last line pointer), as well as determine whether the next fetch will come from the TH-IC or the L1-IC. It is also at this point that we need to determine whether the current instruction is an indirect transfer of control. Direct transfers of control do not change their targets, and this is why the NT bit is sufficient for guaranteeing that a target is available in cache. If we detect an indirect transfer, we need to steer the next fetch to the L1-IC, since we cannot guarantee that an indirect branch target will be unchanged. We also invalidate the last instruction pointer so that the next instruction will not incorrectly set the indirect transfer's NT bit.

We rely on the result of the current cycle's branch prediction to determine whether the next fetch is sequential or not. If a taken direct branch is predicted, then the corresponding NT bit for the current instruction fetch is used to decide whether to fetch from the TH-IC or the L1-IC. If instead it is a sequential access, then we will use the NS bit of the current fetch line if we are at the end of the line. If we are elsewhere in the line, the next instruction will be fetched from the TH-IC based on the line buffer principle. When we have a pipeline flush due to a branch misprediction, we choose to fetch from the L1-IC on the next cycle since we cannot guarantee that the TH-IC contains this potentially new address.

2.3 Tagless Hit Instruction Cache Invalidation Policies

Invalidation policies provide the TH-IC with the ability to efficiently update its metadata so that it operates correctly. Without a proper invalidation scheme, the TH-IC could possibly fetch incorrect instructions since we no longer do tag/ID comparisons for verification. In this section, we present four invalidation policies that vary in complexity from conservative approximations to more precise tracking of the relations between the transfers of control and their target instructions.

Figure 7 shows four sample line configurations for a TH-IC, where each configuration implements a particular policy. In each configuration, each line is composed of four instructions (Insn), a *Next Sequential* bit (NS), a *Next Target* bit (NT) for each instruction, a Valid bit (V), as well as an ID field. The number and size of the *Transfer from* bits (none, T, TLs, or TIs) is what distinguishes each configuration.

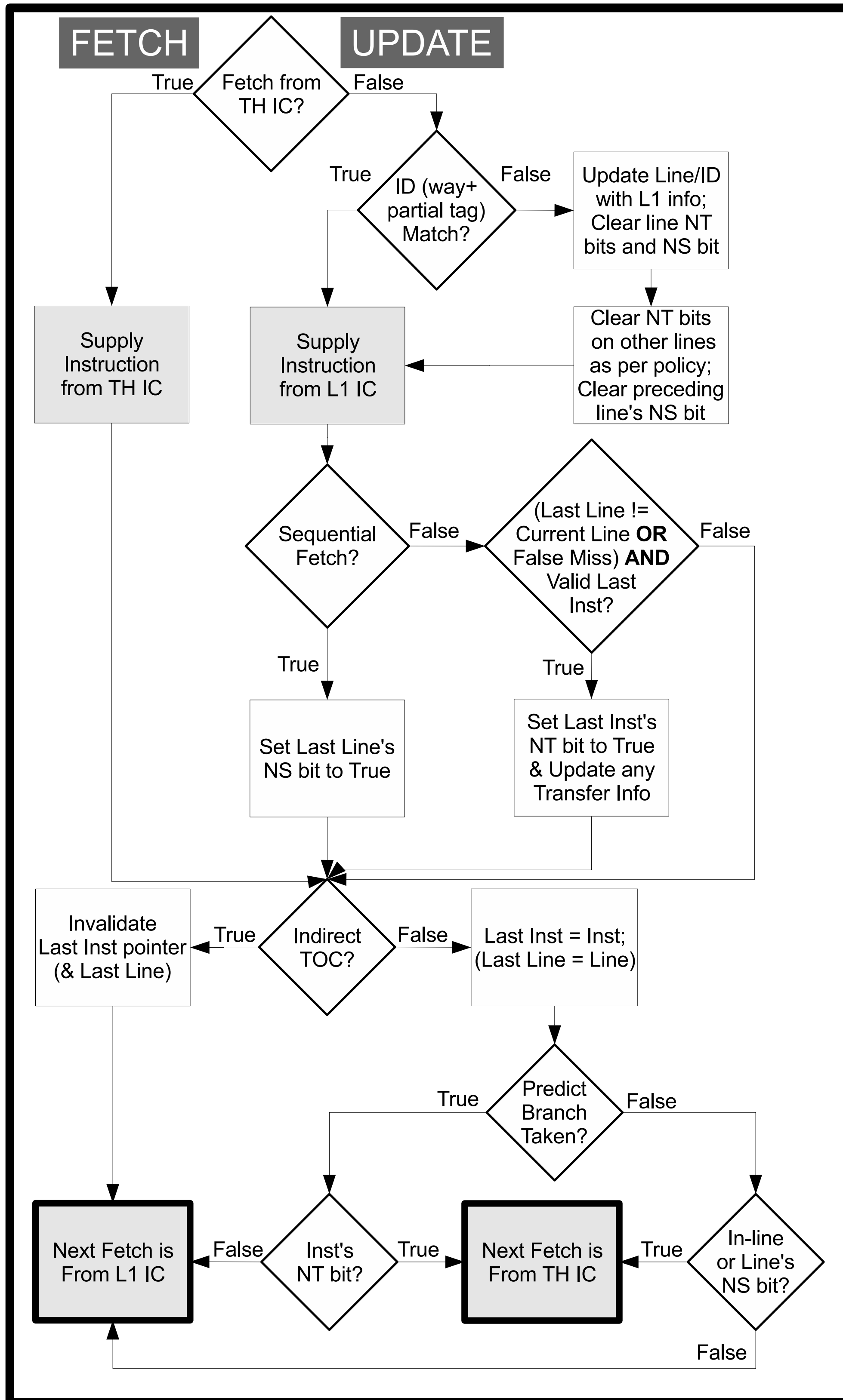


Figure 6. Tagless Hit IC Operation

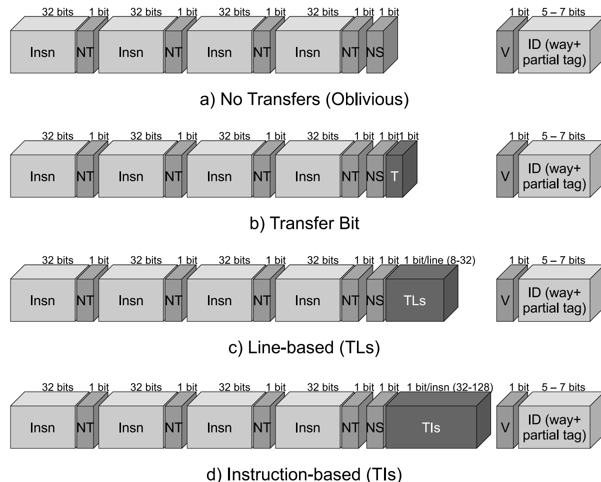


Figure 7. Tagless Hit Instruction Cache Line Configurations

The *Oblivious* case is shown in Figure 7a. In this configuration, there are no bits reserved for denoting the lines that transfer control to the chosen line. Under this policy, whenever any cache line is replaced, all NT bits in the TH-IC will be cleared.

Figure 7b shows the addition of a single *Transfer (T)* bit to each line representing that this line is a potential transfer of control target. This bit is set when any direct transfer of control uses this line as a *target*. Whenever we have a line replacement where the T bit is set, we need to clear all NT bits similar to the oblivious policy. The savings occur when replacing a purely non-target line (one that is not a known target of any direct transfers of control currently in cache), which does not necessitate the clearing of any of the other NT bits. This allows the existing NT metadata within all the lines not being fetched to continue to remain unchanged in cache. It is important to note, however, that any replaced line will always have at least its own NT bits cleared.

The configuration shown in Figure 7c adds 1 bit to each line for every line in the cache (*Line-based (TLs)*). The bit corresponding to a particular line in the TL field is set when there is a direct transfer of control to this line. When the line is replaced, all NT bits will be cleared in lines whose corresponding bit in the TLs is set.

Finally, Figure 7d shows the addition of 1 bit to each line for every instruction available in the cache (*Instruction-based (TIs)*). A bit corresponding to an individual instruction in the TI field is set when the instruction transfers control to the line. The only NT bits cleared when a line is replaced are due to the corresponding instruction-specified TIs. Of all the schemes proposed, this one is the most aggressive and requires the greatest area overhead (and hence increased energy consumption to operate). This scheme maintains almost perfect information about NT transfers of control for invalidation. It is only conservative in that replacing a line (and thus clearing its NT bits) will not clear the corresponding TIs still contained in any other lines.

Although we have proposed these four schemes, there are other possible variants. For instance, one could disallow the set of NT for intra-line transfers, thus saving 1 bit per line when using line-based transfer information, since the line itself would not need a self-referential bit. A similar approach could save additional bits with the instruction-based transfer metadata. One can also imagine a completely omniscient policy that clears other line's matching TIs when a line is replaced in the instruction-based policy. Each of these schemes requires greater complexity than the four simpler policies that we have presented. Further experimentation and analysis could help in developing even more area and energy efficient invalidation policies.

3 Experimental Setup and Evaluation

We used the SimpleScalar simulator for evaluating the TH-IC [1]. The Wattch extensions [5] were used to estimate energy consumption including leakage based on the *cc3* clock gating style. Under this scheme, inactive portions of the processor are estimated to consume 10% of their active energy. The machine that is modeled uses the MIPS/PISA instruction set, although the baseline processor is configured with parameters equivalent to the StrongARM. Table 1 shows the exact details of the baseline configuration used in each of the experiments. We refer to the TH-IC invalidation policies as follows in our graphs: TN - Oblivious, TT - Transfer bit, TL - Line-based Transfers, and TI - Instruction-based Transfers. Each policy or L0-IC designation is suffixed with a corresponding cache size configuration. The first value is the number of lines in the cache. We evaluate configurations consisting of 8, 16, and 32 lines. The second value is the number of instructions (in 4-byte words) present in each cache line. Since the L1-IC uses a 16-byte line size on the StrongARM, all of our configurations will also use 16-byte lines (4 instructions). Thus, the three small cache size configurations are 8x4 (128-bytes), 16x4 (256-bytes), and 32x4 (512-bytes). We also evaluate a tagless hit line buffer (TH LB), which guarantees hits for sequential instruction fetches within the same line. This is essentially a degenerate form of the oblivious TH-IC that uses only a single line (1x4) with no additional metadata bits.

Although the Wattch power model is not perfect, it is capable of providing reasonably accurate estimates for simple cache structures for which it uses CACTI [21].

Table 1. Baseline Configuration

I-Fetch Queue	4 entries
Branch Predictor	Bimodal – 128
Branch Penalty	3 cycles
Fetch/Decode/Commit	1
Issue Style	In-order
RUU size	8 entries
LSQ size	8 entries
L1 Data Cache	16 KB 256 lines, 16 B line, 4-way assoc. 1 cycle hit
L1 Instruction Cache	16 KB 256 lines, 16 B line, 4-way assoc. 1 cycle hit
Instruction/Data TLB	32 entries, Fully assoc., 1 cycle hit
Memory Latency	32 cycles
Integer ALUs Integer MUL/DIV Memory Ports FP ALUs FP MUL/DIV	1 1 1 1 1

Table 2. MiBench Benchmarks

Category	Applications
Automotive	Basicmath, Bitcount, Qsort, Susan
Consumer	Jpeg, Lame, Tiff
Network	Dijkstra, Patricia
Office	Ispell, Rsynth, Stringsearch
Security	Blowfish, Pgp, Rijndael, Sha
Telecomm	Adpcm, CRC32, FFT, Gsm

The structures involved in the evaluation of TH-IC are composed primarily of simple regular cache blocks and associated tags/metadata. Although the L0-IC and TH-IC may have differing functionality (tag checks vs. metadata updates), they remain very similar in overall latency and area. Writing of metadata bits can be viewed as a small register update, since the overall bit length is often short, even for some of the larger configurations.

Table 2 shows the subset of MiBench benchmarks we used for each of the experiments [9]. MiBench consists of six categories of applications suitable for the embedded domain in a variety of areas. Each benchmark is compiled and optimized with the VPO compiler [4], which yields code that is comparable in quality to GCC. All applications are run to completion using their small input files (to keep the running times manageable). Large input experiments were also done, yielding similar results to the small input files, so any further discussion is omitted. MiBench results are presented by category along with an average due to space constraints. Results are verified for each run to ensure that the application carries out the required functionality. Sanity checks are performed to ensure correct behavior and verify that the TH-IC does not unfairly use information that should not be available.

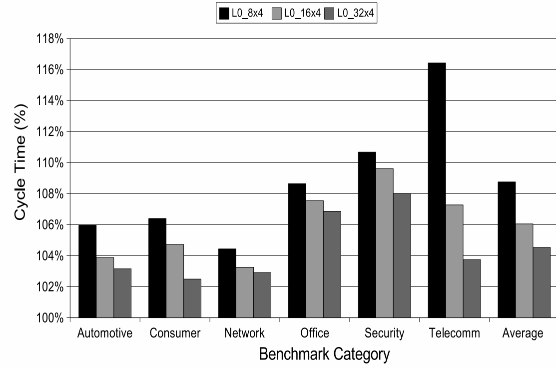


Figure 8. Performance Overhead of L0 Instruction Caches

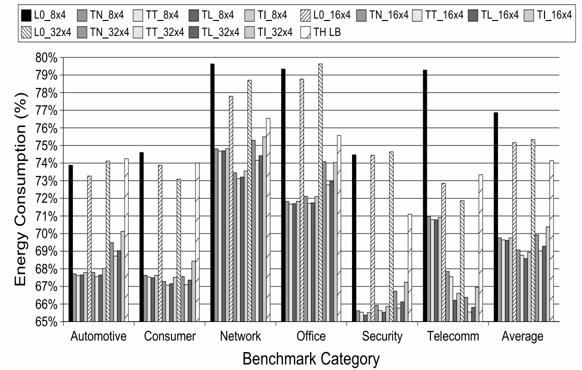


Figure 9. Energy Consumption of L0 and Tagless Hit Instruction Caches

Figure 8 shows the performance overhead for using a conventional L0-IC. Cycle time is normalized to the baseline case, which only uses an L1-IC. Results for the various TH-IC configurations are not shown, since they yield exactly the same performance as the baseline case (100%). The 128-byte L0-IC increases the average execution time by 8.76%, while the 256-byte L0-IC (L0_16x4) increases it by 6.05%, and the 512-byte L0-IC (L0_32x4) still increases the average execution time by 4.53%. The relative complexity of encryption and decryption procedures keeps the Security category's loop kernels from even fitting completely within the 512-byte L0-IC. The additional 1-cycle performance penalty of a cache miss when using an L0-IC can clearly accumulate into a sizable difference in application performance.

The energy consumption of the various L0-IC and TH-IC configurations are shown in Figure 9. Each of the TH-IC configurations outperform their corresponding L0-IC configurations. Similar to the execution cycles results, the Security and Telecomm benchmark cat-

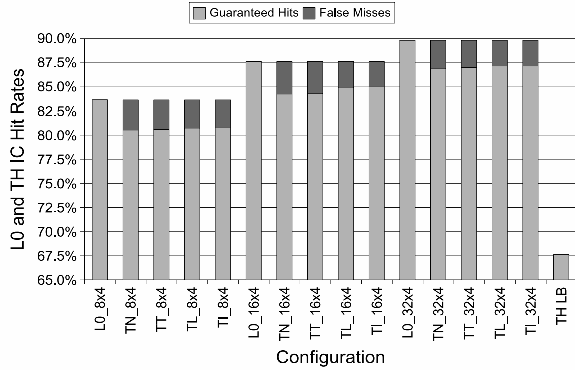


Figure 10. Hit Rate of L0 and Tagless Hit Instruction Caches

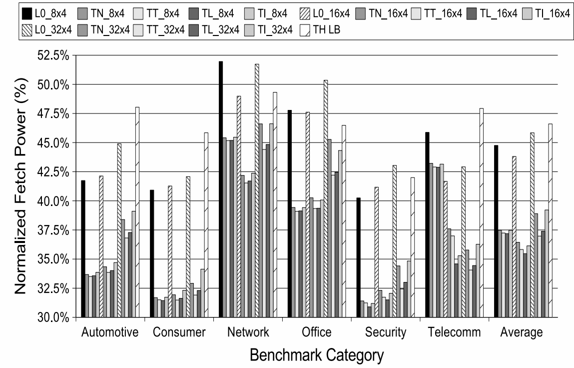


Figure 11. Fetch Power of L0 and Tagless Hit Instruction Caches

egories gain the most from the addition of the TH-IC. The most efficient L0 configuration is the 256-byte L0-IC(L0_16x4), which reduces energy consumption to 75.71% of the baseline value. The greatest energy savings is achieved by the 256-byte TH-IC using a line-based transfer scheme (TL_16x4), which manages to reduce the overall energy consumption to 68.77% of the baseline energy. The best performing invalidation policy for the TH-IC is to use TL bits until we reach the 32-line cache configuration. Here, the target bit transfer scheme (TT_32x4) performs better than the line-based scheme (TL_32x4) due to the additional energy requirements of maintaining 32 TLs in each TH-IC line. We expect that larger TH-ICs will be more energy efficient with a TT scheme due to its linear scaling of cache metadata. We also see that the tagless hit line buffer (TH LB) manages to reduce overall energy consumption more than any of the tested L0-IC configurations. This is due in part to the faster application running times, but a significant portion of the savings comes from the reduced fetch structure size (essentially a single line with little metadata).

Figure 10 shows the average hit rate of the L0-IC and TH-IC configurations. The false miss rate is also shown for the non-line buffer TH-IC configurations. It is important to note that the sum of these two rates is the same for all policies in each cache organization of the TH-IC and L0-IC. The TH-IC really applies a different access strategy to the existing L0-IC. Thus it is not surprising that the TH-IC contents are the same as an equivalently sized L0-IC on a given instruction fetch. As the invalidation policy becomes more complex, the number of guaranteed hits increases for a TH-IC, while the number of false misses is correspondingly reduced. This is due to a reduction in the number of invalidations that were overly conservative. Overall, however, we see that the hit rate for a TH-IC is competitive with the hit rate of a comparably-sized L0-IC. Considering the relatively small false miss rate, we see that the sample policies perform quite well, particularly the target-bit and line-based transfer schemes. For instance, the TL_16x4 configuration has a hit rate of 84.99%, while the L0_16x4 has a hit rate of 87.63%. More complicated policies could unnecessarily increase energy utilization and overall circuit complexity. The TH LB has a hit rate of approximately 67.62%, which shows that the majority of instructions fetched are sequential and in the same line. False misses are not captured since they cannot be exploited in any way by a line buffer.

Figure 11 compares the average power required of the instruction fetch stage of the pipeline for the L0-IC and TH-IC configurations. The baseline case again is using only an L1-IC and has an average power of 100%. These results are not surprising considering the overall processor energy results already shown. However, this shows that the power requirements for the TH-IC are considerably lower than an equivalently sized L0-IC. For the best TH-IC configuration (TL_16x4), the fetch power is approximately 35.47% of the baseline fetch power, while the corresponding 256-byte L0-IC yields only a 43.81% fetch power. The average fetch power reduction comes from the elimination of cheaper tag/ID comparisons on cache hits, as well as fewer ITLB accesses. The TH LB is able to reduce the fetch power to 46.60%. This is a considerable savings for such a small piece of hardware, but it is obvious that the higher miss rate (and thus increased number of L1 fetches) reduces the energy efficiency below that achievable with a true TH-IC.

Energy-delay squared or ED^2 is a composite metric that attempts to combine performance and energy data together in a meaningful way. Energy is directly proportional to the square of the voltage ($E \propto V^2$), so decreasing the voltage reduces the energy of a processor, but in-

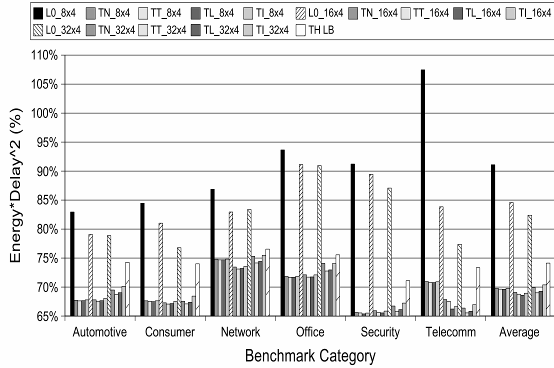


Figure 12. Energy-Delay^2 of L0 and Tagless Hit Instruction Caches

creases the clock period and hence execution time. Dynamic voltage scaling is one such technique that reduces the clock rate in an effort to save energy [18]. ED^2 is then used as an indicator of a design's relative performance and energy characteristics. Figure 12 shows ED^2 computed for each of the L0-IC and TH-IC configurations that we tested. Again, the TL_16x4 configuration performs best (68.58%), since it had the greatest energy reduction with no performance penalty. The TH LB also shows greater potential than any of the L0-IC configurations.

In addition to evaluating the efficacy of TH-IC with traditionally embedded applications such as those found in MiBench, we also performed similar experiments using the benchmark 176.gcc available in SPECInt2000. We selected this benchmark because it is representative of a fairly complex general-purpose application. While most of the embedded MiBench applications will spend their time fetching the same tight loops, the fetch patterns of 176.gcc should be more diverse due to its longer running time and varied processing phases. This benchmark is run to completion using the cccp.i test input file with the L0_16x4, TH LB, and TL_16x4 configurations, which correspond to the most efficient configurations found during the MiBench experiments. Similar results have been obtained using the reference input (expr.i).

Table 3 compares the average MiBench results with the experimental values we obtained from running 176.gcc. The execution time penalty for using an L0-IC with 176.gcc (4.1%) is lower than the MiBench average (6.5%). There is a clear difference in average fetch power between 176.gcc and MiBench. With MiBench, the fetch power is lower because the hit rate of the small cache is much higher. For 176.gcc, the guaranteed hit rate for the TH-IC is 73.57%, and adding false misses only brings the rate up to 77.86%, which is still much smaller than the MiBench results (84.96% → 87.63%). Overall, the TL_16x4 TH-IC configuration again outperforms both the TH LB and the traditional L0-IC in all aspects. Despite 176.gcc having completely different data access patterns than our embedded applications, we see that fetch behavior is still comparable, and thus TH-IC can be beneficial for processors running general-purpose applications as well.

4 Related Work

There has been some previous work to decide whether the filter cache or the L1-IC should be accessed on each cycle. A predictive filter cache has been developed to allow direct access to the L1-IC without accessing a filter cache first for accesses that are likely to be filter cache misses [20]. A dynamic prediction is made regarding whether or not the subsequent fetch address is in the filter cache. This prediction is accomplished by storing for each line the four least significant tag bits for the next line that is accessed after current line. When an instruction is being fetched, these four bits are compared to the corresponding bits for the current instruction's tag. These bits will often be identical when consecutive lines are accessed in a small loop. Significant energy savings were obtained with a slight performance degradation by accessing the L1-IC directly when these bits differ. Conventional tag comparisons and the additional 4-bit comparison are both required for the predictive filter cache. The HotSpot cache uses dynamic profiling to determine when blocks of instructions should be loaded into the filter cache based on the frequency of executed branches [22]. Instructions are fetched from the L0-IC as long as hot branches are encountered and there are no L0-IC misses. Like the predictive filter cache, the HotSpot cache also requires a tag comparison to be made for each L0 access. In contrast, the TH-IC requires no tag comparison for guaranteed hits and only a small ID comparison for potential misses. This should result in reduced energy consumption compared to both of these approaches.

There has also been previous research on guaranteeing hits in an L1-IC. The goal for these caches is to reduce energy consumption by avoiding unnecessary tag checks in the L1-IC. Way memoization was used to guarantee that the next predicted way within a 64-way L1-IC is in cache [16]. Valid bits are used to guarantee that the next sequential line accessed or line associated with the target of a direct transfer of control are in cache. Different invalidation policies were also investigated. When the next instruction is guaranteed to be in cache, 64 simultaneous tag comparisons are avoided. The history-based tag-comparison (HBTC) cache uses a related approach. It stores in the BTB an execution

Table 3. Comparing Fetch Efficiency Across Different Application Domains

	MiBench Average			176.gcc from SPECInt2000		
	L0_16x4	TH LB	TL_16x4	L0_16x4	TH LB	TL_16x4
Execution Cycles	106.50%	100.00%	100.00%	104.10%	100.00%	100.00%
Total Energy	75.17%	74.13%	68.58%	83.81%	80.41%	79.03%
Small Cache Hit Rate	87.63%	67.62%	84.96%	77.86%	66.61%	73.57%
Fetch Power	43.81%	46.60%	35.47%	63.72%	58.33%	56.07%
Energy-Delay Squared	84.57%	74.13%	68.58%	90.82%	80.41%	79.03%

footprint, which indicates if the next instruction to be fetched resides in the L1-IC [11, 10]. These footprints are invalidated whenever an L1-IC miss occurs. Rather than guaranteeing hits in a conventional L1-IC, we believe that the most beneficial level to guarantee hits is within a small instruction cache. First, most of the instructions fetched can be guaranteed to be resident in the small IC, which will result in a smaller percentage of guaranteed hits in an L1-IC. Since the L1-IC will be accessed fewer times in the presence of a small instruction cache, the additional metadata bits in the L1-IC will be costly due to the increased static versus dynamic energy expended. Second, the invalidation of metadata when a line is replaced is much cheaper in a small IC since there are fewer bits of metadata to invalidate.

There are other small structures that have been used to access a large percentage of the frequently executed instructions within an application. A zero overhead loop buffer (ZOLB) is a compiler managed IC, where the innermost loops of an application are explicitly loaded into the ZOLB and executed [6]. The advantages include reduced energy consumption and elimination of loop overhead (increments, compares, and branches). The disadvantages of a ZOLB are that the number of loop iterations must be known and there can be no other transfer of control instructions within the loop besides the loop branch. In contrast, loop caches are used to hold innermost loops when short offset backward branches are detected during execution [15]. Loop caches are exited whenever the loop branch is not taken or another transfer of control occurs. The L-cache is similar to the loop cache, but the compiler statically places innermost loops in the address space of the executable that will be resident in the L-cache [2, 3]. Thus, an L-cache can hold only a pre-specified and limited number of loops. Line buffers are essentially degenerate L0/filter caches that contain only a single line [19]. The cache access latency of a line buffer is typically prolonged such that a line buffer miss will trigger the corresponding fetch from the L1-IC during the same processor clock cycle. The TH-IC provides better performance than a conventional L0-IC or line buffer, while capturing more instruction accesses than a ZOLB, loop cache, or L-cache.

5 Conclusions

L0/filter instruction caches can greatly reduce the energy consumption of a processor by allowing the most frequent instructions to be accessed in a more efficient manner. However, L0-IC misses can accumulate and lead to significant application slowdown. In this paper, we proposed the Tagless Hit instruction cache, a small IC that does not require a tag/ID comparison or ITLB access to be performed on cache hits. The TH-IC is designed to take advantage of the properties of a small direct-mapped instruction cache and common instruction fetch behaviors. The reason that a TH-IC works is that it can identify nearly all of the accesses that will be hits in the cache before the instructions are fetched. The L0-IC impacts performance because trying to capture the few remaining accesses (false misses as hits) comes at the cost of a cycle of latency for all L0-IC misses. The TH-IC features a hit rate that is competitive with that of a similarly sized L0-IC (because there are not many false misses). Furthermore, the TH-IC can be bypassed during fetch when an instruction is not guaranteed to reside in one of the cache lines, eliminating the performance penalty normally associated with small instruction caches that can't be bypassed, like L0. Some TH-IC configurations require fewer metadata bits than a corresponding L0-IC, and even the larger TH-IC configurations require only a few extra bits per line. We also designed a compact line buffer based on the same principles as the TH-IC that provides better performance and energy efficiency than any L0-IC at a fraction of the area requirement. By exploiting fetch behavior, the TH-IC provides an attractive solution for improving processor energy efficiency without any negative performance impact or additional programmer effort. The TH-IC is even desirable for general-purpose processors as power and heat issues have become more of a concern. Even though execution characteristics of general-purpose applications tend to be more diverse, fetch behavior remains similar and can thus be exploited by the TH-IC. The lack of any additional miss penalty makes the TH-IC a viable alternative for high-performance processors, something which could not be said of the L0-IC.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants CCR-0312493, CCF-0444207, and CNS-0615085.

References

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

[2] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in a microprocessor design using a loop cache. In *Proceedings of the 1999 International Conference on Computer Design*, pages 378–383, October 1999.

[3] N. E. Bellas, I. N. Hajj, and C. D. Polychronopoulos. Using dynamic cache management techniques to reduce energy in general purpose processors. *IEEE Transactions on Very Large Scale Integrated Systems*, 8(6):693–708, 2000.

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[6] J. Eyre and J. Bier. DSP processors hit the mainstream. *IEEE Computer*, 31(8):51–59, August 1998.

[7] K. Ghose and M. Kamble. Energy efficient cache organizations for superscalar processors. In *Power Driven Microarchitecture Workshop, held in conjunction with ISCA 98*, June 1998.

[8] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 70–75, New York, NY, USA, 1999. ACM Press.

[9] 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. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[10] K. Inoue, V. Moshnyaga, and K. Murakami. Dynamic tag-check omission: A low power instruction cache architecture exploiting execution footprints. In *2nd International Workshop on Power-Aware Computing Systems*, pages 67–72. Springer-Verlag, February 2002.

[11] K. Inoue, V. G. Moshnyaga, and K. Murakami. A history-based I-cache for low-energy multimedia applications. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 148–153, New York, NY, USA, 2002. ACM Press.

[12] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, pages 219–230, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[14] J. Kin, M. Gupta, and W. H. Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Transactions on Computers*, 49(1):1–15, 2000.

[15] L. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 267–269, 1999.

[16] A. Ma, M. Zhang, and K. Asanović. Way memoization to reduce fetch energy in instruction caches. *ISCA Workshop on Complexity Effective Design*, July 2001.

[17] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160-mhz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Tech. J.*, 9(1):49–62, 1997.

[18] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, New York, NY, USA, 1998. ACM Press.

[19] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: A case study. In *Proceedings of the 1995 International Symposium on Low Power Design*, pages 63–68, New York, NY, USA, 1995. ACM Press.

[20] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proceedings of the International Conference on Computer Design*, pages 68–73, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

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

[22] C.-L. Yang and C.-H. Lee. HotSpot cache: Joint temporal and spatial locality exploitation for I-cache energy reduction. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 114–119, New York, NY, USA, 2004. ACM Press.