Guaranteeing Instruction Fetch Behavior with a Lookahead Instruction Fetch Engine (LIFE)

Stephen Hines
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
shines@nvidia.com

Yuval Peress Peter Gavin
David Whalley Gary Tyson
Computer Science Department
Florida State University
Tallahassee, FL 32306-4530
{peress,gavin,whalley,tyson}@cs.fsu.edu

Abstract

Instruction fetch behavior has been shown to be very regular and predictable, even for diverse application areas. In this work, we propose the Lookahead Instruction Fetch Engine (LIFE), which is designed to exploit the regularity present in instruction fetch. The nucleus of LIFE is the Tagless Hit Instruction Cache (TH-IC), a small cache that assists the instruction fetch pipeline stage as it efficiently captures information about both sequential and non-sequential transitions between instructions. TH-IC provides a considerable savings in fetch energy without incurring the performance penalty normally associated with small filter instruction caches. LIFE extends TH-IC by making use of advanced control flow metadata to further improve utilization of fetch-associated structures such as the branch predictor, branch target buffer, and return address stack. These structures are selectively disabled by LIFE when it can be determined that they are unnecessary for the following instruction to be fetched. Our results show that LIFE enables further reductions in total processor energy consumption with no impact on application execution times even for the most aggressive power-saving configuration. We also explore the use of LIFE metadata on guiding decisions further down the pipeline. Next sequential line prefetch for the data cache can be enhanced by only prefetching when the triggering instruction has been previously accessed in the TH-IC. This strategy reduces the number of useless prefetches and thus contributes to improving overall processor efficiency. LIFE enables designers to boost instruction fetch efficiency by reducing energy cost without negatively affecting performance.

Categories and Subject Descriptors C.1 [*Computer Systems Organization*]: Processor Architectures

General Terms Experimentation, Measurement, Performance

Keywords Lookahead Instruction Fetch Engine (LIFE), Tagless Hit Instruction Cache (TH-IC), L0/Filter Cache

1. Introduction

Processor design and development requires careful consideration of execution, power, and area characteristics. The requirements for embedded systems are often tighter due to production costs and reliance on batteries. One active area for exploration in embedded processor design is instruction fetch. Existing research has focused on reducing instruction cache power, which accounts for approximately 27% of the total processor power on the StrongARM SA110 [16]. There are, however, other components of instruction fetch that can also have a sizable impact on the processor power characteristics. In particular, speculation logic can account for a significant percentage of fetch power consumption even for the limited speculation performed in scalar embedded processors. In fact, with advances in low power instruction cache design, these other fetch components can dominate instruction fetch power requirements. This paper describes the Lookahead Instruction Fetch Engine (LIFE), which is a new approach for instruction fetch that attempts to reduce access to these power-critical speculation logic structures when it can be determined that such an access is unnecessary.

Central to LIFE is the presence of a Tagless Hit Instruction Cache or TH-IC, which we introduced in prior work [11]. TH-IC is an improvement upon the traditional notion of an L0 or filter cache (L0-IC) [13]. Both TH-IC and L0-IC are small caches that are placed before the L1 instruction cache (L1-IC) for the purpose of providing a more energy-efficient access path to frequently fetched instructions. If the L0-IC experiences a cache miss, then the instruction needs to be fetched from the appropriate line in the L1-IC on the following cycle. This entails an additional 1-cycle miss penalty before the appropriate instruction is fetched. Although the L0-IC reduces fetch energy requirements, these penalties can result in a significant performance degradation for many applications. TH-IC completely eliminates this miss penalty by providing a direct bypass to the L1-IC when the instruction to be fetched cannot be guaranteed to reside in the TH-IC. Guaranteeing instruction cache hits is accomplished through the use of specialized cache metadata bits that represent simple relations between instructions within the TH-IC. LIFE extends TH-IC by incorporating additional metadata which can be used to further improve decisions made during instruction fetch.

LIFE exploits knowledge available in the TH-IC about the next instruction to be fetched to selectively bypass speculation mechanisms present in the fetch pipeline stage. If the next sequential instruction can be *guaranteed* to not be a transfer of control instruction, then the branch predictor (BP), branch target buffer (BTB), and return address stack (RAS) do not need to be activated during its fetch. Thus LIFE improves utilization of fetch-associated structures by selectively disabling access. This results in reductions in both fetch power and energy consumption, while not increasing execution time. The reduction in power is affected by the hit rate

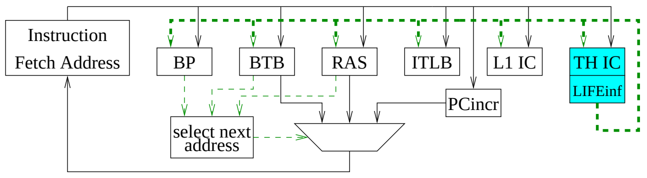


Figure 1. LIFE Organization

of the TH-IC and the ratio of branch and non-branch instructions in the TH-IC. LIFE can further capitalize on common instruction fetch behavior by detecting transfers of control that will be predicted as not-taken, which allows bypass of the BTB and RAS on subsequent fetches since they act as non-branch instruction flow. A substantial number of branches resolve as not-taken and are consistent for the entire execution. Identifying those branches residing in the TH-IC that are almost always not-taken enables the branch predictor to also be bypassed.

Figure 1 shows the organization of LIFE, which extends the capabilities of TH-IC with additional metadata for tracking the branching behavior of instructions. Bold dashed lines represent control of fetch components by LIFE. Depending on the available metadata, LIFE can choose to enable or disable access to appropriate fetch structures during the following instruction fetch cycle.

In addition to directing access of fetch structures, the behaviors detected by LIFE can also be used to guide decisions that are made in subsequent pipeline stages. Traditional next sequential line prefetching (NSLP) schemes often trade off energy efficiency for improved performance from the data cache [9, 19, 20]. Although the previously published TH-IC approach is not suitable for data caches due to the lack of identifiable access regularity, the fetch behavior of an application can yield clues about data access patterns. This behavioral information can then be used to improve NSLP.

This paper makes the following contributions:

- We have performed the first study of instruction residency within small instruction caches.
- An enhanced analysis of fetch behavior with a TH-IC motivates the need to improve the efficiency of other fetch components beyond caches as they become the dominant energy consumers.
- We propose the Lookahead Instruction Fetch Engine (LIFE), a microarchitectural enhancement that exploits fetch regularity to better control access to these power-hungry speculation resources. Using just 128 bits (2 bits per TH-IC instruction), LIFE can eliminate 61.17% of accesses to the speculative fetch structures, leading to significant reductions in fetch power and total energy consumed.
- We also show that the behaviors captured by LIFE can be effectively used to tune other pipeline optimizations through a case study with NSLP.

2. Background: Tagless Hit Instruction Cache (TH-IC)

Figure 2(a) shows the traditional layout of a small L0/filter cache. Since the L0-IC is accessed instead of the L1-IC, any miss in the L0-IC will incur an additional 1-cycle miss penalty prior to fetching the appropriate line from the L1-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 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 into a memory system design is essentially a tradeoff providing a savings in fetch energy at the expense of longer execution times.

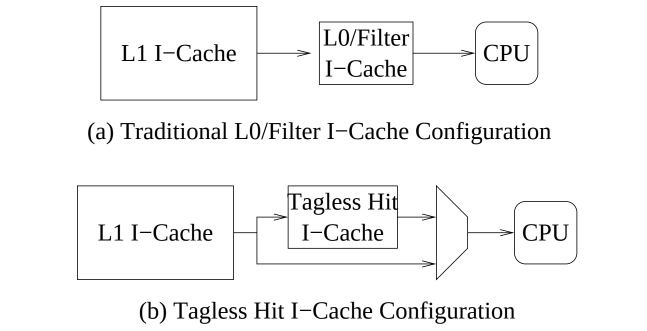

L1 I−Cache
L0/Filter I−Cache
CPU
(a) Traditional L0/Filter I−Cache Configuration
L1 I−Cache
Tagless Hit I−Cache
CPU
(b) Tagless Hit I−Cache Configuration

Figure 2. Traditional L0 and Tagless Hit I-Cache Layouts

The Tagless Hit Instruction Cache, or TH-IC, that our LIFE research in this paper extends is shown in Figure 2(b) and is an alternative configuration [11]. 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, tag comparisons become unnecessary on hits, hence the term “Tagless Hit”. The small size of the cache and its novel use of metadata are 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 an L0-IC due to the faster execution time (the TH-IC has no miss penalty), the reduction in Instruction Translation Lookaside Buffer (ITLB) accesses (the TH-IC can be accessed on guaranteed hits 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 tags are not used to verify a hit).

One of the key principles in the design of the TH-IC is the idea of bypassing the TH-IC when it is *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 a hit in TH-IC, 2) it resides in TH-IC, but it is not certain, so the L1-IC is directly accessed the, or 3) it did not reside in TH-IC, and the miss penalty was avoided by attempting to access it directly from the L1-IC. Only in the first case will the instruction from the TH-IC be read. When the instruction is not guaranteed to reside in the TH-IC, the appropriate tags in both the L1-IC and the TH-IC are checked. The term *false miss* is adopted in the second case to indicate that the instruction does actually reside in the TH-IC without being guaranteed to do so. In this case, no TH-IC line will be evicted.

Figure 3 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. Note that the amount of metadata in the TH-IC and a comparably sized L0-IC are similar since the size of the tags in the TH-IC can be decreased due to the elimination of the high portion of the tag that is redundant to the tag in the L1-IC being checked during the same cycle. TH-IC uses a single decision bit (*Fetch From TH-IC*) to determine from where to fetch the next

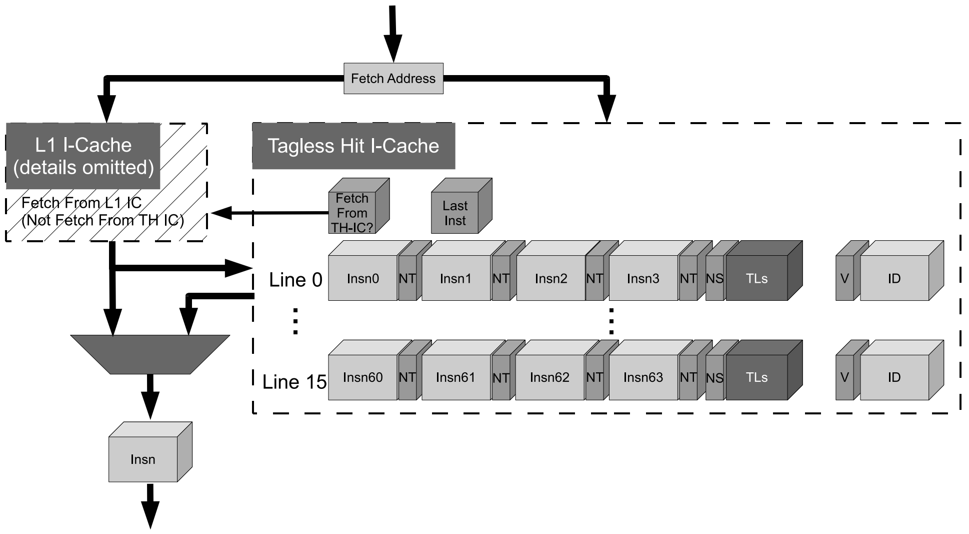


Figure 3. Tagless Hit Instruction Cache

instruction. The last instruction accessed from the TH-IC is also kept as a pointer in *Last Inst*.

There are two distinct types of access in the TH-IC or any other instruction cache for that matter: sequential accesses and transfers of control. On a sequential fetch access (branch predicted not-taken), there are two possible scenarios to consider. If the access is to any instruction other than the last one in a line, then this design will always choose to fetch the next instruction from the TH-IC, since it is guaranteed that the next sequential instruction in the same line will still be available on the subsequent access. If it is the last instruction in the line that is being fetched instead, 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 modulo line in the cache is actually 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.

If the predictor specifies a 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. 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. 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. In each TH-IC line, *TLs* is a bit vector that refers to the lines that may have NT transfers that target this line. When a line is replaced, the corresponding NT bits are invalidated.

3. Analyzing Instruction Residency within a Small IC

The success that TH-IC has demonstrated in reducing instruction cache power consumption has led us to examine how the same approach can be used to identify further opportunities in pipeline optimization. To do this, it is important to understand how instructions are flowing through the TH-IC. In order to get a better understanding of the abilities of TH-IC, we took a closer look at the eviction behavior of the individual lines. The first study was designed to identify how eviction was handled in a 16x4 TH-IC, which was the most energy efficient configuration found in a previous study [11]. The term 16x4 refers to 16 lines, where each line contains 4 instruction words. We collected statistics for each line evicted, counting how many of the other 15 lines have been replaced since the last time the current cache line was replaced. This information can shed some light on how the TH-IC is utilized. Figure 4 shows the results using an average of the MiBench benchmarks [10] described later in this paper. Conflict misses occur when few lines are displaced between consecutive evictions of a single line. This is seen (and expected), but does not constitute the common case. Complete cache replacement shows the highest individual frequency (22.48%), indicating a fairly large number of capacity misses. This result is not particularly surprising for such a small cache, but total cache line replacement of a direct mapped cache suggests a large loop structure equal to or exceeding twice the size of the TH-IC or frequent changes in the working set. In either case, we would expect that the miss behavior should be bimodal with some consecutive TH-IC line misses as well as long periods of TH-IC hits. This is a nice feature since it means that long sequences of instruction fetch should exhibit the same behavior.

Figure 5 shows both individual and cumulative results for consecutive hits in a 16x4 TH-IC. The graph shows values between 1 and 255+, with all consecutive streaks over 255 being collected under the 255+ data point. This data reinforces the observed eviction behavior. We see two distinct spikes in the figure, at 3 instructions (28.28% ind. and 35.56% cum.) and at 255+ instructions (26.91% ind.). The spike at 3 instructions shows standard line buffer behavior – a miss on the first instruction in a line followed by 3 consecutive hits for the remainder of the line. This is exactly what would occur with a long sequence of sequential fetches. This accounts for the majority of TH-IC misses resulting in a very clustered sequence of misses. The other spike (at 255+) indicates that when a longer sequence of hits occurs, it tends to be much longer. Thus, the TH-IC is very efficient for capturing simple loop behavior.

Figure 6 similarly shows individual and cumulative results for consecutive non-misses (guaranteed hits and false misses) with a 16x4 TH-IC. False misses can be caused by complex control

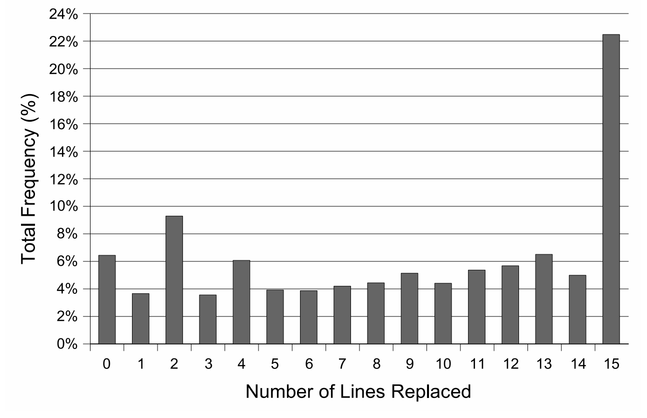


Figure 4. Line Replacements

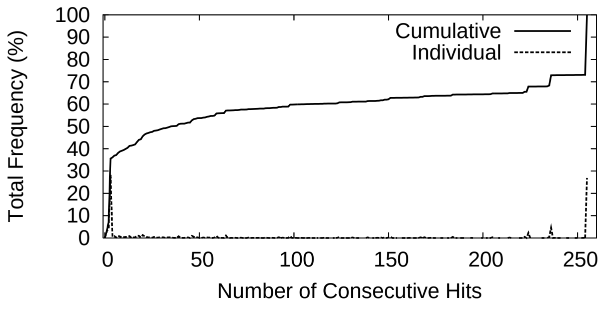


Figure 5. Consecutive Hits

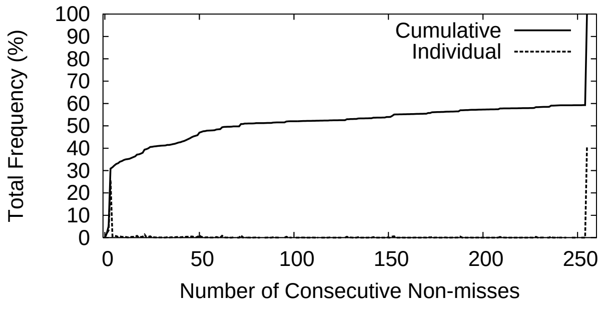


Figure 6. Consecutive Non-misses

flow in small loops. When we plot non-misses, we see an even greater number of very long instruction sequences residing in the TH-IC (40.71% 255+ ind.). This shows the TH-IC also does a respectable job in capturing more complex loop behavior. These results indicate that filtering instruction references through TH-IC provides an efficient means of partitioning the instruction stream, identifying sequences that have very well-structured behavior.

We also experimented with incorporating set associativity into TH-IC and LIFE. However, as other studies have shown, increased associativity does not always improve hit rate. During the course of these experiments, we found that although we could virtually eliminate the penalties of set associativity, the set associative TH-IC performed worse than the direct mapped version. Although some conflict misses may be avoided, the overall problem stems from overwriting useful lines twice as frequently as normal. These very small caches often encounter loops that are just slightly larger than the cache, leading to worst case behavior for both LRU and FIFO replacement policies. Even though other studies of very small caches have used direct mapped purely as a simpler mechanism, our analysis shows that direct mapped also provides better hit rates. It appears that set associativity is not useful for very small instruction caches.

4. Eliminating Unnecessary BTB, BP, and RAS Accesses

Although L1-IC energy tends to dominate the total energy required for the instruction fetch pipeline stage, a TH-IC reduces this impact. We will show a TH-IC actually reduces the cache power requirements so that it is less than the power consumed by the rest of instruction fetch. LIFE is focused on making fetch more energy-conscious, and thus it becomes increasingly important to reduce energy consumption in the remaining speculation components (BP, BTB, RAS) present in instruction fetch. Based on our analysis of TH-IC residency in the previous section, it is apparent that other microarchitectural state involved in fetch can be managed more efficiently. Preliminary studies with LIFE revealed that 86.75% of the executed instructions in our benchmarks are non-branch instructions, which means that access to the BTB, BP, or RAS is unnecessary *at least* 86.75% of the time. Figures 5 and 6 make it clear that additional information can be kept regarding branch/non-branch status of instructions, thus making a fetch engine more efficient.

LIFE employs both a conservative strategy and a more aggressive strategy for handling access to speculative components of instruction fetch. The conservative strategy disables speculation whenever the next fetched instruction can be guaranteed to not be a transfer of control. Of the branches that are executed in an instruction stream, we found that 23.73% are predicted strongly not-taken (state 00) by our bimodal branch predictor. The more aggressive strategy will further disable speculation when the next fetched instruction is a transfer of control, but has been previously predicted as strongly not-taken (00). Combined together, the BTB, BP, and RAS structures need not be accessed for 89.89% of fetched instructions.

LIFE depends on TH-IC to both supply and manage fetched instruction metadata. Figure 7(a) shows the baseline TH-IC metadata configuration used in this paper. Each line is composed of four instructions and their associated NT bits. A single NS and valid bit are associated with each line and 16 TL bits are used to facilitate line-based invalidation of NTs. ID is 6 bits long to uniquely identify the corresponding L1-IC line (replacing the longer tag of conventional caches), and only needs to be checked on a potential miss.

In Figure 7(b), a single *Next Sequential Non-Branch* bit (NSNB) has been added to each instruction in the line. On sequential transitions both within and across lines, this bit will be set when the next fetched instruction is not a transfer of control instruction. Whenever this bit is set and we fetch sequentially, the BP, BTB, and RAS need not be activated on the following cycle.

We can also extend the utilization of the NSNB bit to accept transfer of control instructions that are strongly not-taken (approximately 23.73% of branches). This usage will be referred to as NS00 due to 00 being the strongly not-taken bimodal BP state. In this configuration, whenever a branch is encountered and a prediction is made that it is strongly not-taken (state 00), the previous (sequential) instruction can set its NSNB bit. When this instruction is later fetched, the NSNB bit will indicate that no prediction should be made. While most branches that reach the strongly not-taken state remain not-taken, some of the branches would suffer if their prediction remained not-taken while the instruction is in the TH-

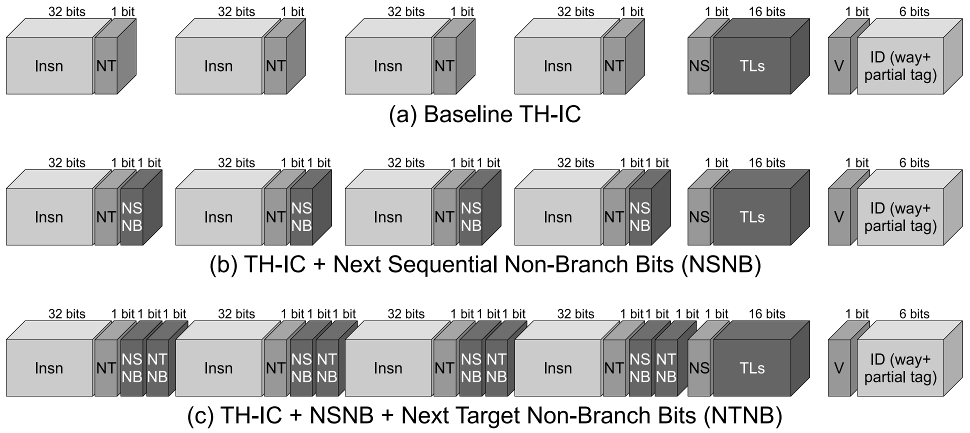

32 bits
1 bit
Insn
NT
NS
TLs
16 bits
V
ID (way+ partial tag)
6 bits
(a) Baseline TH-IC
NS NB
(b) TH-IC + Next Sequential Non-Branch Bits (NSNB)
NT NB
(c) TH-IC + NSNB + Next Target Non-Branch Bits (NTNB)

Figure 7. LIFE Metadata Configurations

IC. Since additional speculative misses would cause an increase in cycle count, selective update of NS00 is necessary. On a branch misprediction, the TH-IC must be accessed to unset the NS00 bit of the previous sequential instruction since we can no longer guarantee that the branch will be predicted as not-taken.

Figure 7(c) is a further enhancement for LIFE that adds a single *Next Target Non-Branch* bit (NTNB) for each instruction in the line. This bit serves a similar role as NSNB, but it is set for branch instructions whenever the corresponding target instruction is not a transfer of control or is strongly not-taken. Most branches do not target unconditional jumps since compiler optimizations such as branch chaining can replace such a chain with a single branch. Conditional branch instructions are also rarely targets, since they are typically preceded by comparison instructions. Calls are rarely targets since they are usually preceded by additional instructions to produce arguments. Finally, returns are rarely targets since registers are often restored before the return. Thus, NTNB bits are generally quickly set for each direct transfer of control. Again, this speculatively reduces the need to access the BP, BTB, and RAS structures.

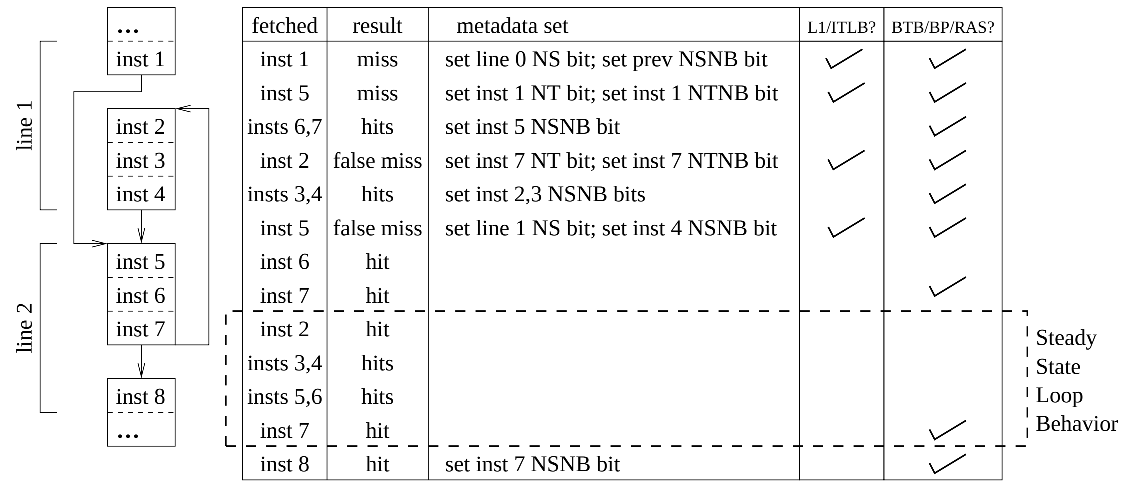
Adding these metadata bits to the TH-IC requires only a minor change in the steps to take for line invalidation. When a line is evicted, all of its NSNB and NTNB bits must be cleared. One interesting difference with the invalidation of NSNB versus the NS is that the previous line's last NSNB bit need not be cleared. This is due to the fact that any subsequent fetch after crossing that line boundary will still not need a BP/BTB/RAS access, as that instruction will not change branch status whether it was fetched from L1-IC or TH-IC. This same principle holds for NTNB bits when NT bits are cleared due to target line evictions. Thus, accesses to the BP/BTB/RAS structures can sometimes be avoided even when the L1-IC has to be accessed due to an instruction not being guaranteed to reside in the TH-IC.

Figure 8 shows an example of using LIFE to fetch a loop. This example includes both the NSNB and NTNB extensions. We track the number of BTB/BP/RAS accesses required in addition to the L1-IC and ITLB accesses. The baseline loop sets the appropriate NS and NT bits as the instructions are initially fetched from the L1-IC. Each of the instructions in the main loop (2–7) can be guaranteed to hit in TH-IC once the NS and NT links have been set, thus leading to extremely efficient cache behavior during the steady state operation. NSNB and NTNB bits are set as LIFE gathers information about the instructions fetched in the loop. During subsequent loop executions, the BTB/BP/RAS need only be accessed when instruction 7 is fetched, thus leading to only one single speculation access per loop iteration. Without NTNB bits, the fetch of instruction 2 would also require a BTB/BP/RAS access, since the branch transition from instruction 7 could not be guaranteed to target a non-branch.

Modifying the use of speculative components requires an understanding of the regularity in which these components are used and their results relative to the TH-IC. Figure 9 shows the taxonomy of branches as they are fetched using LIFE with NSNB, NS00, and NTNB extensions. This figure clearly demonstrates the correlation of speculative hardware with the fetch of instructions. We find that 56.48% (45.92% taken + 10.56% not-taken) of all branches will have their predicted next instruction as a guaranteed hit in the TH-IC. Further, 58.61% of all branches will have metadata available at fetch[1]. Combined, we find that almost all branches with metadata result in a guaranteed hit in the TH-IC. This fact could potentially lead to the discovery of new relevant metadata that could further influence speculative execution. A somewhat intuitive step was to enhance LIFE with speculative information. Unfortunately, as can be seen in Figure 9, 41.39% (11.94% from L1 + 29.45% first access) of branches either are not fetched from the TH-IC or do not contain metadata. This means that the smaller structure of an embedded BTB and BP would be applicable to only 58.61% of branches. The level of accuracy obtained did not offset the cost of additional metadata in LIFE and updates to the original hardware structures. While this result does mean that we cannot eliminate accesses to the 512-entry bimodal BP and BTB, it does not mean that new metadata cannot be used to influence the use of these structures to allow for a reduced speculative miss rate.

In addition to dealing with strongly not-taken branches, one might also consider having LIFE handle strongly taken conditional branches (encoded as 11 in the bimodal BP), since they will frequently occur due to loops. For instance, the BTB/BP/RAS structures are accessed each time instruction 7 in Figure 8 is fetched. LIFE could enable just the BTB and RAS, while always automatically predicting such branches as taken. To recognize the 11 state, an additional metadata bit (or a special encoding of the existing bits: NSNB, NTNB, and NT) would have to be introduced to the TH-IC, thus increasing the average energy for processing any instruction. This approach proved unfruitful since branches which are strongly taken and remain strongly taken account for very few of the overall instructions executed. Thus, any benefit is outweighed by the overhead of the additional metadata handling.

[1] For a branch to be considered as having metadata it must not be the first fetch of the given branch. If a TH-IC line eviction occurs and the branch leaves the TH-IC, then the next fetch of that branch will be considered its first fetch since the metadata was cleared.



fetched	result	metadata set	L1/ITLB?	BTB/BP/RAS?
inst 1	miss	set line 0 NS bit; set prev NSNB bit	✓	✓
inst 5	miss	set inst 1 NT bit; set inst 1 NTNB bit	✓	✓
insts 6,7	hits	set inst 5 NSNB bit		✓
inst 2	false miss	set inst 7 NT bit; set inst 7 NTNB bit	✓	✓
insts 3,4	hits	set inst 2,3 NSNB bits		✓
inst 5	false miss	set line 1 NS bit; set inst 4 NSNB bit	✓	✓
inst 6	hit			
inst 7	hit			✓
inst 2	hit			
insts 3,4	hits			
insts 5,6	hits			
inst 7	hit			✓
inst 8	hit	set inst 7 NSNB bit		✓

Figure 8. Reducing BTB/BP/RAS Accesses Example

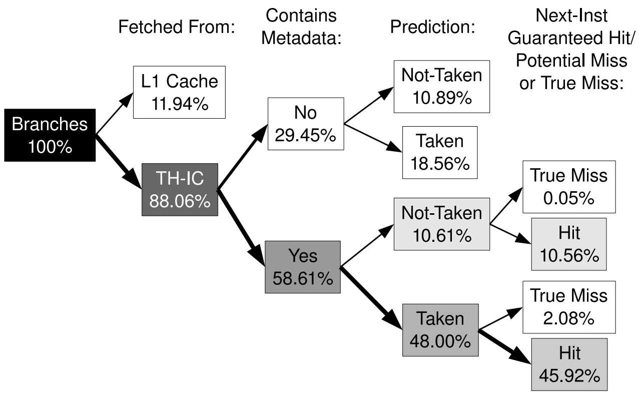


Figure 9. Branch Taxonomy in TH-IC.

Table 1. Baseline Configuration

I-Fetch Queue	4 entries
Branch Predictor	Bimodal – 512
Branch Target Buffer	512 entries
Branch Penalty	3 cycles
Return Address Stack	8 entries
Fetch/Decode/Issue/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
L0 Instruction Cache (when configured)	256B 16 lines, 16B line, direct mapped, 1 cycle hit
Tagless Hit I-Cache (when configured)	256B 16 lines, 16B line, direct mapped, 1 cycle hit Line-based invalidation

5. Experimental Results with LIFE

In order to evaluate LIFE, we obtained a copy of the previous research framework for TH-IC [11]. This framework uses the SimpleScalar simulator [2] with Wattch extensions [6] to estimate energy consumption. The *cc3* clock gating style approximates the effect of leakage. Under this scheme, inactive portions of the processor consume 10% of their active energy. We also use the MIPS/PISA instruction set, although the baseline processor is configured with parameters that are similar to the StrongARM. Table 1 shows the exact configuration parameters that were used in each of the experiments. The L0-IC and TH-IC are only configured when specified in the evaluation. We evaluate using just TH-IC and then extend it with the LIFE techniques for eliminating BP, BTB, and RAS accesses. In subsequent graphs, NSNB indicates using the NSNB bits only to handle instructions that are not transfers of control, while NS00 corresponds to a configuration that uses the NSNB bits to handle strongly not-taken branches as well. Finally, NTNB includes NSNB, NS00, and adds NTNB bits to handle targets that are not transfers of control.

While the Wattch power model is only an approximation, it is sufficient for providing reasonably accurate estimates for simple cache structures using CACTI [21]. The structures being evaluated in this work (TH-IC, BP, BTB, RAS, L0-IC, L1-IC, ITLB) are composed primarily of simple regular cache blocks and associated tags/metadata. Although L0-IC and TH-IC may differ in functionality (tag checks vs. metadata updates), they should 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.

Table 2 shows the subset of MiBench benchmarks that we used for each of our experiments [10]. 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 [5], 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 ex-

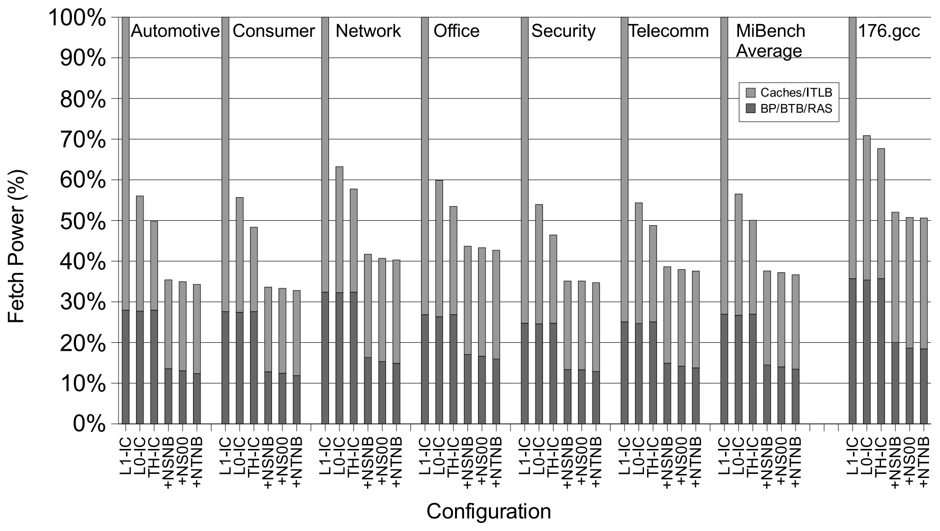


Figure 10. Impact of LIFE on Fetch Power

Table 2. MiBench Benchmarks

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

periments have also been done with TH-IC, and results were very similar to the smaller input results. We also present results for the *176.gcc* benchmark available in SPECInt2000 in order to evaluate the impact of LIFE on a more complex general purpose application. This benchmark is run to completion using its test input file (cccp.i). We have obtained similar results using the reference input (expr.i). MiBench results are presented by category along with an average due to space constraints. All results for *176.gcc* are displayed after the MiBench average. Results are verified for each benchmark and sanity checks are performed to ensure correct behavior and verify that LIFE does not unfairly use information that should not be available to it.

Figure 10 shows the fetch power distribution for each of the various L0-IC and LIFE configurations. The fetch power bars are split into I-cache and speculation components, and results are normalized to the overall L1-IC values. The MiBench average results show that while the L0-IC and TH-IC alone can reduce cache power, they have a negligible impact on speculation power. The baseline speculation power is 27.97% of the total fetch power, while caches account for 72.03%. Adding an L0-IC cuts the cache power to 28.30% and the speculation power to 27.73%. The reduction in speculation power is explained by power estimates which are averaged across all execution cycles. The extra cycles due to L0-IC misses do not require access to the speculative resources, thus reducing the average but keeping the same total. Total energy (discussed later) will show an increase in energy consumed during execution. TH-IC reduces cache power to just 21.82% of the original fetch power with no change in speculation power since branch prediction is unmodified and total execution time is unchanged. Once TH-IC is employed, speculation power exceeds the average I-cache access power, thus providing the motivation for our exploration. LIFE, using just NSNB bits for non-transfers of control, results in cache power of 21.87% and a reduction in speculation power to 13.55%. The cache power is slightly increased from the baseline LIFE with TH-IC due to additional NSNB bits and associated upkeep. Allowing strongly not-taken branches (NS00) results in a 13.06% speculation power and 21.91% cache power. Finally, adding the NTNB bits reduces the speculation power to 12.34% and the cache power to 21.95%, for an overall fetch power savings of 65.70% over the baseline L1-IC. LIFE eliminates an average of 61.17% of speculative hardware accesses due to being able to exploit the NSNB/NTNB metadata. Results for *176.gcc* show slightly reduced but still significant savings in fetch power for LIFE.

While an L0-IC degrades performance by about 6.44% (with a 17.67% maximum increase on *rijndael*), TH-IC and LIFE have no associated performance penalty. The baseline TH-IC as well as LIFE with any of the configured NSNB, NS00, or NTNB extensions do not exhibit any associated performance difference from the baseline configuration.

Figure 11 shows the benefit of LIFE on total processor energy. Adding an L0-IC reduces total energy to 79.00%. This is less than the power savings because total energy accounts for the increase in execution time caused by the increased L1-IC latency when servicing L0-IC misses. TH-IC cuts total energy to 72.23%, in line with power analysis since execution time is unaffected. Similarly, LIFE with NSNB reduces total energy to 65.44%, in line with power estimates. Adding NS00 reduces the total energy to 65.27% and NTNB yields a final total processor energy of 64.98%. With *176.gcc*, applying all the LIFE extensions reduces the total energy to 72.88%. In all cases, LIFE extensions are able to achieve equivalent power reduction in speculation components that TH-IC achieved with the I-cache/ITLB. The combination of these enhancements yields a much greater energy savings than any other existing instruction fetch energy reduction technique evaluated.

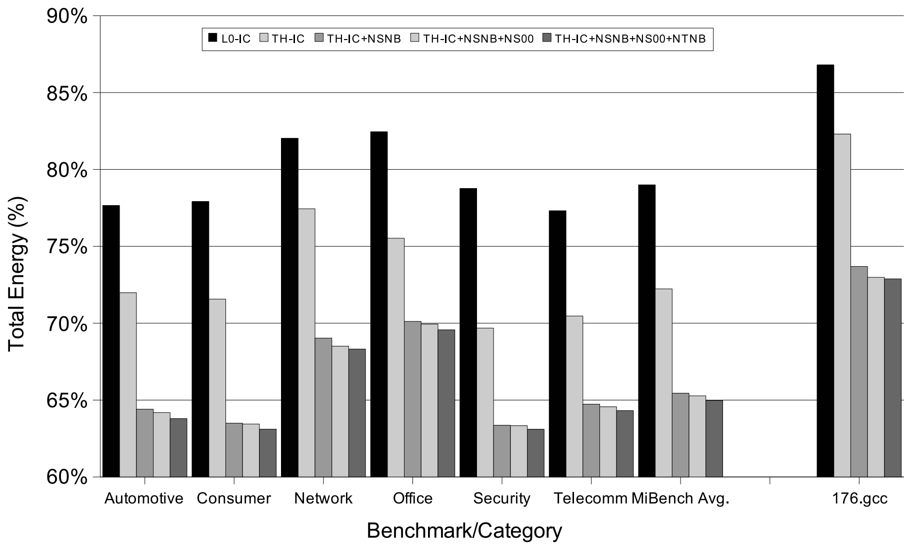


Figure 11. Impact of LIFE on Processor Energy

6. Improving Next Sequential Line Prefetch with LIFE

LIFE is designed to exploit program fetch behavior in an effort to better utilize pipeline resources. In examining the functionality of the TH-IC, we found that presence of instructions in the TH-IC can be used as a simple execution phase predictor. When instructions are fetched from the TH-IC, they are identified as being part of very regular instruction fetch. This regularity of fetch behavior can identify potential optimizations in later pipeline stages either to improve execution performance or to reduce power consumption. There have been numerous proposed microarchitectural enhancements which seek to identify regular resource usage patterns and scale the pipeline resources to best match the expected requirements. LIFE can perform as an efficient early predictor to trigger these optimizations. Cyclone is a dynamic scheduler that attempts to reduce the scheduling complexity by implementing a form of selective instruction replay [7]. LIFE can provide additional information about program behavior to such a scheduler while improving overall processor energy efficiency. Throttling of processor resources is also becoming increasingly important, as portions of the processor are over-designed and cannot be fully exploited throughout a program's entire execution [1, 3, 15]. LIFE provides a simple mechanism for subsequently fetched instructions to take advantage of improved resource assignment or scheduling based on previous execution characteristics.

One area of the pipeline that would seem to benefit the least from knowledge of instruction fetch behavior is the data cache. While previous TH-IC results could not extend benefits to the data cache, LIFE can use the fetch metadata to intelligently select when to apply aggressive optimization techniques for the data cache. Such optimizations may not be able to be normally applied for highly constrained processor designs due to their associated tradeoffs. Prefetching is an example of such an aggressive optimization that attempts to reduce cache miss penalties by using idle cache cycles to fetch data speculatively from predicted addresses. When a triggering condition is met, the cache will prefetch an additional line of data. Although prefetching can improve performance through reducing cache misses, it significantly increases memory traffic and may force the early eviction of other cache lines that are still useful, thus polluting the cache. With next sequential line prefetching (NSLP), the first non-prefetch access of a line triggers the next sequential line to be fetched from memory once the current cache access operation has finished [9, 19, 20]. By associating a first access bit with each cache line, a subsequent true use of a prefetched line triggers the next sequential line to be prefetched. This technique potentially allows the prefetching to stay one line ahead of the current accesses. If a line is already available in the cache, then the data is not fetched from memory unnecessarily.

Although LIFE only keeps track of instruction fetch behavior, we can exploit this information to have NSLP selectively enabled or disabled. NSLP works best on sequentially accessed arrays that exhibit a good deal of spatial locality. Small loops are often used to iterate through arrays of data, and LIFE can provide a simple mechanism for detecting such loops. We can use a single bit to denote whether the *initial* line access was a hit or a miss, which we will refer to as the TH-IC *line-hit bit*. It is possible for a TH-IC line-hit access to not be in a small loop, but the majority of cases will show that this fetch behavior corresponds to small loop execution. Thus we can enable NSLP when we have a TH-IC line-hit and disable it on any miss. The TH-IC line-hit information will be propagated through the pipeline stages with the fetched instruction and will only be checked if the instruction is a memory access.

Throttling the NSLP mechanism will help to reduce the frequency of useless prefetches. This approach conserves energy and can also help to reduce cache pollution for benchmarks that typically do not exhibit good prefetch behaviors. Table 3 presents a simplified view of the combinations of instruction and data flow behavior in a pipelined processor. Both instruction and data flow can be simply characterized as stable or unstable. Stable instruction flow corresponds to loop behavior where the instruction sequences are able to be fetched from a small cache. Stable data flow indicates a data access pattern that exhibits good spatial locality, possibly improving with prefetching. LIFE and TH-IC provide their greatest benefits during periods of stable instruction flow as they rely on past instruction behavior to make guarantees. NSLP only provides benefits during periods of stable data flow. When data flow is unstable, NSLP performs useless prefetches that waste energy and can evict useful cache lines. The majority of execution time in most applications will consist of stable instruction flow.

Table 3. Correlating Instruction and Data Flow Behavior

Inst. Flow	Data Flow	Comments
Stable	Stable	Typical loop (strided array data accesses)
Stable	Unstable	Typical loop (pointer-chasing data accesses)
Unstable	Stable	Does not occur often in practice
Unstable	Unstable	Transitional code (non-loop)

Table 4. Impact of LIFE with Next Sequential Line Prefetch

	Useless Prefetches		Execution Time	
	NSLP	LIFE+NSLP	NSLP	LIFE+NSLP
Automotive	34.52%	31.85%	93.61%	94.39%
Consumer	25.15%	16.27%	97.00%	97.60%
Network	38.02%	25.23%	95.95%	96.14%
Office	49.43%	29.01%	98.05%	98.53%
Security	52.26%	36.99%	99.96%	99.96%
Telecomm	22.36%	27.17%	99.86%	99.88%
MiBench Avg.	36.82%	28.52%	97.54%	97.88%
176.gcc	51.31%	38.17%	97.61%	98.79%

Table 4 summarizes the results of our experiments with LIFE and next sequential line prefetch. Useless prefetches denotes the fraction of prefetches that are never accessed by the actual application before being evicted. NSLP reduces the overall application execution times, while LIFE+NSLP can attain most of the benefits by just selectively prefetching. For MiBench, the number of useless prefetches is reduced by 22.54% when applying NSLP selectively based on LIFE metadata. With *176.gcc*, the use of LIFE reduces the useless prefetches by an additional 25.6%. This translates into a significant reduction in memory traffic due to data access. Wattch does not model the off-chip energy consumption of main memory, so energy results are not included. Simplified energy analysis based on memory access counts shows that LIFE+NSLP results in an overall greater system energy savings than NSLP alone.

7. Related Work

There are other small structures that have been used to access a large percentage of the frequently executed instructions within an application. Zero overhead loop buffers (ZOLBs) [8], loop caches [14], and L-caches [4] all target regular fetch patterns that include small tight loop behavior. These techniques can obtain similar benefits in reducing the number of speculative accesses of the BTB/BP/RAS by knowing which instructions can potentially branch to other code.

Reinman et al. propose a serial prefetch architecture that splits tag and data accesses into separate pipeline stages for instruction fetch [18]. By performing tag accesses first, only the relevant data line from the proper cache way needs to be fetched, saving processor energy. By fetching ahead with the tags, instruction lines can be prefetched into a special buffer from which they can later be promoted to the actual instruction cache on a direct fetch request, thus improving performance. In contrast, LIFE completely eliminates tag accesses for guaranteed hits, and also reduces the number of accesses to other fetch structures relating to speculation, which serial prefetching cannot do.

There has been some previous work on reducing the energy consumption of the BTB. Utilizing banked BP organization along with a prediction probe detection to identify cache lines containing no conditional branches was evaluated by Parikh et al. [17]. This study showed that careful layout of BP resources and a course granularity bypass capability can make much larger BPs feasible from a total energy perspective. Just as there are fewer bits of metadata to manage in a TH-IC versus an L1-IC to guarantee hits, it is also more efficient to avoid accesses to the BP/BTB/RAS by tracking this information in a small TH-IC versus a larger structure for the entire BTB. A leakage power reduction technique has also been used to turn off entries in the BP and BTB if they are not accessed for a specified number of cycles [12]. The counters used will limit the amount of energy savings. Yang et al. reduce BTB energy consumption by altering the instruction set and producing setup code to store the distance to the next branch instruction from the beginning of each basic block in a small branch identification unit [22]. The BTB is not accessed until the next branch is encountered. While reducing energy consumption, this approach requires alteration of the executable to store the branch distance information and a counter to detect when the next branch will be encountered.

8. Future Work

There are a number of different ways that LIFE can be tuned for other fetch stage configurations. The size of the BP and BTB can be varied. It is likely that LIFE will support a larger BP and BTB to reduce the number of branch mispredictions while still retaining energy efficiency since the number of accesses to these larger components are significantly reduced. It would also be interesting to use a correlating branch predictor with LIFE. Since the current design of LIFE, when using the NS00 configuration, involves an update to the metadata in the case of a misprediction, other or even new metadata may be updated simultaneously without the overhead of the TH-IC access.

In this paper we have limited our changes to the fetch stage and data cache portions of the pipeline. This makes sense for embedded processors since they tend to have simple pipeline organization after fetch. Since both TH-IC and LIFE have no impact on processor performance, we predict that their use in high performance, general purpose processors could yield positive results. Total energy reduction will not be as great as for embedded architectures since a greater portion of the power budget is used by later (more complex) pipeline stages, but since any reduction comes without performance penalty, inclusion of LIFE is still warranted. For these complex pipelines, LIFE can be used to improve resource scheduling decisions. We already explored a case involving next sequential line prefetch, showing that LIFE can act as a filter, identifying small loops with repetitive, consistent execution behavior. The ability to identify repeated instruction sequences very early in the pipeline can also affect the design of dynamic instruction scheduling, data forwarding, instruction wakeup logic and other power intensive portions of pipeline execution. New designs for these structures can exploit the regularity of instruction execution for LIFE-accessed instructions to reduce power consumption or even to enhance performance since improved scheduling decisions can be cached in the later pipeline stages for these instructions. Basically, LIFE can be used to identify those phases of program execution that exhibit very structured, consistent execution patterns. These phases can then be exploited in various other stages of the pipeline architecture to throttle back resources to reduce power consumption or to iteratively capture best resource allocation to improve performance (or both). We believe that the ability to partition the instruction execution into (at least) two very different execution behaviors opens up new research opportunities in many areas of processor design.

9. Conclusions

As power has become a first class design constraint, architects continue to exploit regularity in application behavior to improve processor designs. In this paper we have demonstrated how instruction fetch power utilization can be improved by exploiting those portions of the program execution that exhibit well-defined, repetitive behavior. The goal of the LIFE extensions to the processor fetch logic is to identify repetitive code sequences and cache enough information about their prior behavior to control access to the BP, BTB, and RAS structures. LIFE eliminates access to these structures an average of 61.17% of the time, resulting in a total savings of 63.33% on instruction fetch power and an additional savings of 9.42% on total processor energy beyond what is achievable with TH-IC alone. All of these improvements are made with no increase in execution time and require only 128 additional metadata bits (2 bits per TH-IC instruction). Furthermore, we showed that this enhanced knowledge of instruction fetch behavior can guide other pipeline decisions, particularly for handling aggressive optimizations like next sequential line prefetch. By focusing on the loops detected by LIFE, we were able to reduce the number of useless data prefetches while still maintaining the majority of the performance benefit afforded by prefetching. Other throttling techniques may benefit similarly from the phase behaviors detected by LIFE. The design complexity of TH-IC and LIFE is minimal, with an area cost comparable to similar conventional filter caches using a few simple index structures to maintain the metadata required to control resource allocation. LIFE can easily be incorporated into any processor pipeline, requiring only minor additions to the fetch stage and the routing of branch outcomes to update the LIFE metadata. We believe that the techniques presented in this paper can become a common design choice for both low-power embedded processors as well as high-performance general purpose processors. This will become even more important as multicore architects focus their efforts towards further reducing per core power consumption.

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] Aragón, J. L., González, J., and González, A. Power-aware control speculation through selective throttling. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 103–112.

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

[3] Baniasadi, A., and Moshovos, A. Instruction flow-based front-end throttling for power-aware high-performance processors. In *Proceedings of the 2001 international symposium on Low power electronics and design* (New York, NY, USA, 2001), ACM Press, pp. 16–21.

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

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

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

[7] Ernst, D., Hamel, A., and Austin, T. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th annual International Symposium on Computer Architecture* (New York, NY, USA, 2003), ACM, pp. 253–263.

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

[9] Gindele, J. Buffer block prefetching method. *IBM Tech Disclosure Bulletin 20*, 2 (July 1977), 696–697.

[10] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization* (December 2001).

[11] Hines, S., Whalley, D., and Tyson, G. Guaranteeing hits to improve the efficiency of a small instruction cache. In *Proceedings of the 40th annual ACM/IEEE International Symposium on Microarchitecture* (December 2007), IEEE Computer Society, pp. 433–444.

[12] Hu, Z., Juang, P., Skadron, K., Clark, D., and Martonosi, M. Applying decay strategies to branch predictors for leakage energy savings. In *Proceedings of the International Conference on Computer Design* (September 2002), pp. 442–445.

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

[14] Lee, L., Moyer, B., and Arends, J. 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* (1999), pp. 267–269.

[15] Manne, S., Klauser, A., and Grunwald, D. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th annual International Symposium on Computer Architecture* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 132–141.

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

[17] Parikh, D., Skadron, K., Zhang, Y., Barcella, M., and Stan, M. Power issues related to branch prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture* (February 2002), pp. 233–244.

[18] Reinman, G., Calder, B., and Austin, T. M. High performance and energy efficient serial prefetch architecture. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing* (London, UK, 2002), Springer-Verlag, pp. 146–159.

[19] Smith, A. J. Cache memories. *ACM Comput. Surv. 14*, 3 (1982), 473–530.

[20] Srinivasan, V., Davidson, E. S., and Tyson, G. S. A prefetch taxonomy. *IEEE Trans. Comput. 53*, 2 (2004), 126–140.

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

[22] Yang, C., and Orailoglu, A. Power efficient branch prediction through early identification of branch addresses. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (October 2006), pp. 169–178.