A

Reducing Instruction Fetch Energy in Multi-Issue Processors

Peter Gavin, David Whalley, Magnus Själander, Florida State University

The need to minimize power while maximizing performance has led to recent developments of powerful superscalar designs targeted at embedded and portable use. Instruction fetch is responsible for a significant fraction of microprocessor power and energy, and is therefore an attractive target for architectural power optimization. We present novel techniques that take advantage of guarantees so that the I-TLB, BTB, and BPB can be frequently disabled, reducing their energy usage, while simultaneously reducing branch predictor contention. These techniques require no changes to the instruction set and can easily be integrated into most single- and multiple-issue processors.

Categories and Subject Descriptors: []

ACM Reference Format:
ACM Trans. Architec. Code Optim. V, N, Article A (January YYYY), 24 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In recent years, the need to reduce the power and energy requirements in computer microprocessors has dramatically increased. In the past, microprocessor architects succeeded in improving the performance of their designs by increasing clock frequency, and building wider and deeper pipelines. These design choices inevitably lead to increased power and energy usage, and thus increased heat dissipation. However, new developments in portable devices, such as tablets and smartphones, have led to overlapping demands for low power and high performance. Additionally, large scale corporate server farms have significant performance and power requirements, and the corresponding cooling demands they entail. While much attention has been given to reducing power in embedded systems through architectural techniques, less attention has been given to reducing power in high-performance architectures. Thus, it is important to reexamine more performant processor architectures to see if more efficient designs are possible. A significant portion of pipeline power can be attributed to instruction fetch, due to the number of large structures involved that are commonly accessed on every clock cycle. Thus instruction fetch is an attractive target for power optimization.

The contributions of this paper include: (1) techniques for reducing instruction cache power in multiple instruction fetch pipelines, (2) a technique enabling significant reductions in *instruction translation lookaside buffer* (I-TLB) power, and (3) a technique enabling significant reductions in *branch prediction buffer* (BPB) and *branch target buffer* (BTB) power and slightly improved prediction rates. Novel aspects of these techniques include (1) exploitation of the faster access time of a very small instruction cache to avoid BPB/BTB accesses and improve branch predictor performance, (2) the use of guarantees that the targets transfers of control reside in the same page, thus obviating the need for I-TLB accesses, and (3) the use of guarantees of that BTB entries are present to avoid BTB tag array accesses. All of these techniques reduce power without adversely effecting performance.

The remainder of this paper is organized in the following manner. We first provide background on the *Tagless-Hit Instruction Cache* (TH-IC) [Hines et al. 2007; Hines et al. 2009], which we used as a baseline for our study. We then discuss the environment for our experiments. We next present how the TH-IC can be extended to support multiple instruction fetch. Afterwards, we describe our proposed techniques to reduce I-TLB, BPB, and BTB accesses. We then show the experimental results from simulations of three processor models to which these techniques have been applied. We will

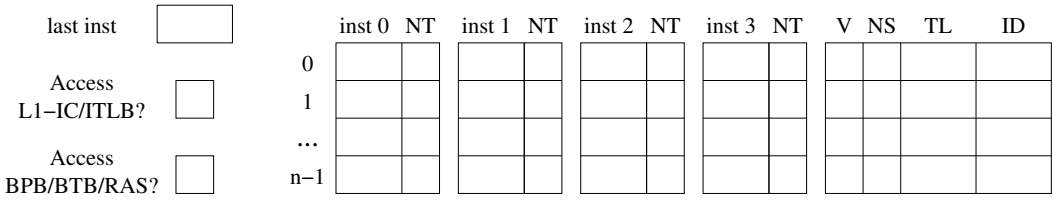

Fig. 1. TH-IC Organization

show that over 40% of fetch energy can be saved for low leakage technologies. Following that, we contrast our research with several related fetch power reduction techniques. Finally, we discuss future work and conclusions for this study.

2. BACKGROUND: THE TAGLESS-HIT INSTRUCTION CACHE (TH-IC)

The TH-IC [Hines et al. 2007] was introduced as an effective alternative to an L0/filter cache for single-issue pipelines, with the goal of reducing instruction fetch energy. The primary motivator in targeting instruction fetch for energy reduction is that this stage of the pipeline requires accessing a number of large structures in order to maintain high throughput. A *level zero instruction cache* (L0-IC) reduces fetch energy when compared to using just a *level one instruction cache* (L1-IC), as an L1-IC is much larger, often containing a factor of at least 64 times as many instructions [Kin et al. 2000]. However, an L0-IC has much higher miss rates and each miss incurs a 1-cycle miss penalty as the L0-IC is checked before accessing the L1-IC. These miss penalties cause both a loss in performance and reduce some of the energy benefit from accessing the lower power L0-IC due to increased application execution time. The TH-IC holds instructions and provides them to the core in nearly the same fashion as an L0-IC, but has no associated miss penalty when the instruction is not present. Instead, it makes guarantees about when the next instruction to be fetched will be resident in the TH-IC by using a small amount of additional metadata. When such a guarantee cannot be made, the TH-IC is bypassed and the L1-IC is instead directly accessed. No tag check is performed when an instruction is guaranteed to reside within the TH-IC. In addition, I-TLB accesses are unnecessary on guaranteed hits as the page number of the virtual address is not used to index into the TH-IC. The energy savings for a TH-IC actually exceeds the savings from an L0-IC due to no execution time penalty on TH-IC misses and no tag checks or I-TLB accesses on TH-IC hits.

Figure 1 shows the organization of a TH-IC with a configuration of four instructions per cache line. There is a *next sequential* (NS) bit associated with each line that indicates that the next sequential line in memory is guaranteed to be the next sequential line in the TH-IC. Associated with each instruction is a *next target* (NT) bit indicating that the instruction is a direct ToC and that its target instruction resides within the TH-IC. These bits enable the processor to determine, while accessing a given instruction, whether the next instruction to be fetched will be a guaranteed TH-IC hit. Whenever an instruction is guaranteed to be present in the TH-IC, the bit labeled *"Access L1-IC/I-TLB?"* is cleared, and is set otherwise.

For instance, if the next instruction to be fetched sequentially follows the current instruction, and the current instruction is not the last in its line, then the next fetch is guaranteed to be present within the TH-IC, since the line will not be evicted before the next fetch is performed. Otherwise, if the current instruction is the last in its line, we must check the line's NS bit. If the NS bit is set, then the next line is a guaranteed hit, and will be accessed from the TH-IC. If the NS bit is not set, then the next line is not a guaranteed hit, and it will be accessed directly from the L1-IC. At the end of

the cycle in which the successive instruction is fetched, the L1-IC line containing it is written into the TH-IC, and the previous line's NS bit is set. As long as neither line in the pair is evicted, later traversals across the pair will result in guaranteed hits to the second line. If either line in the pair is evicted, the first line's NS bit will be cleared.

The NT bit plays a similar role in guaranteeing hits. However, it is used to determine whether the target of a direct transfer of control (ToC) is a guaranteed hit. For example, suppose the current instruction being fetched is a direct transfer of control, and the branch predictor determines that it should be taken. If the NT bit associated with the instruction is set, then the target is guaranteed to be present within the TH-IC. Otherwise, the target will be fetched from the L1-IC. At the end of the cycle in which the target is fetched, it will be written into the TH-IC, and the NT bit associated with the ToC will be set. If neither the ToC's line nor the target's line are evicted prior to the next time the ToC is fetched, then the NT bit will still be set. If the branch predictor again indicates that the ToC should be taken, then the target will be accessed from the TH-IC.

In addition to the bits used to provide guarantees, the TH-IC tracks state to aid in invalidating these bits when they become obsolete. Each line has an associated TL field that indicates which lines should have its NT bits cleared when the corresponding line is evicted. Since the TH-IC is accessed in the same cycle as the L1-IC on potential misses to avoid invalidating metadata for false misses, the TH-IC can eliminate the high portion of the tag that is redundant to the L1-IC. These smaller tags are referred to as IDs. Despite containing additional types of metadata, the TH-IC metadata size per line is about the same size as the per line metadata in an L0-IC due to the significantly decreased tag size.

3. ENVIRONMENT FOR EXPERIMENTATION

Before going into detail regarding microarchitectural innovations, we describe the processor configuration, benchmarks, simulation environment, and target instruction set used in this study. This information is referenced later in the paper when describing the design. As the currently predominant architecture in embedded systems is the ARM architecture, we have based our model on the ARM Cortex series of cores. The series includes a wide variety of cores, from simple, single-issue, in-order architectures with short pipelines, to complex, superscalar, out-of-order cores with pipelines of up to 20 stages. These cores are intended for low power usage scenarios, and can be implemented using synthesized logic and complied RAMs, and without use of custom blocks. The new techniques presented in this paper can similarly be implemented without custom blocks, and using synthesized logic.

Our experimental configurations are shown in Table I. For single issue experiments, we used a similar configuration to the one described in [Hines et al. 2007] to verify that the TH-IC results can be reproduced and to facilitate comparisons. We also used two more aggressive processor models: a two-way fetch, superscalar in-order model, and a four-way fetch, superscalar out-of-order model. As mentioned, all the models have been parameterized to closely represent the features found in the ARM Cortex series used in smartphones and tablets. We have attempted to use configurations similar to those described in the ARM online documentation [ARM Holdings 2013].

The simulation environment used includes the SimpleScalar simulator [Burger and Austin 1997; Austin et al. 2002] with Wattch extensions [Brooks et al. 2000]. Wattch assumes clock gating, and approximates the leakage in inactive processor components as using a constant percentage of their dynamic energy. We have modified Wattch to allow this leakage to be configured, and run simulations assuming 10%, 25%, and 40% leakage. The Wattch power model using CACTI [Wilton and Jouppi 1996] has been accepted for providing reasonably accurate estimates for simple structures. The bench-

Table I. Experimental Configuration

Similar to	ARM Cortex-A5	ARM Cortex-A8	ARM Cortex-A15
Issue Style	In-order		Out-of-order
Fetch/Decode/Issue Width	1	2	4
Fetch Queue Size	1	3	7
Memory Latency	100 cycles		
Page Size	4 kB		
DTLB, I-TLB	10-entry, fully assoc.	32-entry, fully assoc.	32-entry, fully assoc.
L1-IC	4 kB, 2-way assoc	16 kB, 4-way assoc	32 kB, 2-way assoc
L1-DC	4 kB, 4-way assoc, 16 B line	16 kB, 4-way assoc, 64 B line	32 kB, 2-way assoc, 64 B line
L2-UC	128K, 8-way assoc	256K, 8-way assoc	512K, 16-way assoc
BPB	Bimodal, 256 entries	2-level, 512 entries	Comb., 1024 entries
BTB	256 entries	512 entries	1024 entries
Branch Penalty	2 cycles	4 cycles	8 cycles
RAS	8 entries		16 entries
Integer ALUs	1	2	4
Integer MUL/DIV	1	2	2
Memory Ports	1	2	3
FP ALUs	1	2	2
FP MUL/DIV	1	1	1
LSQ Size	2	4	8
RUU Size	2	4	8

mark suite used in this study is the MiBench suite [Guthaus et al. 2001]. We compiled all benchmarks using the GCC compiler included with the SimpleScalar toolset.

The structures we evaluate in this study include the TH-IC, L1-IC, I-TLB, BPB, and BTB. The TH-IC has a similar structure to that of a small, direct-mapped L0-IC, and its metadata can be stored in registers since the size of each metadata type is very small.

Similarly to the TH-IC study, we also used the MIPS/PISA instruction set. The formats used for direct unconditional ToCs are shown in Figures 2 and 3. Each MIPS instruction is four bytes in size and aligned on a four byte boundary. Thus, the branch offset represents a distance in instructions and the target address in the jump format is an instruction address. In this paper we describe the *program counter* (PC) as referring to instruction addresses rather than byte addresses as the two least significant bits in a byte address are always zero.

We also modified the baseline instruction fetch architecture simulated by SimpleScalar in order to more realistically model a multiple issue processor. Our modified version of SimpleScalar has a configurable fetch width, allowing up to one, two, or four sequentially consecutive instructions to be fetched per cycle. We call the instructions fetched in a given cycle a *fetch group*. Each fetch group is always aligned to a boundary of its size. For instance, when simulating a four-wide fetch, each fetch group is aligned on a 16-byte boundary. If a transfer of control targets an instruction that is not at the start of a group, then the instructions in the group that reside prior to the target are not accessed. A beneficial side effect of this alignment is that exactly one L1-IC line is accessed per fetch cycle, which is not the case in the original SimpleScalar model. Once instructions are fetched, they are placed into a $(2 \times fetch\_width - 1)$-instruction deep queue. We assume that the number of instructions that can be issued from this queue per cycle is the same as the fetch width. We do not fetch instructions unless

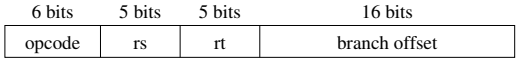

Fig. 2. Conditional Direct Branch Format

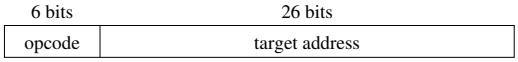

Fig. 3. Unconditional Direct Jump Format

the queue has $fetch\_width$ free entries. In this way, the fetch architecture ensures that $fetch\_width$ instructions are available for issue each cycle, and yet does not fetch farther ahead than is immediately useful.

4. DESIGNING A MULTIPLE FETCH TH-IC

The original design of the TH-IC is limited in use to single issue microarchitectures. Such simple microarchitectures are useful in applications with demands for low power, but are less applicable when higher performance is demanded. Since multiple-issue processors are now becoming commonplace in low-power applications, we have designed a multiple fetch TH-IC that can be used in these more complex microarchitectures.

The layout and operation of the TH-IC is similar under multiple fetch pipelines as under single fetch. The output port on the TH-IC is widened to allow $fetch\_width$ adjacent instructions to be accessed each cycle. In addition, rather than tracking the position of the instruction fetched in the previous cycle, the position of the last issued instruction in the last fetch *group* is tracked. For instance, if on the next cycle the processor will fetch the group that sequentially follows the current fetch group, the position of the final instruction in the current group is stored. If the processor will fetch the target of a taken direct transfer of control, then the position of the transfer of control is stored, and *not* the position of the final instruction in its group. This position tracking behavior for taken direct transfers enables the correct NT bit to be set once the target is fetched.

A consequence of increasing the fetch width is that the number of opportunities for sequential intra-line fetches are less frequent. Suppose, for instance, that a basic block of code is being executed for the first time, and that it is several TH-IC lines in length. If the TH-IC line size is four instructions, a single issue processor will be able to take advantage of guaranteed hits for three out of four instructions in each line. However, when the fetch width is widened to four instructions, no hits can be guaranteed, as each sequential fetch will consume the entire line, and successive fetches will be from new lines. Thus it is quite beneficial to increase the instruction cache line size when using the TH-IC on multiple fetch processors.

5. FURTHER REDUCING FETCH ENERGY

While the instruction cache may be the dominating contributor to fetch energy dissipation, there are other structures that consume significant additional power. These include the I-TLB and the structures involved in branch prediction.

We have devised a new organization, as show in Figure 4, that stores metadata that is used to avoid accesses to the I-TLB and the branch related structures for multi-fetch architectures. Figure 5 illustrates the entire fetch engine once these changes have been put into place. The primary changes are changes to the TH-IC itself, and the introduction of a small amount of logic to detect small direct jumps, whose purpose and function will be explained later. Some additional control logic must be routed from the TH-IC to other structures, allowing the TH-IC to control when these structures are accessed. In a standard pipeline, the BTB, BPB, return address stack (RAS), L1-IC, and I-TLB are accessed together in parallel, during a single pipeline stage. The TH-IC is also accessed in parallel with these structures, during the same pipeline stage. Data that is read from the L1-IC must be fed back to the TH-IC so it can be written at the

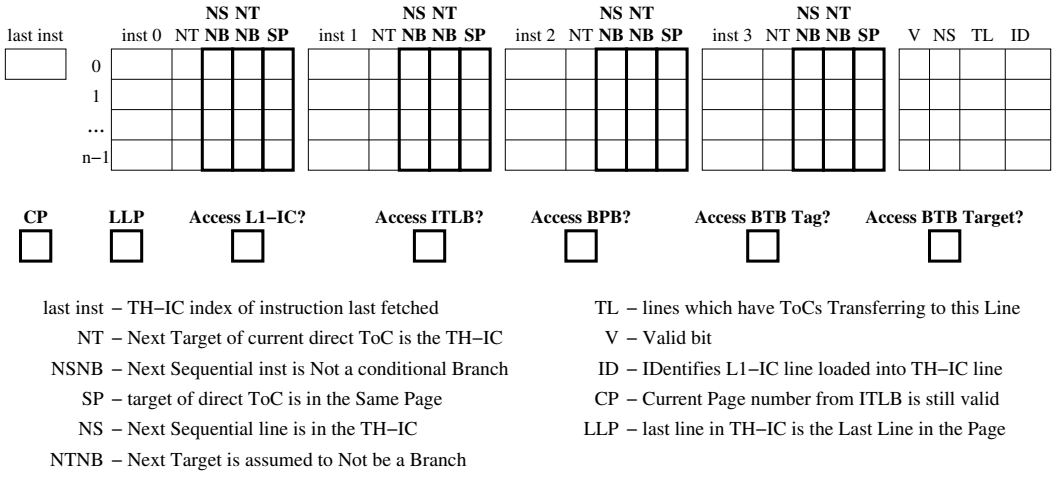

Fig. 4. New Metadata Organization

end of the cycle. As the TH-IC may be accessed for a read and a write in the same cycle, it must be implemented using dual-ported memories or with standard registers.

While the TH-IC already disables the I-TLB much of the time, it provides opportunities to disable it even when the fetch will not occur from the TH-IC. This is facilitated by adding the CP, LLP, and SP bits as shown in Figure 4. There is one CP (*current page*) bit, which indicates current physical page is the same as the last page accessed from the I-TLB. This bit is necessary because, if the I-TLB is disabled, there may be occasions where the physical page number is not known, but fetching can proceed anyway. There is one LLP (*last line in page*) bit, which indicates that the last line in the TH-IC is also the last line in the page. This bit is used to detect when a sequential fetch leaves the current page. Finally, an SP (*same page*) bit is associated with each instruction, and indicates that the instruction is a direct transfer of control whose target is in the same page.

For disabling the branch prediction hardware, no new metadata is needed beyond that described in [Hines et al. 2009]. However, some of this metadata has been rearranged or is used in a different fashion. These changes are described in more detail in the following subsections.

5.1. Disabling I-TLB Accesses on TH-IC Misses

Disabling I-TLB accesses needs to be addressed within the context of the TH-IC, which disables the I-TLB on guaranteed hits. Consider Figure 6, which depicts a code segment that straddles a page boundary. There are three different ways that the application can switch between these two pages. First, the unconditional jump (instruction 3) in block 1 can transfer control to block 3 (instruction 7). Second, the conditional branch (instruction 9) in block 3 can transfer control to block 2 (instruction 4). Finally, there can be a sequential fetch within block 2 (instruction 5 to instruction 6). There is an NT bit associated with each instruction indicating if the associated instruction is a direct ToC and that its target instruction is resident within the TH-IC. There is an NS bit associated with each line of instructions indicating if the next sequential line is resident within the TH-IC. The figure shows the status of these bits after each of the three blocks has initially been executed. The I-TLB will not be accessed for the transitions 3→7, 9→4, and 5→6 when the NT bits associated with the first two transitions

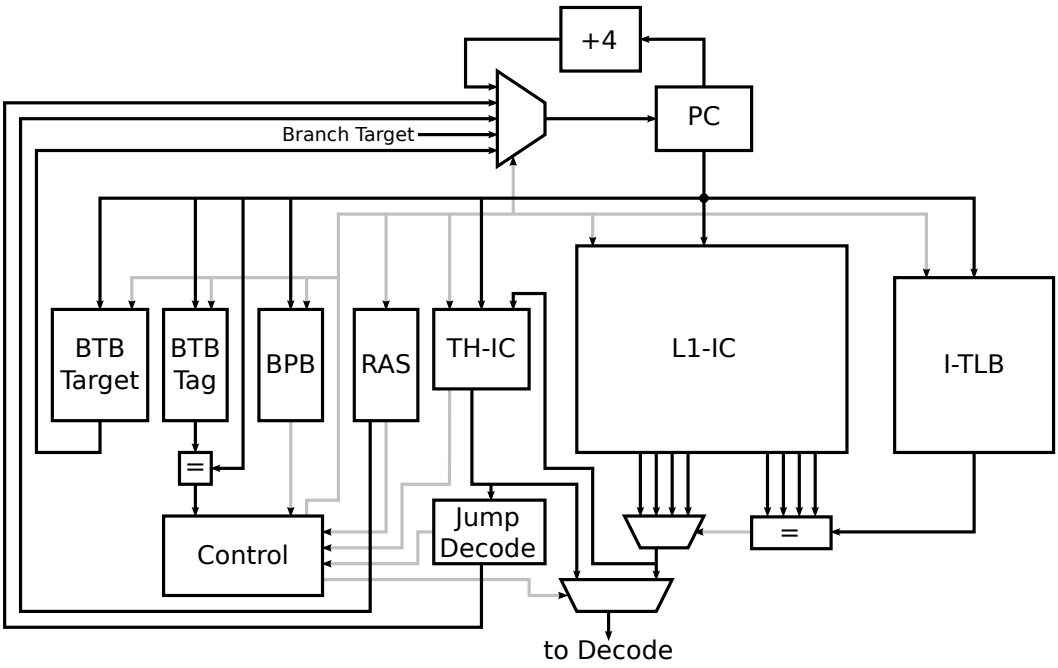

Fig. 5. New Fetch Engine Block Diagram. Grey lines indicate control signals

and the NS bit associated with the third transition are set. Note that when the next instruction is guaranteed to be in the TH-IC, the physical page number is not needed as there will be no tag check for guaranteed hits. Thus, as long as the loop continues to iterate, there will be no I-TLB accesses even though the page boundary is repeatedly being crossed.

We attempt to disable the I-TLB on TH-IC misses if the next instruction is guaranteed to remain in the same page. However, the processor needs to detect when a page boundary has been crossed on a TH-IC hit so that the next TH-IC miss will always access the I-TLB to obtain the current physical page number (PPN). We add an internal register to store the PPN immediately after it has been read from the I-TLB, and a *current page* bit (CP, shown in Figure 4) that indicates whether or not the stored PPN corresponds to the page of the instruction to be fetched. We additionally keep a single bit of state, called *"Access I-TLB?"*, that determines if the I-TLB should be enabled. We use this information to avoid I-TLB accesses when accessing the L1-IC. If the instruction resides on the same page, the most recently retrieved PPN is reused.

The CP bit is set each time the I-TLB is accessed as the PPN register is updated. Whenever a transition between instructions is made that crosses a page boundary and the next instruction is a guaranteed hit, then the CP bit will be cleared as the I-TLB is not accessed on TH-IC hits. The first instruction accessed that is a TH-IC miss when the CP bit is clear will result in an I-TLB access and the CP bit will be reset, as the PPN register is updated with the current page.

We store additional metadata to make guarantees when the next access is a TH-IC miss, but will reside in the same page so we can avoid additional I-TLB accesses. The analysis of this problem can be divided into distinct cases: (1) sequential accesses, (2) direct ToCs, (3) returns, and (4) other indirect ToCs. We attempt to disable I-TLB accesses in the first three cases. The following subsections describe when we can guar-

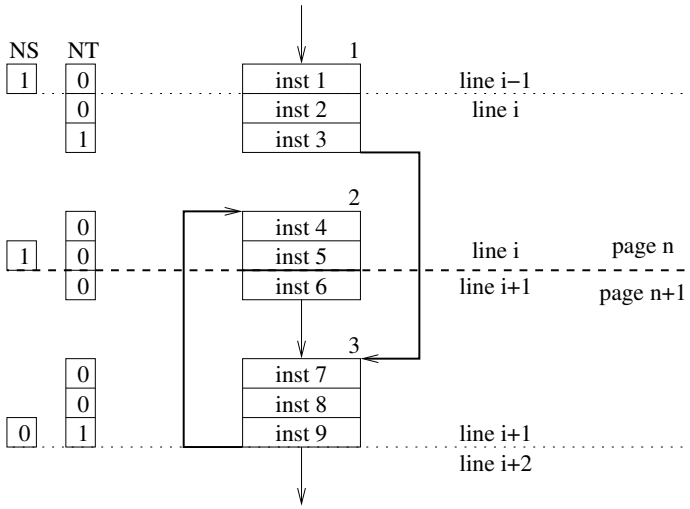

Fig. 6. Example Loop Straddling a Page Boundary

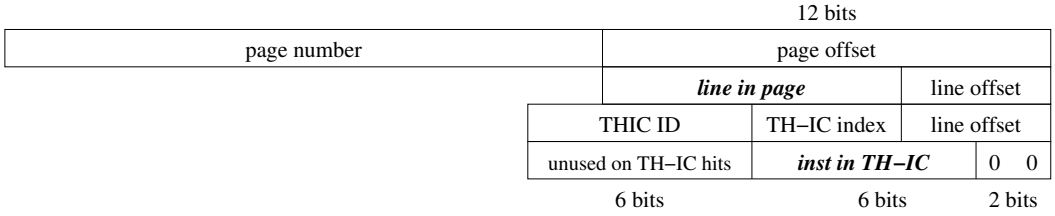

Fig. 7. Address Fields to Check

antee that the next instruction to be fetched for TH-IC misses will remain on the same page. I-TLB accesses can only be disabled in these cases when the CP bit is set.

5.1.1. Disabling the I-TLB for Sequential Accesses. If the current instruction is not the last instruction in the page, then the next sequential instruction accessed is guaranteed to reside on the same page. We perform this check in two steps. First, when the last line in the direct mapped TH-IC is replaced, a check is made to determine if the new line is the last line in the page, which is indicated using the *last line in page* bit (LLP, shown in Figure 4). Note that if the last line of the TH-IC is not being replaced, then the LLP bit is unchanged. Second, when there is a sequential fetch, the current instruction is checked to see if it is the last instruction in the TH-IC, which is accomplished by detecting if the portion of the virtual address containing the TH-IC index and the two most significant bits of the line offset identifying the instruction within the line are all 1's. This check examines six bits for a TH-IC configured with sixteen 16-byte lines, as there are 64 instructions within the TH-IC. This second check only needs to be performed when the LLP bit is set. Figure 7 depicts the *line in page* and *inst in TH-IC* bits that must be accessed for these two checks. We have found that in 94.7% of all sequential fetches, the last line in the TH-IC is not also the last line in the page. This large percentage is due in part to the page size containing 16 times the number of instructions as the TH-IC in our configuration.

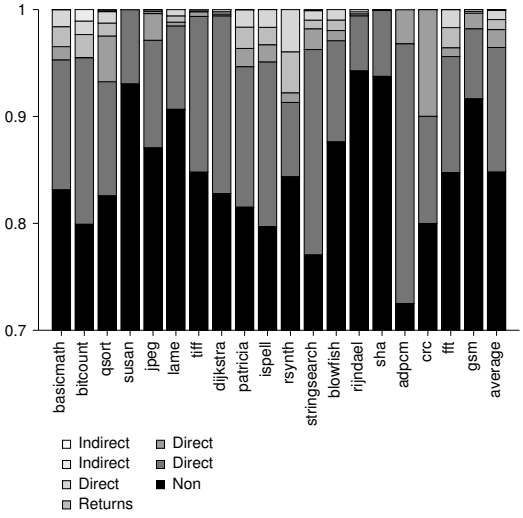

Fig. 8. Distribution of Dynamic Instruction Types

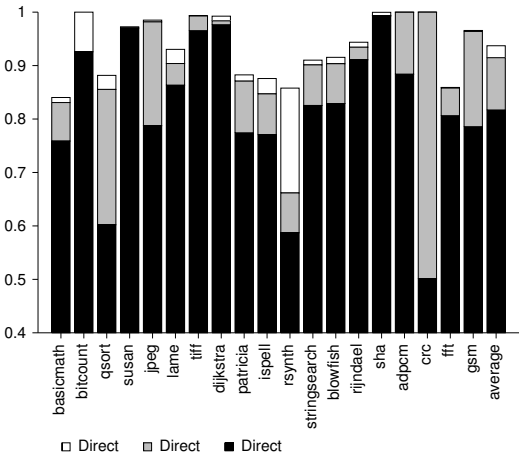

Fig. 9. Direct ToCs with Same Page Targets

5.1.2. Disabling the I-TLB for Targets of Direct ToCs. If the target of a ToC is within the same page as the ToC instruction itself, then the I-TLB need not be accessed, as the current PPN register value can instead be used. Figure 8 shows how often each type of ToC occurred in the MiBench benchmark suite. Direct ToCs for the MIPS ISA include conditional branches, direct unconditional jumps, and direct calls, which comprised 14.2% of the total instructions and 93.3% of the ToCs. Figure 9 shows the ratios of how often these direct ToCs have targets to the same page, which occurs 93.7% of the time on average. We believe this frequency is high enough to warrant metadata to avoid I-TLB accesses when fetching these direct targets. Thus, we include a *same page* (SP) bit for each TH-IC instruction, depicted in Figure 4, that indicates when the instruction is a direct ToC and its target is on the same page. We check if the virtual page number returned by the I-TLB on the cycle when the target instruction is fetched is the same as the virtual page number used in the previous cycle for the ToC. If so, then the SP bit associated with the direct ToC instruction is set and subsequent fetches of the target fetch group when the ToC is taken need not access the I-TLB.

The processor does not always have to wait until the following cycle to set the SP bit when the direct ToC instruction is fetched from the TH-IC as a guaranteed hit. Instead, we can often exploit the faster TH-IC access time to determine within the same cycle if the target address is within the same page. Our experimental configuration used a four-kilobyte page size, which is a common size for many processors. A four-kilobyte page contains 1024 MIPS instructions, where each instruction is four bytes in size and aligned on a four byte boundary. Thus, only the ten most significant bits of the 12-bit page offset can have nonzero values for instruction addresses. If the instruction is a direct unconditional jump or call, then the SP bit is set and the I-TLB is disabled for the next instruction when the 16 most significant target address bits shown in Figure 3 are the same as the corresponding bits of the *program counter* (PC), as the ten least significant bits represent the instruction offset within the page.

5.1.3. Disabling the I-TLB for Targets of Returns. A *return address stack* (RAS) is used in our configuration to avoid mispredictions associated with return instructions. The processor can be made to compare the virtual page number of the target address in the top RAS entry with the virtual page number in the PC. This is possible because the top

RAS entry is available very early in the cycle, and so the comparison can be performed without creating additional timing pressure. If they match and a return instruction is fetched on a TH-IC hit, then the I-TLB access on the next cycle is disabled.

5.2. Disabling BPB/BTB Accesses

On single-issue pipelines without delayed transfers of control, and on multiple-issue pipelines, branch prediction is a necessary feature in order to mitigate branch penalties. Branch prediction structures can be very large, and it is beneficial to avoid accessing them whenever possible. The BTB is particularly expensive to access, as each entry holds an address, and has an associated tag that must be compared with the ToC PC. We first describe techniques for disabling these structures on single fetch pipelines, then explain how they can be applied to multiple fetch pipelines.

5.2.1. Disabling BPB/BTB Accesses for Non-ToCs and Direct Jumps. The primary structures dealing with ToC speculation include the branch prediction buffer (BPB) and the branch target buffer (BTB). We store two sets of bits in the TH-IC that we use to disable these structures when they are not needed. These bits are the NSNB bits (next sequential not a branch) and NTNB bits (next target not a ToC). Each instruction has an NSNB bit and an NTNB bit associated with it, as shown in Figure 4. Whenever the NSNB bit for an instruction is set, it indicates that the instruction that sequentially follows it is not a direct conditional branch. Similarly, when the NTNB bit for an instruction is set, it indicates that the instruction is a direct ToC, and that its target is *not* a direct conditional branch.[1]

We use these bits in the TH-IC by taking advantage of a single *predecode* bit that we have added to the L1-IC. Predecoding is used in many pipeline designs, such as to facilitate detection of calls and returns for activating the return address stack (RAS).

The bit we predecode prior to insertion in the L1-IC is the NB bit, which is set when the instruction is a not direct conditional branch. We fill the NSNB bits in the TH-IC from the predecoded NB bits in the L1-IC, as shown in Figure 10. The predecoded NB bits in the L1-IC are shifted over to initialize previous instructions' NSNB bits. We can also initialize NSNB bits in the previous TH-IC line, provided that the current line was reached by a sequential fetch, and not via a ToC.

We can avoid accessing the BPB and BTB on the following fetch if it is a guaranteed TH-IC hit, and one of the following is true. (1) The following fetch occurs sequentially after the current fetch, and the current instruction's NSNB bit is set. (2) The following fetch is the target of a direct ToC and the current instruction's NTNB bit is set.

Note that now the BPB and BTB may be disabled for direct unconditional jumps and calls. Due to the faster TH-IC access time, and the simple jump/call encoding used in the MIPS ISA, on TH-IC hits we have sufficient time to detect direct jumps and calls, and simply multiplex in the jump target before the next fetch, making the BPB and BTB irrelevant.

We also avoid updating the branch predictor for ToC instructions that do not activate the BPB and BTB. This can result in reduced contention in these structures, potentially improving branch prediction behavior.

The cost of this approach is extra predecode logic (checking for branch opcodes), the NB bits added per L1-IC instruction, and the NSNB and NTNB bits per TH-IC instruction.

[1]We used a simple branch predictor in this study that always mispredicts indirect jumps that are not returns. If an indirect branch predictor is used that stores target addresses for indirect jumps in the BTB, then the NSNB/NTNB bits could instead mean that the next fetched instruction is not a conditional branch and not an indirect jump other than a return.

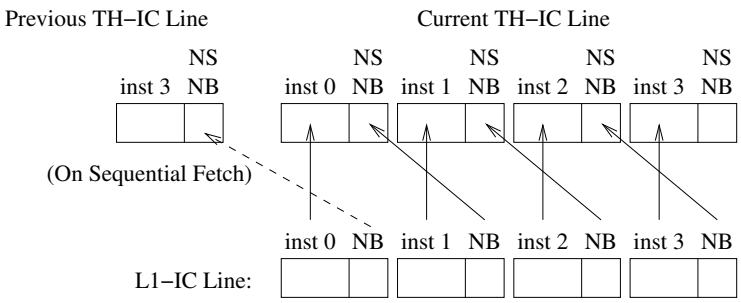

Fig. 10. Filling NSNB Bits from L1-IC Predecode Bits

5.2.2. Disabling BTB Tag Array Accesses. The BTB is a simple cache of ToC targets, and like most caches, has a tag array that comprises a significant portion of its storage. However, since BTBs typically hold a single ToC target per entry, this tag array can be larger than data or instruction caches with similarly sized data arrays. Provided that the BTB has at least as many entries as the TH-IC has instructions, we can take advantage of a set NT bit to disable access to the BTB tag array on sequential fetches. If the fetch on the next cycle will sequentially follow the current instruction, and is either within the same line or the current line's NS bit is set, then we can examine to the next sequential instruction's NT bit. If that bit is set, then we know that the BTB entry for that instruction cannot have been replaced since it was last accessed. In other words, if the BTB entry for the ToC is replaced, then the instruction itself within the TH-IC would have been replaced and the associated NT bit would have been cleared. As a result, a BTB hit is guaranteed when the NT bit is set, and we can avoid the BTB tag check that is normally required.

5.2.3. Disabling BPB/BTB Accesses with Multiple Fetch. When multiple instructions are fetched per cycle, we must slightly adjust the interpretation of the NSNB and NTNB bits. First, we associate the NSNB bits with each fetch group, rather than with each instruction. The total number of NSNB bits remains the same, at one bit per TH-IC instruction. For example, when the pipeline fetches four instructions per cycle, it will access four NSNB bits along with those instructions. However, each bit in this group will indicate the non-branch status of a corresponding instruction in the next sequential fetch group. Thus, when initializing these bits from the NB bits held in the L1-IC, we must shift them over by a whole fetch group, and not just a single instruction.

We also adjust the interpretation of NTNB bits. We still associate each NTNB bit with an instruction, but interpret each NTNB bit to mean that no instruction in the target fetch group (excluding those prior to the target instruction) is also a ToC. We could associate multiple bits per instruction, allowing individual instructions within the target fetch group to be disabled, but this is unlikely to provide much benefit. On average, branches will target the middle of a fetch group, so half these bits will not be used.

If a direct unconditional jump or call instruction is fetched from the TH-IC as a guaranteed hit, then the processor has time in the same cycle due to the faster TH-IC access time to extract the target address from the jump format, which is shown in Figure 3. Accessing the BTB in this case is unnecessary as the target address is already available, regardless of whether or not the target is on the same page. The BPB can also be disabled in this case as direct unconditional jumps and calls are always taken. Thus, when a sequentially accessed instruction (including direct calls and jumps) is a

TH-IC hit and the NSNB bit for the previous instruction is set (meaning the current instruction is not a conditional branch), then both the BTB and BPB are disabled.

This strategy has the added effect that it is not necessary to insert direct jumps and calls into the BPB or BTB. Fewer accesses result in reduced contention in those tables, allowing more room for branches, and thus improving the branch prediction rate.

6. VARIABLE LENGTH INSTRUCTION SET ARCHITECTURES

One of the dominant ISAs in current use is the Intel x86 ISA. Unlike the ISA used in this study, and unlike nearly every other ISA in common use, this ISA uses a variable-length instruction. While such an ISA may appear to make the TH-IC unusable for reducing fetch energy, this is not the case.

In order to adapt the TH-IC to variable-length ISAs, the key concept is to treat each *byte* of an x86 instruction as an individual, complete instruction. The fetch engine can then treat the first $N-1$ bytes of an N byte instruction as noops, and attributes the full effect of the complete instruction to the final byte. Metadata bits are then associated with each byte in the TH-IC, instead of each instruction. For instance, when setting the NT bit for a multi-byte branch instruction, the NT bit associated with the final byte of the branch will be set. With this organization, the multiple-fetch architecture described in this paper can be used to fetch multiple bytes per cycle, thus maintaining performance.

We did not study a TH-IC as just described with a variable-length ISA for this paper. The power savings would likely be marginally lower, as the amount of metadata storage required would be greater than for an ISA with a fixed, 32-bit instruction size.

7. EXPERIMENTAL RESULTS

The baseline shown in the graphs in this section assumes that the fetch associated structures (I-TLB, BPB, and BTB) are accessed every cycle. During our simulations we performed numerous checks to verify the validity of all guarantees. Our model accounts for the additional energy required to transfer an entire line from the L1-IC to the TH-IC, as opposed to just the instructions needed when no TH-IC is used. Our model also properly attributes the additional energy required for configurations that extend the L1-IC with predecoded data. Our experiments include configurations with L1-IC lines of different lengths, while keeping the L1-IC size and associativity constant. TH-IC lines are necessarily always the same length as L1-IC lines, but we included configurations with varying numbers of TH-IC lines.

As a share of total processor power, leakage power has increased with the downward scaling of fabrication technologies. As previously mentioned, in order to cover the range of potential process technologies, we have simulated three different leakage rates, 10%, 25%, and 40%.

7.1. TH-IC

Table II shows the experimentally determined lowest-energy configurations for each fetch width and leakage rate. In this table and the following figures, the notation "*X*/T*Y*" refers to a configuration with an *X*-byte IC line size, and a TH-IC with *Y* lines. The notation "*X*/none" refers to a configuration with an *X*-byte IC line size, and *no* TH-IC.

Figures 11-19 show the total fetch energy for each pipeline, normalized to the best-case L1-IC-only configuration for each pipeline. The most obvious result shown by these graphs is that a larger TH-IC size results in more efficient usage of the other structures, but also increases TH-IC energy usage. The TH-IC hit rate generally increases with size, but eventually its power will exceed the power saved by disabling the other structures.

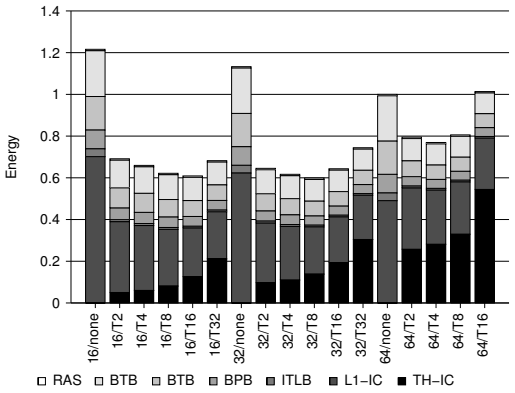

Fig. 11. Fetch Energy, 1-wide Fetch, 10% Leakage

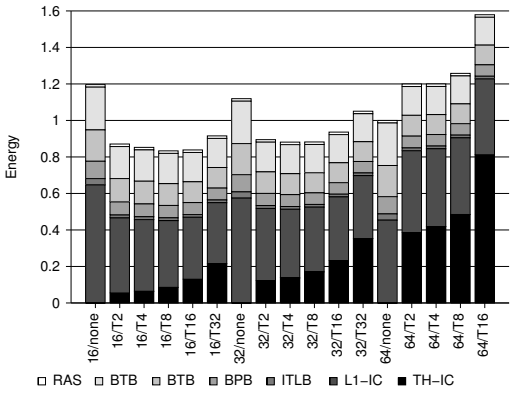

Fig. 12. Fetch Energy, 1-wide Fetch, 25% Leakage

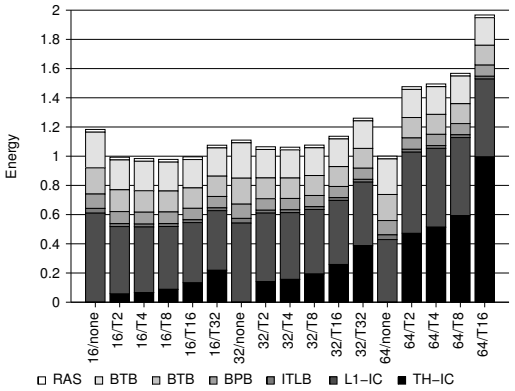

Fig. 13. Fetch Energy, 1-wide Fetch, 40% Leakage

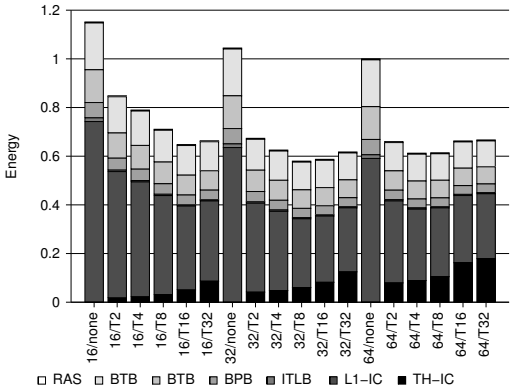

Fig. 14. Fetch Energy, 2-wide Fetch, 10% Leakage

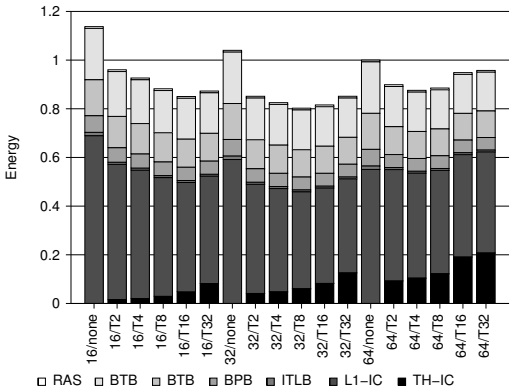

Fig. 15. Fetch Energy, 2-wide Fetch, 25% Leakage

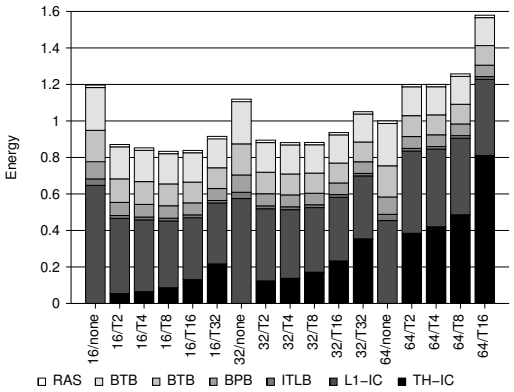

Fig. 16. Fetch Energy, 2-wide Fetch, 40% Leakage

Table II. Optimal Configurations

Fetch Width	10% leakage	25% leakage	40% leakage
1	32/T8	16/T8	64/none
2	32/T8	32/T8	32/T8
4	32/T16	32/T16	32/T8

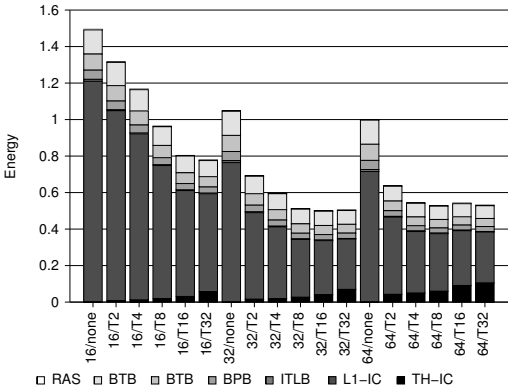

Fig. 17. Fetch Energy, 4-wide Fetch, 10% Leakage

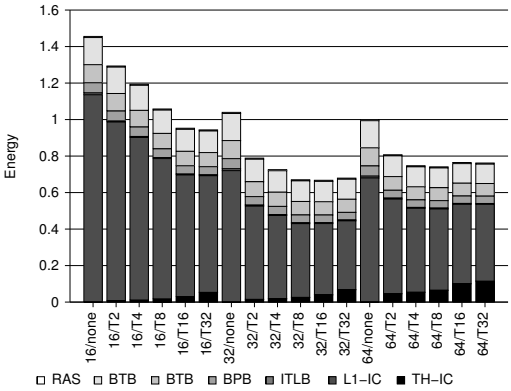

Fig. 18. Fetch Energy, 4-wide Fetch, 25% Leakage

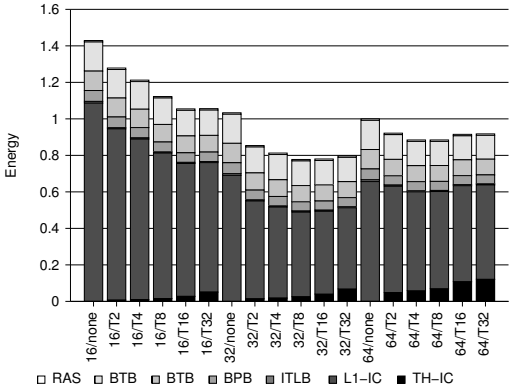

Fig. 19. Fetch Energy, 4-wide Fetch, 40% Leakage

As expected, with higher leakage rates, the effectiveness of the TH-IC diminishes, and smaller TH-ICs are more effective than larger ones. In spite of this, the TH-IC provides benefit in all configurations, although that benefit may be miniscule at higher leakage rates. In addition, it is anticipated that technologies such as *high-k metal gate* (HKMG) will drastically reduce leakage power even as technology scales smaller. For instance, Global Foundries' 28nm HKMG Low-Power/High-Performance Platform (LPH), intended for use in high-end smartphones and tablets, reduces active power by up to 40%, and leakage currents by up to 90% when compared to earlier technologies [Global Foundries 2011]. Where power is a much greater concern than performance, a very low leakage, low power technology will likely be used in combination with a low performance, single issue design, a situation in which the TH-IC is highly effective.

The following paragraphs describe the reduction in access rates of the individual components.

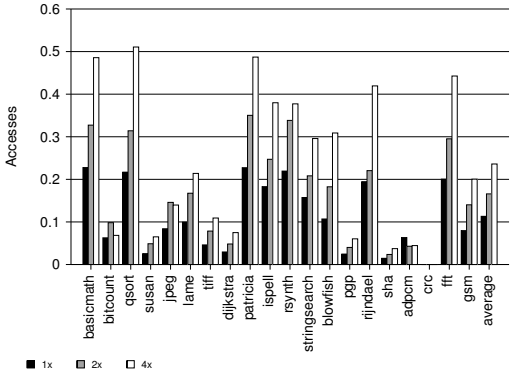

Fig. 20. L1-IC Accesses (Normalized)

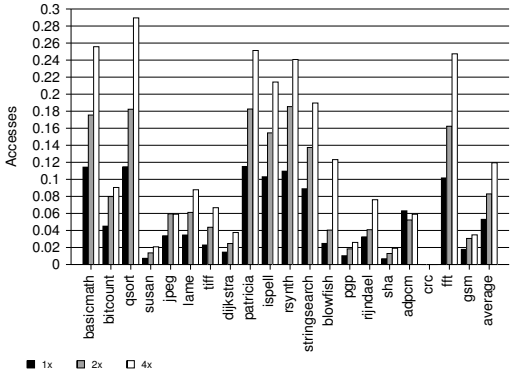

Fig. 21. I-TLB Accesses (Normalized)

Figure 20 shows the resulting L1-IC accesses. This reduction in accesses is the primary contributor of energy savings, as the L1-IC is the largest of the fetch associated structures. We are able to avoid accesses to the L1-IC for between 77% and 88% of instructions in all configurations. Benchmarks with small, tight, and easily predicted innermost loops, such as *pgp*, *sha*, *adpcm*, and *crc* show the best performance. This is due to the high TH-IC hit rate, and optimal branch predictor performance while executing these loops. Benchmarks such as *qsort*, which have higher indirect jump counts, have higher L1-IC access rates, as the targets of indirect jumps cannot be fetched from the TH-IC, and cannot be predicted.

Figure 21 shows the resulting I-TLB accesses. The original TH-IC alone already results in significant reductions in I-TLB accesses (in fact, exactly at the rate for L1-IC accesses), but the techniques presented in this paper further reduce the access frequency down to less than 6% of cycles for 1-wide fetch, less than 9% of cycles for 2-wide fetch, and less than 12% of cycles for 4-wide fetch. Much of the benefit comes from detecting when the current line is not the last line in the page during sequential fetches. However, some of the benefit is obtained by the use of the SP bit for direct ToCs, which allows ToCs that leave their page to be detected. The differences in I-TLB access rates between the benchmarks occur for reasons similar to the differences in L1-IC access rates.

Figure 22 shows the resulting BPB and BTB target array accesses. Figure 23 shows the resulting BTB tag array accesses. The BTB tag access rate is lower than the BPB/BTB target rate because the tag array is only accessed when the branch instruction's NT bit is not set, whereas the target is accessed for all direct branches. We are able to prevent accesses to the BPB and BTB target for at least 77% of instructions for 1-wide fetch, 72% of instructions for 2-wide fetch, and 64% of instructions for 4-wide fetch. We are able to reduce BTB tag array accesses even further, to 83% of instructions for 1-wide fetch, 78% of instructions for 2-wide fetch, and to 68% of instructions for 4-wide fetch. In the MiBench suite, only 14.2% of instructions are direct transfers of control (jumps and branches). In the 1-wide fetch configuration, we are able to come to within 9% of this optimum for BPB and BTB target array accesses, and to within 3% of the optimum for BTB tag array accesses.

Note that, on average, access rates for all the fetch related structures improve when higher fetch widths are used. This is primarily due to the larger L1-IC sizes (and thus higher hit rates) that are present on the more aggressive pipeline models. In our model, L1-IC misses result in reactivation of all structures once the line fill completes, and so higher L1-IC miss rates result in higher energy.

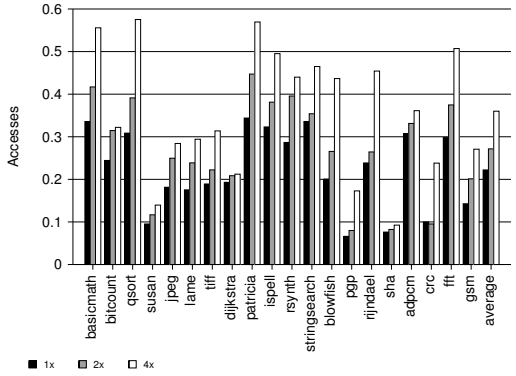

Fig. 22. BPB/BTB Target Accesses (Normalized)

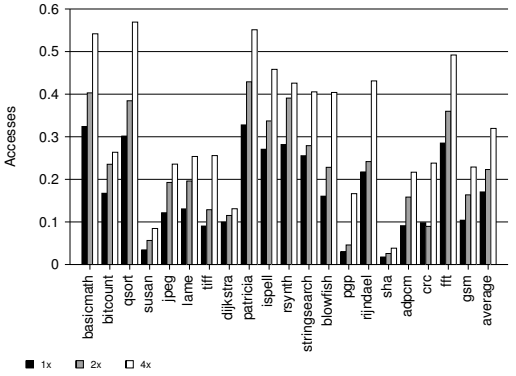

Fig. 23. BTB Tag Accesses (Normalized)

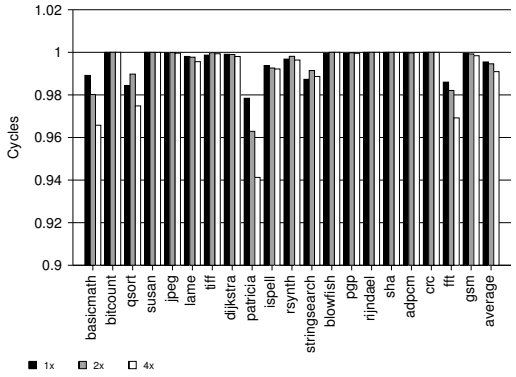

Fig. 24. Simulated Cycles (Normalized)

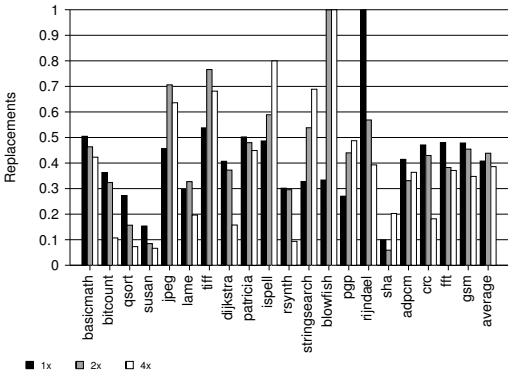

Fig. 25. BTB Replacements (Normalized)

Figure 24 shows the number of execution cycles for each of the simulated benchmarks, normalized against the configuration with no TH-IC (but with the same L1-IC line size). The slight improvement in performance shown here can be explained by the reduction in BTB contention as discussed in 5.2.1, and as evidenced by the corresponding reductions in BTB replacements (Figure 25) and branch mispredictions (Figure 26). Though the frequency of BTB replacements were decreased by similar fractions for all 3 pipelines, the effect on mispredictions is greater on the single-fetch pipeline because its BTB has fewer entries. Note that this improvement may not hold when changing the L1-IC line size, due to different miss rates.

Finally, Figures 27-35 show the energy usage breakdown for each benchmark on the lowest energy TH-IC-based configuration of those we tested. Since the goal is to minimize total fetch energy consumption, whether or not a TH-IC is used, we have chosen to normalize each graph to the optimum L1-IC-only configuration. Thus, each graph shows the ratio of energy savings of the best TH-IC-based configuration versus the best L1-IC-only configuration. This shows that, in all cases, adding an optimized TH-IC provides benefit over simply optimizing the L1-IC and not using a TH-IC.

7.2. Loop Buffers

Loop buffers are small structures that have been proposed for fetching innermost loops using less energy. Their purpose is very similar to the TH-IC: to reduce the number of L1-IC and I-TLB accesses by retrieving the instructions from a much smaller, lower

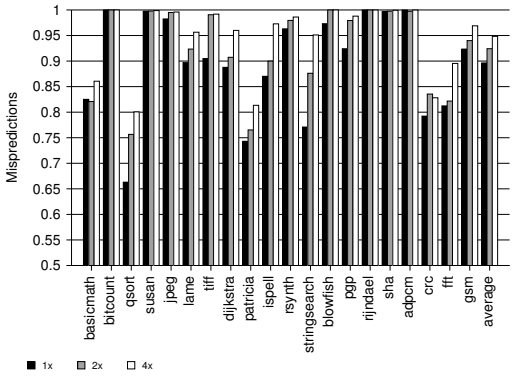

Fig. 26. ToC Mispredictions (Normalized)

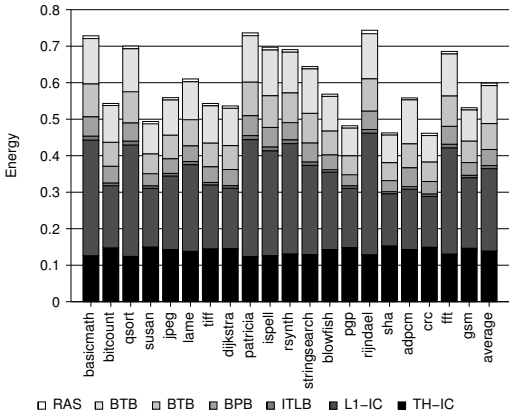

Fig. 27. Fetch Energy, 1-wide Fetch, 10% Leakage, 32/T8 vs. 64/none

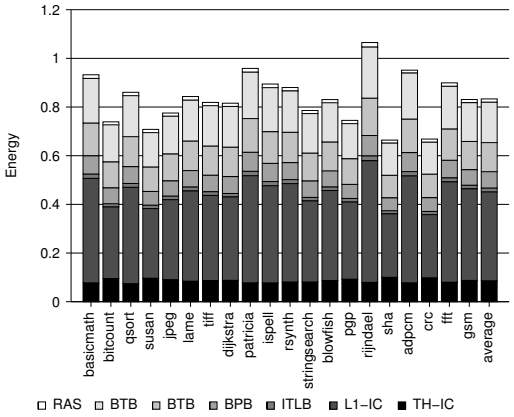

Fig. 28. Fetch Energy, 1-wide Fetch, 25% Leakage, 16/T8 vs. 64/none

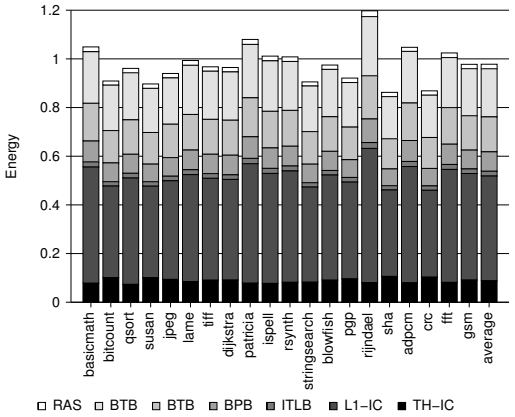

Fig. 29. Fetch Energy, 1-wide Fetch, 40% Leakage, 16/T4 vs. 64/none

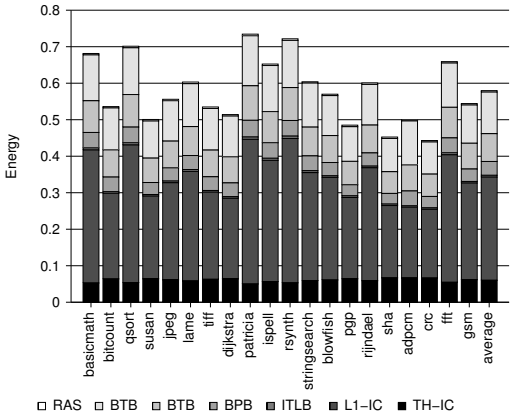

Fig. 30. Fetch Energy, 2-wide Fetch, 10% Leakage, 32/T8 vs. 64/none

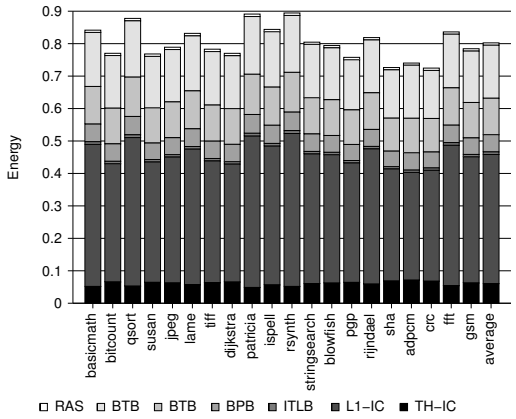

Fig. 31. Fetch Energy, 2-wide Fetch, 25% Leakage, 32/T8 vs. 64/none

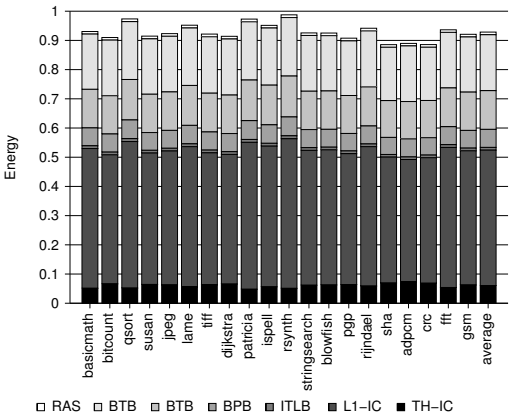

Fig. 32. Fetch Energy, 2-wide Fetch, 40% Leakage, 32/T8 vs. 64/none

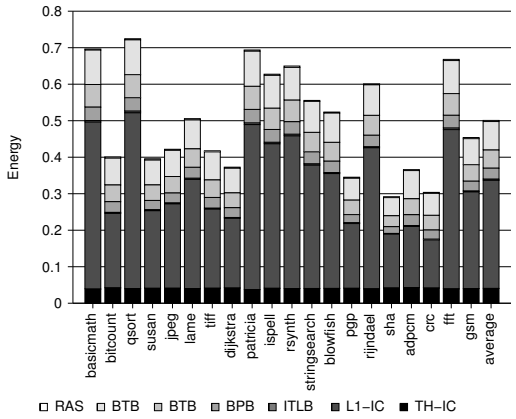

Fig. 33. Fetch Energy, 4-wide Fetch, 10% Leakage, 32/T16 vs. 32/none

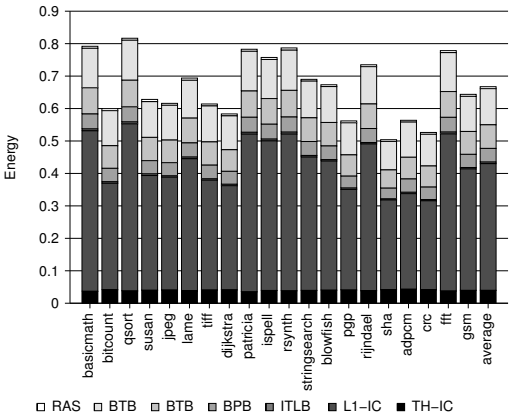

Fig. 34. Fetch Energy, 4-wide Fetch, 25% Leakage, 32/T16 vs. 32/none

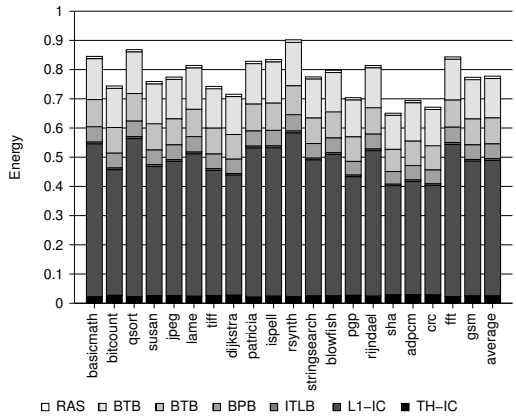

Fig. 35. Fetch Energy, 4-wide Fetch, 40% Leakage, 32/T8 vs. 32/none

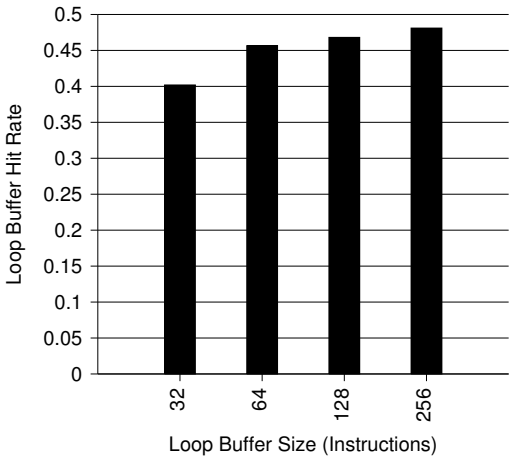

Fig. 36. Loop Buffer Hit Rate

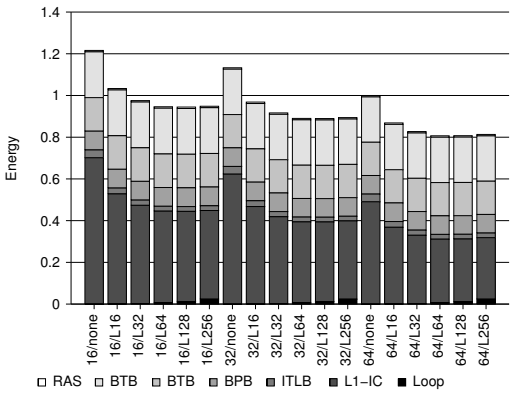

Fig. 37. Loop Buffer Fetch Energy, 1-wide Fetch

power structure. Not only do loop buffers avoid L1-IC accesses, but they also do not require I-TLB accesses while executing from the loop buffer. Loop buffers are automatically exploited when a short backward direct transfer of control is encountered, indicating that looping behavior is likely [Lee et al. 1999]. In order to reduce the rate of false positives, many loop buffer designs will not become active until one or more iterations have occurred.

The simplest of loop buffers can only buffer loops with no (or minimal) control flow. More advanced designs, such as the loop buffer found in the Cortex A15 [Lanier 2011], are capable of handling one or more forward branches, in addition to the backward branch at the end of the loop. This allows simple if/else sequences to be handled, while still fetching from the loop buffer.

We have simulated our baseline models augmented with an "ideal" loop buffer, with sizes equivalent to the sizes of the TH-IC studied. This loop buffer is ideal, in that it can buffer any innermost loop with simple internal control flow. The simulated loop buffer can handle an arbitrary number of forward branch instructions. It can also handle an arbitrary number of backward branch instructions, provided that these backward branches always target the first instruction of the loop. On recovery from a branch misprediction, the loop buffer resumes the most recently buffered loop, if any, unless the misspeculated path results in a new loop being detected. Other backward branches, and all indirect jumps and returns cause the loop buffer to become idle. If an innermost loop exceeds the size of the loop buffer, only the initial instructions of the loop will be buffered, and later instructions will be accessed from the L1-IC. Loops will only be buffered if they meet the above conditions, and after two iterations of the loop have passed.

In nearly all cases the simulated loop buffer hit rate was significantly lower than the TH-IC hit rate (for the same number of instructions stored), and in no case was the loop buffer hit rate over 50%. Figure 36 shows the fraction of instruction fetches that occur from the loop buffer (hit rate), for different loop buffer sizes on a single fetch width core. In no case is the hit rate greater than 50%, while the TH-IC easily manages to exceed 75% of accesses from the TH-IC. (Hit rates for different fetch widths were nearly identical.) The mediocre hit rate achieved by the loop buffer strategy is directly tied to their limited flexibility with regard to control flow. The TH-IC does not have this problem, and can effectively handle loops containing arbitrary conditional and

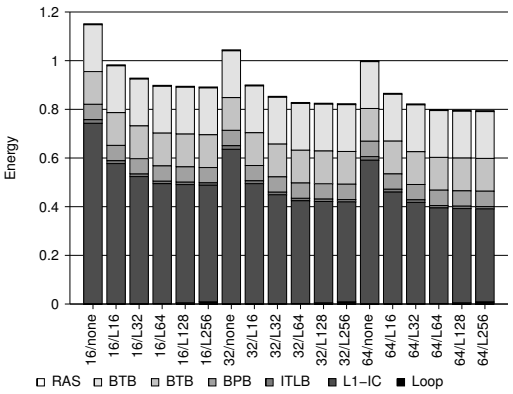

Fig. 38. Loop Buffer Fetch Energy, 2-wide Fetch

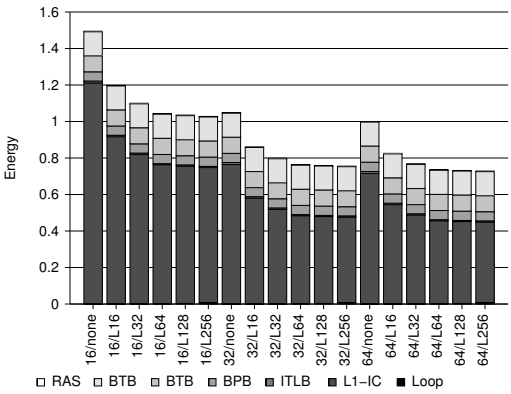

Fig. 39. Loop Buffer Fetch Energy, 4-wide Fetch

non-conditional control flow. Nested loops are easily handled by the TH-IC. The TH-IC can also recover quickly from irregular control flow, such as indirect jumps, which would cause the loop buffer to go idle until the next suitable innermost loop is detected.

As a result of the low hit rate, loop buffer does not reduce fetch energy nearly as well as the TH-IC. Figures 37-39 show the simulated energy usage of the baseline pipelines, augmented with loop buffers of various sizes, under 10% static leakage conditions. In these figures, the notation "*X*/L*Y*" refers to a configuration with an *X*-byte IC line size, and a loop buffer with storage for *Y* instructions. At best, the ideal loop buffer saves 20-25% of the energy used for instruction fetch, while the TH-IC achieves savings in excess of 40%. In addition, the loop buffer simulated here has fewer restrictions than any loop buffer found in an actual processor, and so the hit rates achieved in reality would likely be lower.

8. RELATED WORK

Zero overhead loop buffers (ZOLBs) are structures in which loops are explicitly loaded and are sometimes used in embedded processors [Eyre and Bier 1998]. The ZOLB benefits include reducing loop overhead due to using an internal counter and avoiding L1-IC, I-TLB, and BPB/BTB accesses after the first iteration of the loop. However, a ZOLB also provides limited benefit as the loops it can exploit must have a known number of iterations, must have no conditional control flow except for the loop branch, and must be small enough to fit into the ZOLB. In contrast to both of these approaches, our techniques can disable access to the L1-IC, I-TLB, and BPB/BTB much more frequently.

The trace cache [Rotenberg et al. 1996; Rotenberg 1999] was developed to widen the instruction fetch bottleneck by allowing instruction fetch accesses to traverse transfers of control. It achieves this by concatenating multiple separate basic blocks into a single cache line, which is indexed by the initial PC combined with the taken/not-taken states of each branch in the sequence of blocks. Since the cache is indexed using the branch pattern of the sequence, different branching patterns will lead to duplication of basic blocks within the trace cache. The size of the trace cache is typically fairly close to the size of the L1 cache, while the size of the TH-IC is much smaller. These factors combine to make trace caches less suitable for low-power, embedded architectures.

Kadayif et al. studied not accessing the I-TLB unless the virtual page changes [Kadayif et al. 2007]. One technique they evaluated was to examine the updated PC (sequential accesses, BTB lookups for branches, etc.) and compare the virtual page number with the current virtual page number. This approach works well as long as the

comparison being performed does not affect the critical path for the cycle time as the comparison occurs after a new PC value is obtained, which is typically near the end of the cycle. In contrast the techniques for disabling the I-TLB in this paper should not put extra logic on the critical path. For instance, we determine if the next sequential fetch will cross a page boundary by checking at the beginning of the cycle if the address of the current instruction is the last instruction in the page and is only required to perform this check when the last line in the TH-IC is the last line in the page. We also detect when the target of a ToC instruction is to the same page and sets an SP bit that is used to disable the I-TLB access in the next cycle if the ToC is taken. Kadayif et al. also suggested that the format of ToC instructions could be changed to indicate whether or not the target is to the same page. We capture most of this benefit while not requiring any ISA changes.

There have also been proposed alternative BPB and BTB designs to reduce energy consumption. A leakage power reduction technique has been proposed to make BPB and BTB entries drowsy if they are not accessed after a specified number of cycles [Hu et al. 2002]. The use of counters will offset some of the energy benefits. Another BTB design proposed using fewer tag bits to save power at the cost of some false positive tag matches [Fagin and Russell 1995]. We believe that this paper's proposals are complementary to both of these techniques, and could be advantageous in combination, as the number of accesses are reduced, and so more targets are available for activation of drowsy mode. Yang et al. proposed a small branch identification unit and avoided accessing the BTB until the next branch is encountered. But this approach requires changing executables to store branch distance information and also requires the use of a counter to detect the next branch [Yang and Orailoglu 2006].

Parikh et al. proposed disabling access to the BTB and BPB to save energy by using a *prediction probe detector* (PPD) that contains two bits for each instruction in the L1-IC [Parikh et al. 2002]. These two bits indicate whether or not the BPB and BTB should be accessed and these bits are assigned values while an L1-IC cache line is being filled. These authors proposed that the PPD be checked before going to the BPB or BTB, which they recognized may sometimes be on the critical timing path. In addition, the predecode bits used in this paper are used to set NSNB bits when a line is loaded from the L1-IC into the TH-IC. In contrast to the PPD approach, the techniques presented in this paper do not affect the critical timing path.

Calder et al. proposed making all branches use page offsets instead of PC offsets to allow fast target calculation [Calder and Grunwald 1994]. These authors proposed to use indirect jumps through a register for branches across page boundaries. They also used predecode bits to indicate whether or not an instruction is a branch. The disadvantages of this approach are that it requires an ISA change and the page offset extraction from the L1-IC may be on the critical timing path.

Hines et. al. [Hines et al. 2009] introduce NSNB and NTNB bits in order to disable branch prediction structures for non-ToCs. That work does not discuss a method for using these bits on multiple-fetch processors. Additionally, it proposes waiting until instructions are fetched and decoded to detect whether it is a transfer of control, and requires that this detection is completed before the end of the fetch stage. By using predecoding as described in our research, these bits can be used before an instruction is even accessed, dramatically increasing their effectiveness. That work does not take advantage of the other techniques listed in this paper for disabling the I-TLB and BTB tag arrays. We were able to achieve much greater power reduction with these additional techniques than was found in that paper.

9. FUTURE WORK

Disabling a high percentage of accesses to the I-TLB, BPB, and BTB may provide opportunities to exploit different sizes of these structures. The I-TLB size could possibly be increased to avoid more page table accesses as a higher access energy would be offset by most of the I-TLB accesses being avoided when using a TH-IC. The BPB and BTB may benefit from having fewer entries to require less power on each access as we have decreased contention to these structures.

In the original TH-IC work [Hines et al. 2007], several variations were provided for the TL bit vector associated with each line. These include a TI variant, where one bit is stored for every instruction in the TH-IC. This variation was not studied in this paper, because the storage requirements are significantly larger for this configuration, which outweighs the benefits of increased precision while invalidating NT bits. However, some analysis reveals that the TL configuration is in fact a Bloom filter [Bloom 1970] on the TI array, with a hash function that simply strips the index of the instruction within the line. Any Bloom filter can in fact be used for this purpose, as false positives are always allowed, and simply result in NT bits being unnecessarily cleared. It may actually be more efficient to use a more complex Bloom filter with multiple, more effective hash functions. Such a Bloom filter may provide more effective compression of the metadata, or offer increased precision using the same number of metadata bits.

Studies have investigated permuting the order of functions [Kalamatianos and Kaeli 1998] and/or basic blocks [Pettis and Hansen 1990] to improve L1-IC performance and decrease the number of taken branches, which can cause delays on some processors. Permuting the order of functions and/or basic blocks can also now be used to reduce energy dissipation if the percentage of ToC instructions with targets to the same page can be increased.

A final obvious area of future work is to model a processor using these techniques in a hardware description language (e.g., VHDL, Verilog) and use EDA design tools (e.g., Synopsis, Cadence) to implement the hardware design. We can then validate the power reduction and verify that the cycle time is unaffected.

As mentioned, our experiments required that fetch groups are aligned to the size of the fetch group. Many aggressively superscalar processors do not have this restriction, and can access instructions spanning alignment boundaries. This is achieved, for example, by splitting cache lines between a pair of banks, and interleaving accesses appropriately [Conte et al. 1995]. Such designs are must more complex than those that only allow aligned accesses. The cache may need to decode the address multiple times, because multiple lines are accessed. Additional selection logic and multiplexers are needed in order to align the instruction data prior to supplying it to the remainder of the pipeline.

The TH-IC could be modified to handle cross-line fetching, for instance, by adding an NS bit to the instruction in each cache line that precedes the point at which the line is split in half. Fetches from the TH-IC could then access two halves of a cache line, discarding the second sequential half if the first half's NS bit was not set. Alternatively, each half of each TH-IC line could have *two* NS bits (a total of four bits per cache line). The first bit would indicate that the next sequential half-cache-line is present in the TH-IC, and the second would indicate that the half-cache-line following that is *also* present in the TH-IC. When accessing a half-cache-line, the pair of NS bits associated with it would indicate whether the next two half-cache-lines are present in the TH-IC, or must be fetched from the L1-IC.

The obvious costs of such a TH-IC implementation are the additional metadata bits and bookkeeping required.

10. CONCLUSIONS

The techniques we present in this paper are able to reduce the power and energy usage of a processor by more efficiently fetching the most frequently executed instructions. We present novel techniques that take advantage of guarantees so that the I-TLB, BTB, and BPB can be frequently disabled, and apply these techniques to both single and multiple fetch architectures. We are able to exploit unique aspects of the TH-IC, such as its faster access time and smaller size, to make additional guarantees to disable access to these structures. Our experimental results show that these techniques significantly decrease the total accesses to each of these fetch associated structures, which results in an over 40% decrease in fetch energy in our model for some configurations. In addition, we are able to obtain small performance gains by eliminating many branch mispredictions due to reduced BPB/BTB contention and avoiding some mispredictions for direct unconditional ToCs that have traditionally been considered compulsory. All of these improvements are obtained with the addition of only one bit of metadata per L1-IC instruction, and three bits per TH-IC instruction. These techniques require no changes to the instruction set and can easily be integrated into most processors, which tend to have similar instruction fetch designs. These techniques may provide additional benefits by alleviating hot spots on a processor, which can improve reliability and longevity. We believe our techniques are superior to the related work because we are more effective at reducing dynamic activity, our techniques do not place pressure on circuit timing, and we do not require any changes to the ISA or programs. These features combine to make an attractive solution for improving processor energy efficiency for a wide variety of processor designs.

REFERENCES

ARM Holdings. 2013. ARM Information Center. Website. (2013). http://infocenter.arm.com/help/index.jsp

Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer* 35 (February 2002), 59–67.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. DOI:http://dx.doi.org/10.1145/362686.362692

David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 83–94. DOI:http://dx.doi.org/10.1145/339647.339657

Doug Burger and Todd M. Austin. 1997. *The SimpleScalar Tool Set, Version 2.0*. Technical Report 1342. University of Wisconsin - Madison, Computer Science Dept. citeseer.ist.psu.edu/245115.html

Brad Calder and Dirk Grunwald. 1994. Fast and Accurate Instruction Fetch and Branch Prediction. In *International Symposium on Computer Architecture*. 2–11.

Thomas M Conte, Kishore N Menezes, Patrick M Mills, and Burzin A Patel. 1995. Optimization of instruction fetch mechanisms for high issue rates. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE, 333–344.

J. Eyre and J. Bier. 1998. DSP Processors Hit the Mainstream. *IEEE Computer* 31, 8 (August 1998), 51–59.

Barry Fagin and Kathryn Russell. 1995. Partial Resolution in Branch Target Buffers. In *ACM/IEEE International Symposium on Microarchitecture*. 193–198.

Global Foundries. 2011. Global Foundries 32/28nm HKMG Platforms. Website. (2011). http://www.globalfoundries.com/newsletter/2011/2/3228hkmg.aspx

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE 4th Annual Workshop on Workload Characterization* (December 2001).

Stephen Hines, Yuval Peress, Peter Gavin, David Whalley, and Gary Tyson. 2009. Guaranteeing Instruction Fetch Behavior with a Lookahead Instruction Fetch Engine (LIFE). In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*. 119–128.

Stephen Hines, David Whalley, and Gary Tyson. 2007. Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache. In *ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 433–444.

Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. 2002. Applying Decay Strategies to Branch Predictors for Leakage Energy Savings. In *Proceedings of the International Conference on Computer Design*. 442–445.

I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. 2007. Generating Physical Addresses Directly for Saving Instruction TLB Energy. In *Proceedings of the 40th annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 433–444.

John Kalamatianos and David R. Kaeli. 1998. Temporal-based Procedure Reordering for Improved Instruction Cache Performance. In *International Symposium on High Performance Computer Architecture*. 244–253.

Johnson Kin, Munish Gupta, and William H. Mangione-Smith. 2000. Filtering Memory References to Increase Energy Efficiency. *IEEE Trans. Comput.* 49, 1 (2000), 1–15. DOI:http://dx.doi.org/10.1109/12.822560

Travis Lanier. 2011. Exploring the Design of the Cortex-A15 Processor. Presentation slides. (2011). http://www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf

L. Lee, B. Moyer, and J. Arends. 1999. 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*. 267–269.

D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. 2002. Power Issues Related to Branch Prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 233–244.

Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 16–27. DOI:http://dx.doi.org/10.1145/93542.93550

Eric Rotenberg. 1999. A Trace Cache Microarchitecture and Evaluation. *IEEE Trans. Comput.* 48 (1999), 111–120.

Eric Rotenberg, Steve Bennett, James E. Smith, and Eric Rotenberg. 1996. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *In Proceedings of the 29th International Symposium on Microarchitecture*. 24–34.

S. Wilton and N. Jouppi. 1996. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid State Circuits* 31, 5 (May 1996), 677–688.

Chengmo Yang and Alex Orailoglu. 2006. Power Efficient Branch Prediction through Early Identification of Branch Addresses. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 169–178.