THIC and OpenSPARC: Reducing Instruction Fetch Power in a Multithreading Processor

Peter B. Gavin, Stephen R. Hines, Gary S. Tyson, and David B. Whalley

***Abstract*—The OpenSPARC T1 is a multithreading processor developed and open sourced by Sun Microsystems (now Oracle) [1]. We have added an implementation of our low-power Tagless-Hit Instruction Cache (TH-IC) [2] to the T1, after adapting it to the multithreading architecture found in that processor. The TH-IC eliminates the need for many instruction cache and ITLB accesses, by guaranteeing that accesses within a much smaller L0-style cache will hit. OpenSPARC T1 uses a 16KB, 4-way set associative instruction, and a 64-entry fully associative ITLB. The addition of the TH-IC eliminates approximately 75% of accesses to these structures, instead processing the fetch directly from a much smaller 128 byte data array. Adding the TH-ICto the T1 also demonstrates that even processors designed for high-throughput workloads can be suitable for use in embedded systems.**

I. Introduction

In modern embedded and low-power designs, instruction fetch has been found to be a major component of processor energy consumption–up to 27% of the total processor power requirements on a StrongARM SA110 [3]. Instruction fetch typically involves many large data structures that require substantial energy to access, and must take advantage of speculation in order to provide acceptable performance. Such speculative operation can be quite explicit, as is the case with a branch predictor, which makes outright guesses as to conditional branch outcomes. Implicit speculation also exists, as is the case with the instruction cache–local copies of recently executed instructions are kept with the hope that they will be used multiple times before being discarded. The trouble with speculation, in the context of embedded processor design, is that it typically results in the duplication (or obsolescence) of computational effort. This extra effort manifests itself as additional processor power usage, by way of increased circuit size and additional transistor switching events per cycle. As a result, many embedded systems' designers are hesitant to include too many speculative features, lest they exceed power constraints. On the other hand, reduced performance leads to increased energy consumed per task, so it may in fact be detrimental that many designs have left speculation out, depending on the workload. Consequently, the extent of the speculation to be used in a system must be carefully chosen by the architect.

Speculation in instruction fetch primarily occurs in the prediction of the answer of a question based only on the program counter (PC) and static properties of the instructions themselves. The tag-checking mechanism within the instruction cache provides an example of such a question: *"Will this instruction fetch hit in the cache?"* (The speculation here is that the answer is usually predicted to be yes.) It turns out that we can cache the answers of many of these questions, and reuse them later on, just as we do for other forms of data. As we shall see, this is especially effective when done for the instruction fetch mechanism, since instructions are nearly always fetched in a regular, repetetive, and predictable order. Most instructions that are executed occur over multiple passes through the same sequences of addresses. If we execute the first pass through a sequence of instructions with speculation enabled (as normal), but also cache the important results, we can disable speculation on subsequent passes since we will already know the answers! An important property such a system must have is that switching between speculative and non-speculative modes must never introduce latency; doing so would negate nearly any benefit in using speculation in the first place.

Enterprise computing is another area in which power consumption is of concern. In an effort to maximize operations per watt (as opposed to operations per second), Sun Microsystems has developed a line of 64-bit UltraSPARC chip multithreaded multiprocessors [1]. These processors have multiple cores per die, connected to an on-die L2 cache via a crossbar. Each core is capable of executing multiple threads simultaneously, by interleaving their execution across pipeline stages. This allows cycles that typically would be lost due to pipeline bubbles to be filled by other threads running on the same core. Since fewer cycles are lost, total throughput (in instructions commited per cycle) increases. As a direct consequence, power is saved as well.

This paper presents and examines our newly completed implementation of our Tagless-Hit Instruction Cache (TH-IC) [2], [4] for the OpenSPARC T1 microprocessor. We first provide background by describing

S. R. Hines is with NVIDIA Corporation, Santa Clara, CA.
P. B. Gavin, G. S. Tyson, and D. B. Whalley are with the Department of Computer Science at Florida State University, Tallahassee, FL.

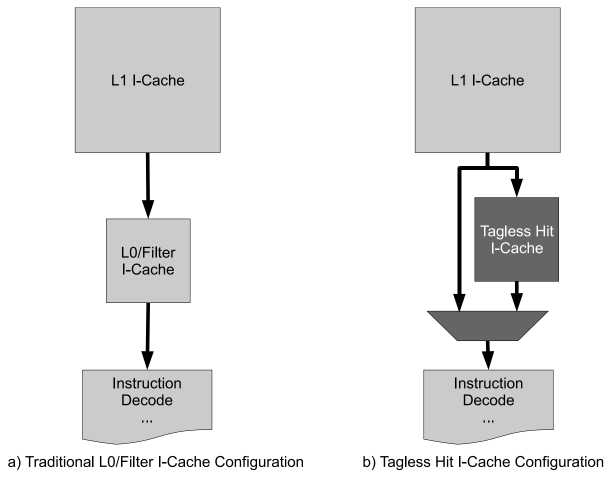


Fig. 1. Traditional L0/Filter and Tagless Hit Instruction Cache Layouts

the operation of the TH-IC, which functions similarly to an L0 instruction cache, but introduces no latencies and requires no tag check or address translation. We will also discuss methods for disabling several optional instruction fetch related structures. Finally, we provide some preliminary results showing the expected power saved by this integration.

II. The Tagless Hit Instruction Cache

The Tagless Hit Instruction Cache (TH-IC), is a small, modified direct mapped instruction cache that is meant to be used in conjunction with a standard L1 instruction cache (L1-IC). The small size of the TH-IC is intended to allow energy-efficient accesses, when compared to the typically much larger and set-associative L1-IC. Although similar in concept to an L0 or filter instruction cache [5], [6] as shown in Figure 1, these caches incur a 1-cycle miss penalty prior to fetching the appropriate line from the L1-IC. These miss penalties can accumulate 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. A TH-IC of similar size to an L0-IC has nearly the same hit rate, but does not suffer a miss penalty, alleviating the concerns of performance minded architects.

A second (though no less substantial) benefit of the TH-IC is that it frequently eliminates the required L1-ICtag array and ITLB accesses required when fetching instructions. Since these structures are only used to check for a hit in the L1-IC, guaranteed hits make them superfluous. The combination of these two benefits provides savings in instruction fetch energy consumption that exceed that of a standard L0-IC, at performance equal to a machine using only an L1-IC. Our work is similar in concept to the loop cache [7], but is more general, and handles a wider range of code behavior such as forward branches. The TH-IC requires more complex logic than any of the mentioned caches, however.

A. *Using Caching to Provide Guarantees*

When sequentially fetching from one line of instructions to the next, a tag check will be required when fetching the first instruction of the second line. However, by fetching the second line from the L1-IC and writing it to the TH-IC, we will have ensured that subsequent fetches hit, provided no evictions occur. We can cache the fact that we've created such a sequential pair with a single bit per line (which we call the NS bit). We set the NS bit for the first line of a sequential pair after we have ensured that the pair is actually sequential in the TH-IC. We associate this bit with the first line, and not the second, because typically when SRAM read data is required for a given cycle, the address and enable signals must be provided sometime during the previous cycle. Assuming no evictions occur, the next time we reach the end of the first line, its NS bit will be true, and so we know we can sequentially access the following line without checking its tag. The NS bit corresponding to a line will need to be reset whenever the line or its successor are evicted or invalidated.

We can distinguish three types of control transfers. Most control transfers are caused by direct branches. A direct branch is a branch that transfers control to the same location every time it is executed. This location is typically encoded in the instruction itself, as an offset from the instruction's PC. Since it may take several cycles to determine the target under normal execution, it is frequently cached in a branch target buffer (BTB), allowing the target to be used the following cycle. The second type of control transfers are caused by indirect branches. Indirect branches typically set the PC to an offset from the value stored in some architected register. Since the resulting branch target PC has no relation to its PPC, there is no question whose answer can be cached that will guarantee the target PC's presence on subsequent traversals. The final type of non-sequential flows are traps. A trap is an event, possibly unrelated to the instruction(s) being executed, that causes the PC to be set to a predetermined value. Each trap event type is generally associated with a corresponding numeric identifier, which is used to index into an array of PC values (possibly stored in main memory). Since the trap PC holds no relation to its PPC, there again exists no question whose answer can be cached to guarantee presence on subsequent traversals. Traps are relatively infrequent, and often cause pipeline flushes and other latencies to occur, so little is lost by not handling them.

Logic similar to that used to derive the NS bit can be used to handle direct branches, since the targets of these branches never change. Once we've fetched the target of a given branch from the L1-IC and stored it within the TH-IC, we will have ensured that subsequent fetches of the target through that branch will hit in the TH-IC, assuming no evictions occur. We can cache this result, but we need to keep one bit per instruction, instead of per line as was the case for the NS bits. We call these bits the NT bits. We set the NT bit for a branch once we've ensured its target is present in the TH-IC. Each subsequent fetch of a taken branch's target can thus be completed without performing a tag check, as long the NT bit for the branch is true. We associate the NT bit with the branch, and not the target, because a given target may be associated with multiple branches, and (if that's not enough) because the branch's target usually isn't determined with enough time left in the cycle to access its NT bit. There are multiple ways to decide when to invalidate this bit, but generally, the NT bit for a given branch instruction needs to be cleared whenever the branch's line is evicted, or when the line containing the branch's target is evicted.

Now we have eliminated accessing the tag array for the most prevalent types of instruction fetch sequences, by caching results collected during initial passes to provide guarantees for later passes. It is still unclear what should be done when a TH-IC hit is not guaranteed, however. We can't just check the tag in the TH-IC, and then process the fetch with the L1-IC on a miss, as that would introduce (at minimum) a 1-cycle latency. Since we know before hand that we need to check the tag, we'll just let the L1-IC process the fetch in place of the TH-IC. There will be times when a hit is not guaranteed, but a hit would have ocurred in a traditional L0-IC–a *false miss*. We cannot treat false misses the same as true misses or evictions, and clear any affected NS and NT bits, as that would unnecessarily prevent hits from being guaranteed further along in the program. Instead, we'll do a tag check in the TH-IC *and* the L1-IC every fetch that is not guaranteed to hit in the TH-IC, and only clear the NS and NT bits when the TH-IC tag doesn't match or the L1-IC misses. Although we miss some opportunities to disable the L1-IC with this solution, any power savings lost are mitigated by avoiding the cache-miss penalty.

B. Other Concerns

In order to reduce the effective tag size of the TH-IC, we can employ a subtle approach based on Ghose and Kamble's work with multiple line buffers [8]. Instead of storing a large tag for the remainder of the address in the TH-IC, it is sufficient to identify the corresponding line in the L1-IC by storing the set index and the location of the line in the set. Not storing the entire tag in the TH-IC is possible since the L1-IC is being accessed simultaneously and we can force a true miss in the TH-IC whenever the L1-IC misses. The cache inclusion principle guarantees that any line in the TH-IC must also reside in the L1-IC. Thus by detecting an L1-IC hit *and* verifying that the precise L1-IC line corresponds to our TH-IC line, we can effectively determine whether we have a false miss.

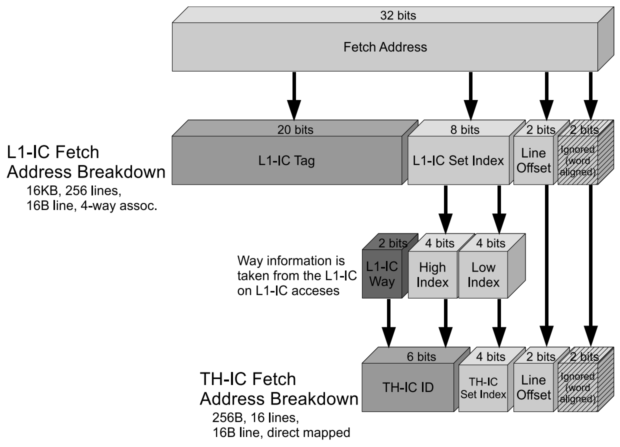


Fig. 2. Fetch Address Breakdown

Note that whenever the tag check is avoided due to a guaranteed hit in the TH-IC, we usually are also able to avoid an ITLB access. Since the total size of the TH-IC is assumed to be smaller than the memory page size, the TH-IC line index will not be affected by virtual to physical address translation. The higher order bits that are translated are only used to verify the L1-IC tag, and so translation is not needed.

C. A Sample Organization

In order to better understand the operation of the TH-IC, it is helpful to consider concrete cache sizes. For this example, we use a 16 KB, 256 line, 16 byte (4 instruction) line size, 4-way set associative L1-IC. We also use a 256 B, 16 line, 16-byte line size, direct-mapped TH-IC.

A breakdown of fetch addresses into their separate bitwise components for properly accessing the various arrays present in our memory hierarchy is shown in Figure 2. Instructions are word-aligned, so the low-order two bits of any fetch address can be safely ignored. Two bits are used to determine the L1-IC line offset, while 8 bits are necessary for the set index, leaving 20 bits for the tag. Two bits are again used to determine the TH-IC line offset, while 4 bits are used for the TH-IC line index. As described earlier, the TH-IC *ID* is derived

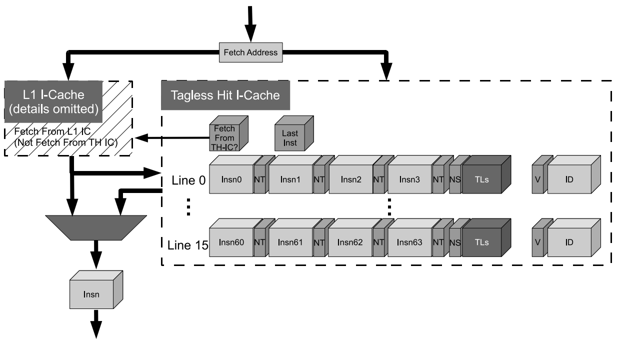
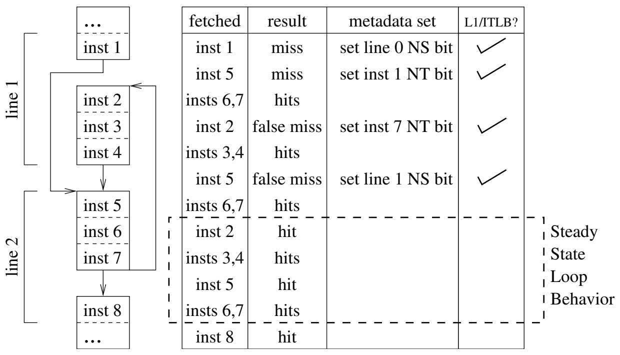


Fig. 3. Tagless Hit Instruction Cache



fetched	result	metadata set	L1/ITLB?
inst 1	miss	set line 0 NS bit	✓
inst 5	miss	set inst 1 NT bit	✓
insts 6,7	hits		
inst 2	false miss	set inst 7 NT bit	✓
insts 3,4	hits		
inst 5	false miss	set line 1 NS bit	✓
insts 6,7	hits		
inst 2	hit		
insts 3,4	hits		
inst 5	hit		
insts 6,7	hits		
inst 8	hit		

Fig. 4. Example TH-IC Instruction Flow

from the L1-IC "way" (2 bits) and enough higher-order bits of the L1-IC set index to uniquely identify the line (4 bits).

Figure 3 shows a detailed view of the organization of the TH-IC and its position within the fetch datapath. As described earlier, we have annotated each line in the TH-IC with an NS bit, and each instruction with an NT bit. Additionally, each line has an TH-IC *ID*, and a bitfield named *TX*, which is used for invalidation, and will explained further in the next section. Finally, included within the TH-IC is the *guaranteed hit* (GH) bit. When GH is true, the current instruction fetch was determined to be a guaranteed hit while fetching the previous instruction. When GH is false (a *potential miss*), the fetch is processed by the L1-IC, and data within the TH-IC are updated. The desired instruction may actually be present in the TH-IC; this is a *false miss*. The case where the instruction is *not* present in the TH-IC is called a *true miss*.

The TH-IC *ID* field is made up of the additional high-order bits from the L1-IC index, and two bits for specifying which line within an associative set corresponds to a particular address. When we are updating the TH-IC (on a potential miss), we are already accessing the L1-IC, so we only need to compare whether we have the appropriate set and way from the L1-IC already in the TH-IC. The miss check can be done by concatenating the two-bit way information for the currently accessed line in the L1-IC and the five high-order bits of the address corresponding to the L1-IC set index, and comparing this result to the stored TH-IC ID of the given set. If these seven bits match, then the TH-IC currently contains the same line from the L1-IC and we indeed have a false miss. If these bits do not match, or the L1-IC cache access is also a miss, then we have a TH-IC true miss and must update the line data as well as the TH-IC ID with the appropriate way and high index information. The ID field can be viewed as a line pointer into the L1-IC that is made up of way information plus a small slice of what would have otherwise been the TH-IC tag. If the L1-IC were direct-mapped, the ID field would only consist of the extra bits that are part of the L1-IC set index but not the TH-IC set index. The cache inclusion property thus allows us to significantly reduce the cost of a tag/ID check even when the TH-IC cannot guarantee a "tagless" hit.

D. An Example Instruction Fetch Sequence

Figure 4 shows an example that illustrates how instructions can be guaranteed to reside in the TH-IC. The example in the figure contains eight instructions spanning four basic blocks and two lines within the TH-IC. Instruction 1 is fetched and is a miss. The previous line's NS bit within the TH-IC is set since there was a sequential transition from line 0 to line 1. Instruction 5 is fetched after the transfer of control and it is also a miss. Instruction 1's NT bit is set to reflect that the target of instruction 1 resides in the TH-IC. Instructions 6 and 7 are fetched and are guaranteed to be hits since they are sequential references within the same line. Instruction 2 is fetched and it resides in the TH-IC, but it is a false miss since it was not guaranteed to hit in the TH-IC (instruction 7's NT bit is initially false). At this point, the NT bit for instruction 7 is set to indicate its target now is in the TH-IC. Instructions 3 and 4 are fetched and are hits due to the intra-line access. Instruction 5 is fetched and is a false miss (line 1's NS bit is false). Line 1's NS bit is set at this point indicating that the next sequential line now resides in the TH-IC. The instructions fetched in the remaining iterations of the loop are guaranteed to be hits since the TH-IC metadata indicates that the transitions between lines (line 1's NS bit and instruction 7's NT bit) will be hits. Finally, instruction 8 is fetched and will be a hit since it is a sequential reference within the same line.

E. TH-IC Line Eviction and Invalidation

There are two events that can require that a line be removed from the TH-IC. The first of these is a conflict eviction. In the TH-IC, a conflict eviction can only occur as a result of true miss, and always causes the line to be replaced by a new line. The other type of event is an external invalidation. External invalidations are caused by an non-fetch-related event, and do not replace removed line with a new one. Events that can cause an external invalidation include writes to the instruction stream by store instructions, and invalidations required to maintain cache coherency between processor cores.

Whenever either of these events occur, NS and NT bits must be reset to maintain correct operation. For example, the NS bit for a line must be cleared when the sequential line in the TH-IC is replaced by a line that is not sequential in memory. Clearing NT bits is more difficult, since multiple instructions may branch into a given line. When the line is evicted, the NT bits for all of these instructions will have to be reset. We present a number of related methods for determining which NT bits to clear that vary in complexity, from conservative approximations, to more precise tracking of the relations between the transfers of control and their target instructions. Most of these methods involve adding a bit field to each line, which we call, in general, the *TX* field. We have studied several specific configurations for the TX field, including an "oblivious" mode where no TX bits are allocated, and all NT bits are reset for any line replacement. Other modes include TI (one TX bit for each potential branch instruction per line) and TL (one TX bit for each potential branch line per line).

Bits within the TX field for a given line can be cleared whenever the lines (instructions) that branch to that line are invalidated. This prevents an NT bit from being cleared due to TX bits that were set for branches no longer present within the TH-IC. The logic required to clear TL/TI bits can be quite complex, and neglecting to clear them may lead to false positives. Fortunately, whenever the TX bits are used to clear NT bits, the TX bits themselves are cleared, so at most one dynamic guarantee will be lost per affected branch instruction. Since branches constitute only a fraction of all instructions executed, a false positive is more likely to clear the NT bit for a non-branch (having no effect), than to clear the NT bit for an actual branch (causing unnecessary potential misses). In the interest of power savings, we can thus choose *not* to attempt to clear individual TX bits.

Since we have decided that false positives are acceptible, and not attempt to clear individual bits, we can generalize the TX field to an arbitrary Bloom filter. Using a Bloom filter allows us to "compress" the TX field to a smaller number of bits. Alternatively, we can reduce the granularity of the relationships remembered without increasing the number of bits needed. The generalized Bloom filter technique allows the TX field to occupy an arbitrary number of bits, and can be optimized for the expected workload.

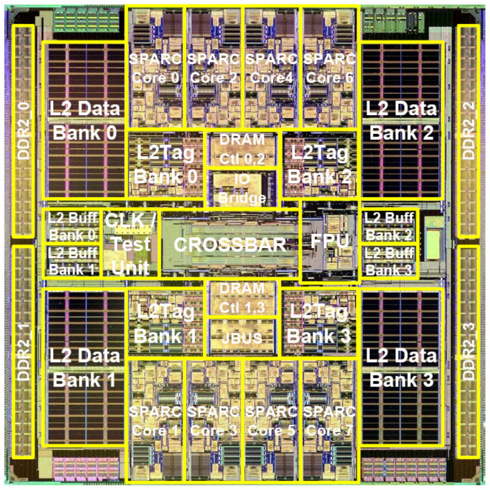


Fig. 5. The UltraSPARC T1 Die.

For example, suppose we would like to determine the branches that target a given line at the granularity of an individual instruction. Additionally, suppose our TH-IC contains 8 lines of 4 instructions each. The TI configuration described earlier would require 32 bits per line, for a total of 256 bits. If instead we choose to use a 12-bit Bloom filter per line with 4 hash functions, assuming no more than 2 branches target a given line gives a false positive rate of approximately 6.3%.

III. The OpenSPARC T1

The OpenSPARC T1 is a multicore, multithreading processor developed by Sun Microsystems. It consists of 8 identical cores, each of which is capable of interleaving the execution streams of up to 4 hardware threads simultaneously. The hardware threads all share the execution resources of the core they reside on, including the caches, TLBs, and functional units. The architectural state for each thread, such as the values of the register file, PC, and so on, is tracked independently, however. Other than the stages related to fetch, each stage can be occupied by at most one thread.

The architectural philosophy behind the T1 is to prefer thread level parallelism over instruction level parallelism at every opportunity. For example, the core has no branch predictor, and once the delay slot of a branch has been fetched, the thread will be stalled until the branch has fully resolved. In order to reduce the need for pipeline flushes, long latency operations such as branches are detected in the instruction cache line fill pipeline. A "long-latency instruction" bit is appended to each instruction prior to being stored in the cache. This

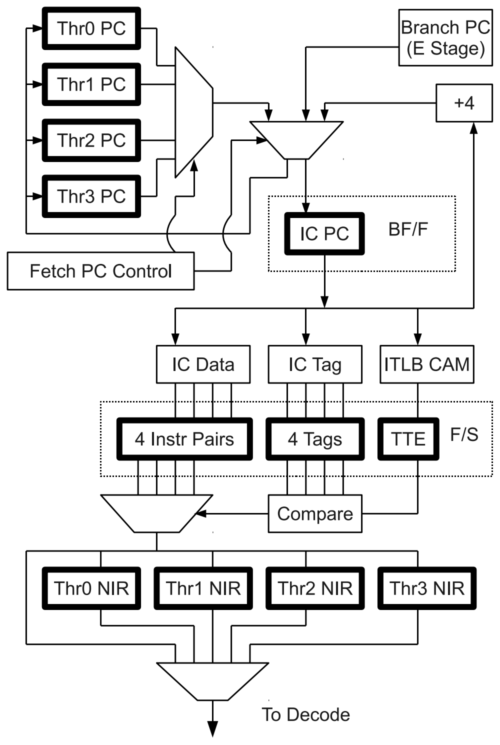


Fig. 6. OpenSPARC T1 Fetch Datapath. Registers are identified by blocks with heavy outlines. Registers that divide pipeline stages are group together and labeled with the previous and following stages. Some details are necessarily omitted.

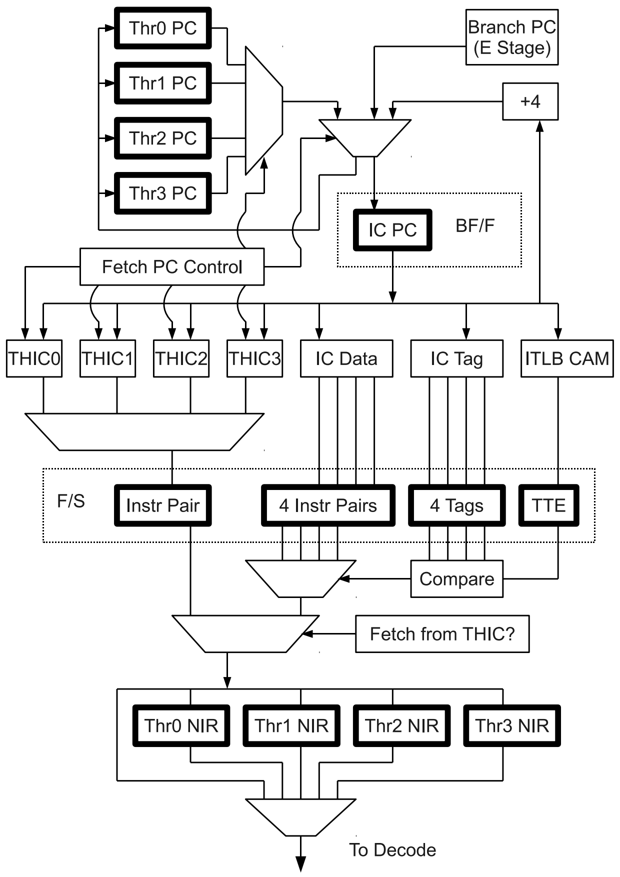


Fig. 7. OpenSPARC T1 Fetch Datapath with THICs.

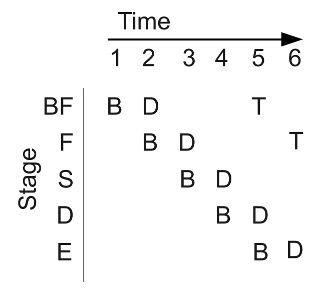


Fig. 8. T1 Branch Flow, Case 1 . **B** represents a branch instruction, **D** the delay slot instruction, and **T** the branch's target.

is what allows to quickly set aside a thread that will have to stall, even before the instruction has been decoded.

The T1 fetch unit operates across 4 pipeline stages: Before Fetch (BF), Fetch (F), Switch (S), and Decode (D). The BF stage is responsible for tracking each thread's fetch address. Instruction data is fetched from the cache during the F stage. The next thread to be executed by the remainder of the pipeline is selected in the S stage. Finally, instructions are decoded and the register file is read in the D stage.

In the BF stage, the next PC to be fetched for each thread is selected from either the sequentially following PC, the branch PC from the excution unit, or the trap PC provided by the trap logic unit (TLU). Of these PCs, one is selected to actually be fetched, and flopped into a register at the end of the stage.

Figure 6 illustrates the OpenSPARC T1 fetch datapath. In the F stage, the PC previously selected by the BF stage is sent to the instruction cache data array, tag array, valid bit array, and the TLB. The L1 instruction cache SRAM is 4-way set associative, with 128 entries of 8 instructions per way (plus parity), for a total of 17 kilobytes of instruction data. The tag SRAM holds 128 entries of 4 ways of 33-bit tags, for a total of 2112 bytes. The 64-entry, fully associative TLB is accessed in parallel to the cache, in order to provide the physical address needed for the tag match. All 4 ways of data and tags are pulled from the arrays, so that the tag match can proceed after the fetch has completed. Each fetch cycle, up to 2 sequential instructions are fetched from a line. When two instruction are fetched, the second instruction is always at an odd address, and is buffered in a "next instruction register" in the S stage.

Due to the intricacies in inverleaving multiple threads of execution within a single pipeline, the way that branch instructions are handled bears discussion. Since there are

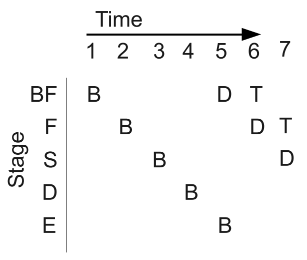


Fig. 9. T1 Branch Flow, Case 2

an unmanageable number of combinations of instructions from different threads in the pipeline, we will only discuss two cases, as shown in Figures 8 and 9.

The first case shows the flow of a branch instruction when no other threads are ready to run. This will only occur when either there is only one thread active, or when the other threads are stalled due, e.g., to an IC miss. In this case, the delay slot instruction is fetched the cycle after the branch instruction is fetched. In the same cycle the delay slot is fetched, the branch has completely been fetched, and the pipeline detects that it is a long latency operation. At that point the thread is put in a waiting state, and after the delay slot, no further fetches (for that thread) will be initiated until the wait is cancelled. When the branch reaches D, the pipeline cancels the wait, and the thread can fetch again the following cycle. When the branch arrives at E, the branch target PC is ready, and can be sent to the IC immediately.

The second case shows the flow for a thread when all other threads fetch an instruction before the delay slot can be fetched. The pipeline's LRU mechanism ensures that a ready thread will wait no more than three cycles between issued instructions. The delay slot PC is stored in the IC PC register at the BF stage at the cycle the branch is in E. The target is written to the threads PC register for the following cycle.

In any case, ignoring exceptions, cache misses, and so on, every instruction that is fetched is eventually committed, regardless of the code's branching behavior.

IV. Integrating the TH-IC

The most significant design decision concerning this integration is whether to use a single TH-IC data structure per core, or to use a copy of the TH-IC for each thread. Prior work has shown that a direct-mapped TH-IC provides better power savings than an associative one. A shared, direct mapped TH-IC is likely to have a significant number of conflict misses, due to the likelyhood that the multiple fetch sequences traversing the cache will intersect at some point. Even if we were to use a shared, associative TH-IC, a single, large SRAM would be needed, which would work against our power savings goals. However, independent per-thread TH-ICs can be much smaller, and the logic can be made to operate similarly to that of a single-threaded core. It was thus decided to use a small, direct-mapped TH-IC for each thread. The resulting fetch architecture is shown in Figure 7.

Timing requirements are tightest in the path through the TLB. The L1 tags are available sooner than the TLB result, and so at the end of F, they are sent directly from the tag array to the TLB. The match is then performed at the beginning of S. Since the TH-IC uses a much smaller (and thus faster) SRAM than the L1-IC, we can use the first phase of S to send the data from the L1-IC to the TH-IC for writing, and perform the actual write in the second phase of S. When a TH-IC hit is guaranteed, the instruction comes directly from one of four small data arrays. Now instead of way selection, we need to perform thread selection. The thread is always known immediately, unlike the way, and so this will complete quickly. Thus the timing impact for this integration on existing critical paths is minimal.

The existing design was largely unmodified. Most of the changes made were in the form of additional logic. Logic was added to consolidate and sanitize the signals within the fetch control module that determine the type of fetch being initiated. Logic and registers were added to track whether each thread is fetching sequentially, or the target of a direct branch, or otherwise. Other logic was added to disable the L1-IC and ITLB when a guaranteed hit is detected. Additional modules were added for the global and per-thread TH-IC datapaths. The global TH-IC datapath is responsible for muxing the data read from the TH-IC, and for tracking the in flight fetch (partial) PC. The per-thread modules track the valid, NS, NT, TL and ID bits for each thread, as well as (part of) the PCs of previous fetches.

A few small operational changes in the TH-IC were required due to branch delay slots, which were not considered in earlier work. On a SPARC processor, when fetching the target of a branch, the branch itself must have been fetched at least two instructions prior to the target. (On the T1, the pipeline only allows the delay slot to be fetched, and no other instructions, so this is always exactly two.) This means we must track two previous instructions; one is not enough.

The T1 has a single cycle latency between the initiation of a fetch, and the availability of the fetched instruction. In certain cases, we cannot guarantee a hit when we otherwise might have. For example, suppose the target of a branch instruction is at the end of a line, and the line containing the target is not in the TH-IC, but whatever line actually *is* present has its NS bit set. Since the branch target access is a miss, it cannot be (and should not be) a guaranteed hit. Without the single cycle latency, we would have detected the miss soon enough

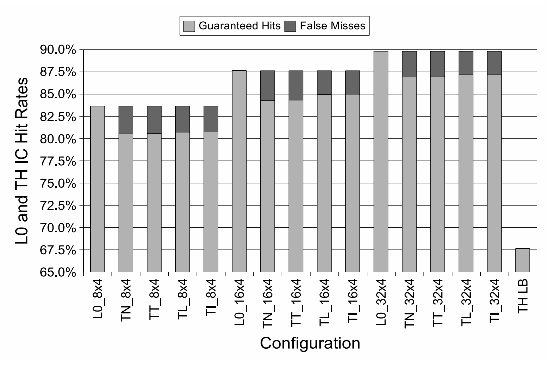


Fig. 10. TH-IC Guaranteed Hit Rate for Mibench Small Set. L0 indicates a standard L0 cache. TN indicates the *oblivious* configuration; TT indicates a single transfer bit; TL and TI indicate per line and per instruction transfer bits. TH-LB refers to the Tagless-Hit line buffer (a single-line TH-IC). NxM indicates N lines of M instructions.

to know we should ignore the NS bit when fetching the instruction after the target. If we add in the single cycle latency, however, we need to ignore the NS bit unless one of the following is true: the current thread has no instruction in the F stage; or the previous fetch was also sequential; or the previous fetch was a guaranteed hit. Since the latency for branches is longer than the fetch latency, no additional logic is needed for NT bits.

Invalidations of TH-IC data come from one of two places. The first is due to invalidations requests on the L1-IC. Whenever a line in the L1-IC is invalidated, it must also be invalidated in any TH-IC that holds it. The other source of invalidations is due to L1-IC fills from the L2 cache. When a miss on the L1-IC occurs, no lines are invalidated in either the L1-IC or the TH-IC. Instead, the core sets the thread aside until the miss fill is returned from the L2. After the returned line is written to the L1, any line that was overwritten gets removed from the TH-ICs for that core.

In the RTL for our design, we allowed for the selection of both an 8-line and 16-line TH-IC for each thread. Both configurations use a 4 byte line size, as opposed to the 8 byte line size in the T1 L1-IC. We chose to use the shorter line size, again in the interest of power savings, and also due to the difficulty in routing a 256-bit bus to 4 different places. Altering the pipeline to move the 256 bits via a 128-bit bus over 2 cycles is also likely to be difficult.

V. Results

Figure 10 shows the hit rate in the TH-IC for the "small" set of benchmarks running on a SimpleScalar single-threaded, out-of-order simulator that was extended to simulate the TH-IC. These results were collected in our ealier work on the TH-IC, and are included for comparison. Various configurations are shown, including

TABLE I
TH-IC Guaranteed Hit Rate on OpenSPARC T1

Benchmark	Threads	8 Lines	16 Lines
Quicksort	1	69.3%	72.5%
Quicksort	4	77.3%	80.2%

TABLE II
Fetch SRAM Power on OpenSPARC T1

	Dynamic power	Leakage power
L1-IC data:	3.10e-1 W	1.41e-3 W
L1-IC tags:	1.17e-2 W	1.48e-5 W
TH-IC data:	3.02e-3 W	1.88e-6 W

a standard L0 cache (with a traditional tag check), the "oblivious" configuration (TN), a single "transfer" bit (TT), the "transfer line" configuration (TL), and the "transfer instruction" configuration (TI). We would expect similar hit rates for a single thread executing on a T1 core. For multiple threads executing on the same core, we would expect the TH-IC guaranteed hit rate to scale with the L1-IC hit rate. (Suppose a single thread is running on a core, and has TH-IC hit rate of t and an L1-IC hit rate of n. When other threads are introduced to the core, the former thread's L1-IC hit rate may change to n'. We would expect the resulting TH-IC hit rate to change to tn'/n.) This is because every TH-IC hit is an L1-IC hit, and we don't expect conflicts between threads to erase vital metadata very often. The 4-way set associativity of the L1-IC effectively reduces the chance that destructive interference on the TH-IC metadata occurs among the 4 threads to zero.

We have run a limited set of benchmarks within a simulator compiled from the T1 Verilog source code. We tested both the original T1 design, and with two configurations of TH-ICs. The percent of accesses that were guaranteed hits in the TH-IC for the tested configurations are listed in Table I.

The data collected thus far are consistent with our expectations.

Cacti version 5.3 [9], [10] provides the estimates of power usage for the SRAMs used by the T1 instruction fetch architecture as shown in Table II. We do not provide performance data, as cycle counts were identical in all cases (as expected).

We used a 16-line THIC for these estimates. The data indicate power per SRAM instance, so the total power for all TH-IC data SRAMs is 4 times the value listed. So the approximate power for a standard L1-IC without TH-IC is 3.23e-1 W. Adding a 16 line TH-IC, and assuming a TH-IC hit rate of 75% gives 8.40e-2 W. Since these SRAMs tend to dominate instruction fetch power, this gives a fetch power savings of approximately 74%. Note that this is in a processor that was designed from the start to be included in a high-efficiency, high-throughput system, making liberal use of clock-gating and other power-efficient techniques. In addition, we have not measured the energy savings from eliminated ITLB accesses, as an adequate power model for that structure could not be obtained.

VI. Future Work

We have developed a set of extensions to TH-IC [4] which allow the branch predictor, BTB, RAS, and other branch related structures to be disabled for instructions known not to be branches. While the T1 does not use a branch predictor, integrating one may justify its power consumption, simply by improving single-threaded performance. Such a branch predictor may in fact be integrated with the TH-IC to take advantage of data it already stores.

VII. Conclusions

We have presented an implementation of our low-power Tagless-Hit Instruction Cache. The implementation shows that, even in designs that are power-concious, there is still room for improvement, especially in instruction fetch. Though these processors are designed to efficiently provide high throughput for enterprise tasks, with additional power savings such as those provided by the TH-IC, these processors may become suitable for high-throughput embedded tasks as well.

Our design also shows the ease with which are technique is adapted to modern CPU designs. We have found that the theoretical benefits predicted from in higher level simulations will bear out when implemented in silicon. Since power is a concern in nearly every modern processing application, and performance is always desirable, a technique such as the TH-IC, which reduces power without impacting performance, is justified in nearly every design.

VIII. Acknowledgements

This research was supported in part by NSF grants CNS-0615085 and CNS-0915926.

References

[1] S. Microsystems, "An Open Source Processor used in Sun SPARC Servers: OpenSPARC," February 2010, http://www.opensparc.net/.

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

[3] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf, "A 160-mhz, 32-b, 0.5-W CMOS RISC microprocessor," *Digital Tech. J.*, vol. 9, no. 1, pp. 49–62, 1997.

[4] S. Hines, Y. Peress, P. Gavin, D. Whalley, and G. Tyson, "Guaranteeing instruction fetch behavior with a lookahead instruction fetch engine (LIFE)," Submitted to the ACM/IEEE International Symposium on Microarchitecture, December 2008.

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

[6] Johnson Kin and Munish Gupta and William H. Mangione-Smith, "Filtering memory references to increase energy efficiency," *IEEE Transactions on Computers*, vol. 49, no. 1, pp. 1–15, 2000.

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

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

[9] S. J. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE Journal of Solid State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[10] H.-P. Corporation, "CACTI 5.3 (rev 174)," February 2010, http://quid.hpl.hp.com:9081/cacti/sram.y.