Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching

Eric Rotenberg Steve Bennett Jim Smith

April 11, 1996

Abstract

Superscalar processors require sufficient instruction fetch bandwidth to feed their highly parallel execution cores. Fetch bandwidth is determined by a number of factors, namely instruction cache hit rate, branch prediction accuracy, and taken branches in the instruction stream. Taken branches introduce the problem of *noncontiguous instruction fetching*: the dynamic instruction sequence exists in the cache, but the instructions are not in contiguous cache locations. This report considers the problem of fetching noncontiguous blocks of instructions in a single cycle.

We propose the *trace cache*, a special instruction cache that captures dynamic instruction sequences. Each line in the trace cache stores a dynamic code sequence, which may contain one or more taken branches. Dynamic sequences are built up as the program executes. If a predicted dynamic sequence exists in the trace cache, it can be fed directly to the decoders.

We investigate other methods for fetching noncontiguous instruction sequences in a single cycle. The Branch Address Cache [2] and Collapsing Buffer [1] achieve high bandwidth by feeding multiple noncontiguous fetch addresses to an interleaved cache and performing complex alignment on the instructions as they come out of the cache. Inevitably, this approach lengthens the critical path through the instruction fetch unit. Extra stages in the fetch pipeline increase branch mispredict recovery time, decreasing overall performance. Our approach moves complexity due to noncontiguous instruction fetching off the critical path and onto the fill side of the trace cache.

We compare the performance of the trace cache against other fetch designs. We first consider simple instruction fetching mechanisms that predict only one branch at a time or fetch only up to the first taken branch. We also consider more aggressive methods that are able to fetch beyond multiple taken branches. For integer benchmarks, the trace cache improves performance on average by 34% over the fetch unit limited to one basic block per cycle, and 17% over the fetch unit limited to multiple contiguous basic blocks. The corresponding improvements for floating point benchmarks are 16% and 9%. Further, the trace cache consistently performs better than the other high bandwidth fetch mechanisms studied *even if single-cycle fetch latency is assumed across all mechanisms*. Simulations with more realistic latencies for the other high bandwidth approaches, based on pipeline stages before and after the instruction cache, show that the trace cache clearly outperforms other approaches: on average, 20% and 10% better than the next highest performer for integer and floating point benchmarks, respectively.

1 Introduction

High performance superscalar processor organizations divide naturally into an instruction fetch mechanism and an instruction execution mechanism (Figure 1). The fetch and execution mechanisms are separated by instruction issue buffer(s), for example queues, reservation stations, etc. Conceptually, the instruction fetch mechanism acts as a “producer” which fetches, decodes, and places instructions into the buffer. The instruction execution engine is the “consumer” which removes instructions from the buffer and executes them, subject to data dependence and resource constraints. Control dependences (branches and jumps) provide a feedback mechanism between the producer and consumer.

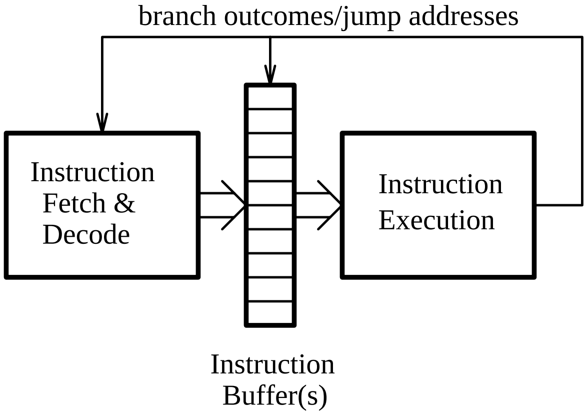


Figure 1: Instruction fetch and execute mechanisms separated by instruction buffers.

Processors having this organization employ aggressive techniques to exploit instruction-level parallelism [19]. Wide dispatch and issue paths place an upper bound on peak instruction throughput. Large issue buffers are used to maintain a *window* of instructions necessary for detecting parallelism, and a large pool of physical registers provides destinations for all of the in-flight instructions issued from the window. To enable concurrent execution of instructions, the execution engine is composed of many parallel functional units. The fetch engine speculates past multiple branches in order to supply a continuous instruction stream to the window.

The trend in superscalar design is to scale these techniques: wider dispatch/issue, larger windows, more physical registers, more functional units, and deeper speculation. To maintain this trend, it is important to balance all parts of the processor – any bottlenecks diminish the benefit of aggressive ILP techniques.

In this paper, we are concerned with instruction fetch bandwidth becoming a performance bottleneck. Instruction fetch performance depends on a number of factors. Instruction cache hit rate and branch prediction accuracy have long been recognized as important problems in fetch performance and are well-researched areas. In this paper, we are interested in additional factors that are only now emerging as processor issue rates exceed four instructions per cycle:

- *branch throughput* – If only one conditional branch is predicted per cycle, then the window can grow at the rate of only one basic block per cycle. Predicting multiple branches per cycle allows the overall instruction throughput to be correspondingly higher.

- *noncontiguous instruction alignment* – Because of branches and jumps, instructions to be fetched during any given cycle may not be in contiguous cache locations. Hence, there must be adequate paths and logic available to fetch and align noncontiguous basic blocks and pass them up the pipeline. That is, it is not enough for the instructions to be present in the cache, it must also be possible to access them in parallel.

- *fetch unit latency* – Pipeline latency has a profound impact on processor performance. This is due to the cost of refilling the pipeline after incorrect control speculation. In the case of the fetch unit, we are concerned with the startup cost of redirecting fetching after resolving a branch mispredict, jump, or instruction cache miss. Inevitably, the need for higher branch throughput and noncontiguous instruction alignment will increase fetch unit latency; yet ways must be found to minimize the latency impact.

Table 1 illustrates why branch throughput and noncontiguous instruction alignment need to be considered. Current fetch units are limited to one branch prediction per cycle, and can therefore fetch 1 basic block per cycle or up to the maximum instruction fetch width, whichever comes first. The data shows that the average size of basic blocks is around 4 or 5 instructions for integer codes. While fetching a single basic block each cycle is sufficient for implementations that issue at most 4 instructions per cycle, it is not so for processors with higher peak issue rates. If we introduce multiple branch prediction [2][1], then the fetch unit can at least fetch multiple *contiguous* basic blocks in a cycle. Data for the number of instructions between taken branches shows that the upper bound on fetch bandwidth is still somewhat limited in this case, due to the frequency of taken branches. Therefore, if a taken branch is encountered, it is necessary to fetch instructions down the taken path in the same cycle that the branch is fetched.

Benchmark	taken %	avg basic block size	avg sequential run size
eqntott	86.2%	4.20	4.87
espresso	63.8%	4.24	6.65
xlisp	64.7%	4.34	6.70
gcc	67.6%	4.65	6.88
sc	70.2%	4.71	6.71
compress	60.9%	5.39	8.85

Table 1: Percentage of all dynamic branches that are taken, average basic block size, and average length of a sequential run of instructions (i.e. number of instructions between taken branches).

1.1 The Trace Cache

The job of the fetch unit is to feed the dynamic instruction stream to the decoder. A problem is that instructions are placed in the cache in their compiled order. Storing programs in static form favors fetching code that does not branch or code with large basic blocks. Neither of these cases is typical of integer code.

We propose a special instruction cache which captures dynamic instruction sequences. This structure is called a *trace cache* because each line stores a snapshot, or trace, of the dynamic instruction stream, as shown in Figure 2. A trace is a sequence of at most n instructions and at most m basic blocks starting at any point in the dynamic instruction stream. The limit n is the trace cache line size, and m is the branch predictor throughput. A trace is fully specified by a starting address and a sequence of up to $m-1$ branch outcomes which describe the path followed.

The first time a trace is encountered, it is allocated a line in the trace cache. The line is filled as instructions are fetched from the instruction cache. If the same trace is encountered again in the course of executing the program, i.e. the same starting address and predicted branch outcomes, it will be available in the trace cache and is fed directly to the decoder. Otherwise, fetching proceeds normally from the instruction cache.

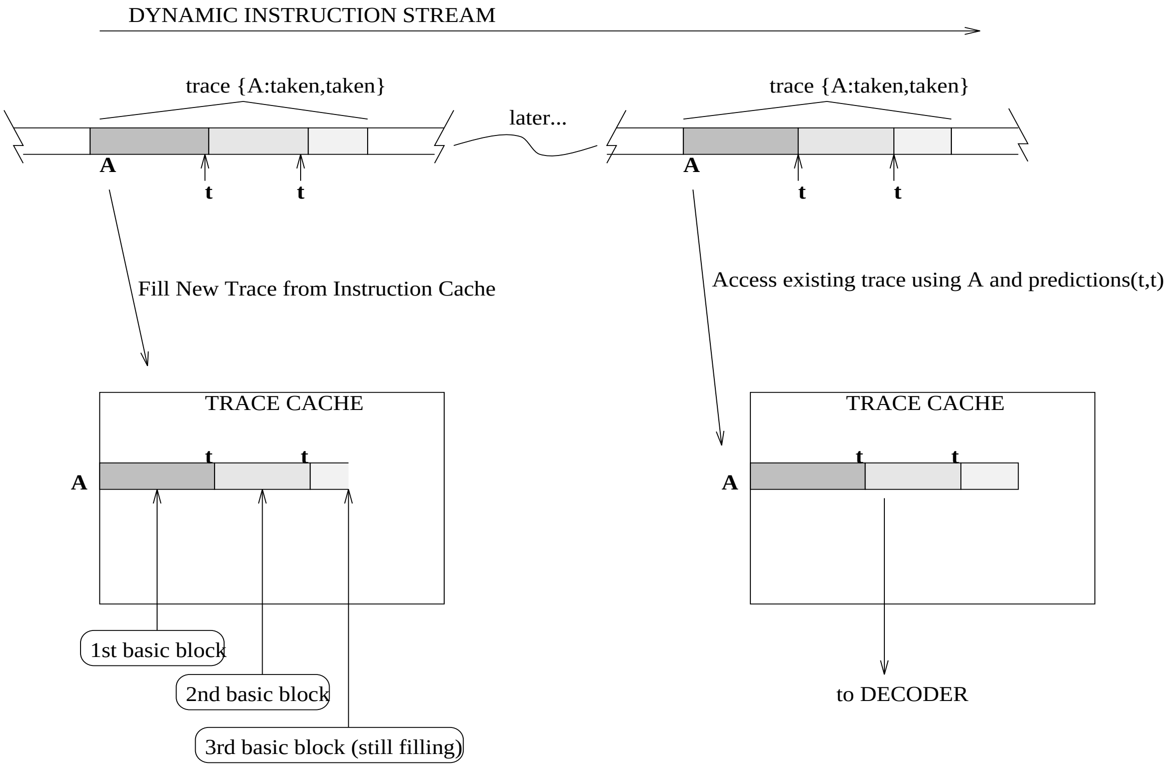

DYNAMIC INSTRUCTION STREAM
trace {A:taken,taken}
later...
trace {A:taken,taken}
A
t
t
A
t
t
Fill New Trace from Instruction Cache
Access existing trace using A and predictions(t,t)
TRACE CACHE
t
t
A
TRACE CACHE
t
t
A
1st basic block
2nd basic block
3rd basic block (still filling)
to DECODER

Figure 2: High level view of the trace cache approach. The instruction trace starting at address A and continuing past 2 taken (t) branches is filled into the trace cache at the first encounter and later fetched in a single cycle. The address A and 2 branch predictions are used to lookup the trace in the trace cache.

The trace cache approach relies on dynamic sequences of code being reused. This may be the case for two reasons:

- *temporal locality* – like the primary instruction cache, the trace cache can count on instructions which have been recently used being used again in the near future.
- *branch behavior* – most branches tend to be biased towards one direction, which is why branch prediction accuracy is usually high. Thus, it is likely that certain paths through the control flow graph will be followed frequently.

1.2 Related Prior Work

Three recent studies have focused on high bandwidth instruction fetching and are closely related to the research reported here. All of these attempt to fetch multiple, possibly noncontiguous basic blocks each cycle from the instruction cache.

First, Yeh, Marr, and Patt [2] consider a fetch mechanism that provides high bandwidth by predicting multiple branch target addresses every cycle. The method features a *Branch Address*

Cache, a natural extension of the branch target buffer [5]. With a branch target buffer, a single branch prediction and a BTB hit produces the starting address of the next basic block. Similarly, a hit in the branch address cache combined with multiple branch predictions produces the starting addresses of the next *several* basic blocks. These addresses are fed into a highly interleaved instruction cache to fetch multiple basic blocks in a single cycle.

A second study by Franklin and Dutta [4] uses a similar approach to the branch address cache (providing multiple branch targets), but with a new method for predicting multiple branches in a single cycle. Their approach hides multiple individual branch predictions within a single prediction; e.g. rather than make 2 branch predictions, make 1 prediction that selects from among 4 paths. This enables the use of more accurate two-level predictors.

Another hardware scheme proposed by Conte, Mills, Menezes, and Patel [1] uses two passes through an interleaved branch target buffer. Each pass through the branch target buffer produces a fetch address, allowing two nonadjacent cache lines to be fetched. In addition, the interleaved branch target buffer enables detection of any number of branches in a cache line. In particular, the design is able to detect short forward branches within a line and eliminate instructions between the branch and its target using a *collapsing buffer*. The work also proposes compiler techniques to reduce the frequency of taken branches.

Two previously proposed hardware structures are similar to the trace cache but exist in different applications. The fill unit, proposed by Melvin, Shebanow and Patt [17], caches RISC-like instructions which are derived from a CISC instruction stream. This predecoding eased the problem of supporting a complex instruction set such as VAX on the HPS restricted dataflow engine. Franklin and Smotherman [4] extended the fill unit's role to dynamically assemble VLIW-like instruction words from a RISC instruction stream, which are then stored in a *shadow cache*. The goal of this structure is to ease the dependency checking and issue complexity of a wide issue processor.

1.3 Problems with Other Fetch Mechanisms

Recall that the job of the fetch unit is to feed the dynamic instruction stream to the decoder. Unlike the trace cache approach, previous designs have only the conventional instruction cache, containing a static form of the program, to work with. Every cycle, instructions from noncontiguous locations must be fetched from the instruction cache and assembled into the predicted dynamic sequence. There are problems with this approach:

- Pointers to all of the noncontiguous instruction blocks must be generated before fetching can begin. This implies a level of indirection, through some form of branch target table (branch target buffer, branch address cache, etc.), which translates into an additional pipeline stage before the instruction cache.

- The instruction cache must support simultaneous access to multiple, noncontiguous cache lines. This forces the cache to be multiported; if multiporting is done through interleaving, bank conflicts are suffered.

- After fetching the noncontiguous instructions from the cache, they must be assembled into the dynamic sequence. Instructions must be shifted and aligned to make them appear contiguous to the decoder. This most likely translates into an additional pipeline stage after the instruction cache.

The trace cache approach avoids these problems by caching dynamic instruction sequences themselves, ready for the decoder. If the predicted dynamic sequence exists in the trace cache, it does not have to be recreated on the fly from the instruction cache's static representation.

In particular, no additional stages before or after the instruction cache are needed for fetching noncontiguous instructions. The stages do exist, but not on the critical path of the fetch unit – rather, on the fill side of the trace cache. The cost of this approach is redundant instruction storage: the same instructions may reside in both the primary cache and the trace cache, and there even might be redundancy among lines in the trace cache.

1.4 Contributions

As with prior work in high bandwidth instruction fetching, this report demonstrates the importance of fetching past multiple possibly-taken branches each cycle. Unlike other work in the area, we place equal emphasis on fetch unit latency. The end result is the trace cache as a means for low latency, high bandwidth instruction fetching.

Another contribution is a detailed simulation study comparing proposed high bandwidth fetch mechanisms including the trace cache. Previously, the approaches described in Section 1.2 could not be directly compared due to different experimental setups – different ISAs, processor execution models, branch predictors, caches, workloads, and metrics.

In the course of this work, many microarchitectural and logic design issues arose. We looked at design issues for not only the trace cache, but other proposed mechanisms as well. The results of this detailed study are documented in the Appendix of this report.

1.5 Paper Overview

In the next section the trace cache fetch unit is described in detail. Section 3 follows up with an analysis of other proposed high bandwidth fetch mechanisms. In Section 4 we describe the simulation methodology including the processor model, workload, and performance metric. Simulation results are presented in Section 5. As part of the study in Section 5, we compare the trace cache with previously proposed high performance fetch mechanisms on a “level playing field” in terms of caches, branch prediction methods, execution engines, and performance measures.

2 Trace Cache

In Section 1.1 we introduced the concept of the trace cache – an instruction cache which captures dynamic instruction sequences. We now present a trace cache implementation. Because the trace cache is not intended to replace the conventional instruction cache or the fetch hardware around it, we begin with a description of the core fetch mechanism. We then show how the core fetch unit is augmented with the trace cache.

2.1 Core Fetch Unit

The core fetch unit is implemented using established hardware schemes. It is called *interleaved sequential* in [1]. Fetching up to the first predicted taken branch each cycle can be done using the combination of an accurate multiple branch predictor [2], an interleaved branch target buffer (BTB) [1][5], a return address stack (RAS) [13], and a 2-way interleaved instruction cache [1][7]. Refer to Figure 3.

The core fetch unit is designed to fetch as many contiguous instructions possible, up to a maximum instruction limit and a maximum branch limit. The instruction constraint is imposed by the width of the datapath, and the branch constraint is imposed by the branch predictor throughput. For demonstration, a fetch limit of 16 instructions and 3 branches is used throughout.

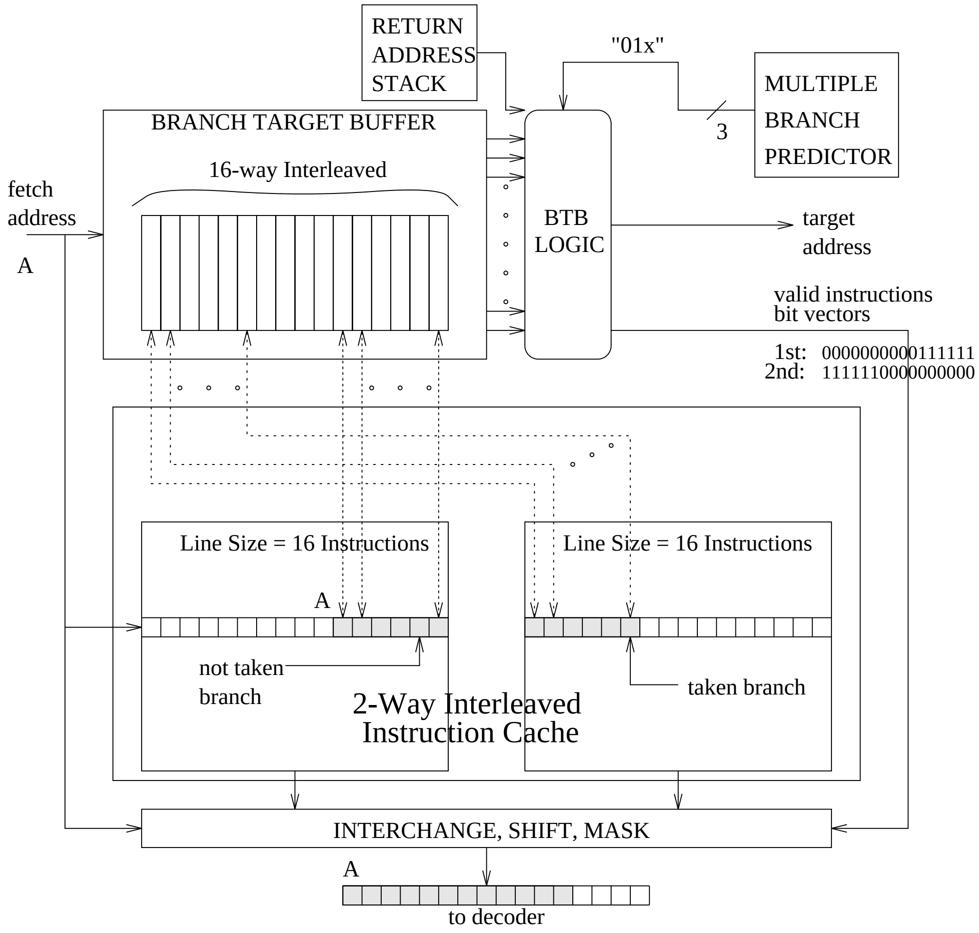


Figure 3: The core fetch unit.

The cache is interleaved so that 2 consecutive cache lines can be accessed; this allows fetching sequential code that spans a cache line boundary, always guaranteeing a full cache line or up to the first taken branch [7]. This scheme requires minimal complexity for aligning instructions: (1) logic to swap the order of the two cache lines (interchange switch), (2) a left-shifter to align the instructions into a 16-wide instruction latch, and (3) logic to mask off unused instructions.

All banks of the BTB are accessed in parallel with the instruction cache. They serve the role of (1) detecting branches in the instructions currently being fetched and (2) providing their target addresses, in time for the next fetch cycle. The BTB must be n-way interleaved, where n is the number of instructions in a cache line. This is so that all instructions within a cache line can be checked for branches in parallel [1]. The BTB can detect other types of control transfer instructions as well. If a jump is detected, the jump address may be predicted. (Jump target predictions are not considered in this paper, however.) Return addresses can almost always be obtained with no penalty by using a call/return stack. If the BTB detects a return in the instructions being fetched, it pops the address at the top of the RAS.

Notice in Figure 3 that the branch predictor is separate from the BTB. This is to allow for predictors that are more accurate than the 1-bit or 2-bit counters normally stored with each branch entry in the BTB. While storing counters with each branch achieves multiple branch prediction trivially, branch prediction accuracy is limited. Branch prediction is fundamental to ILP, and should have precedence over other factors. For high branch prediction accuracy, we use a 4kB GAg(14) correlated branch predictor [10]. The 14 bit global branch history register indexes into a single pattern history table. This predictor was chosen for its accuracy and because it is more easily extended to multiple branch predictions than other predictors which require address information [2][4]. It is relatively straightforward to extend the single correlated branch predictor to multiple predictions each cycle, as proposed in [2]. An actual hardware implementation is shown in Figure 4.

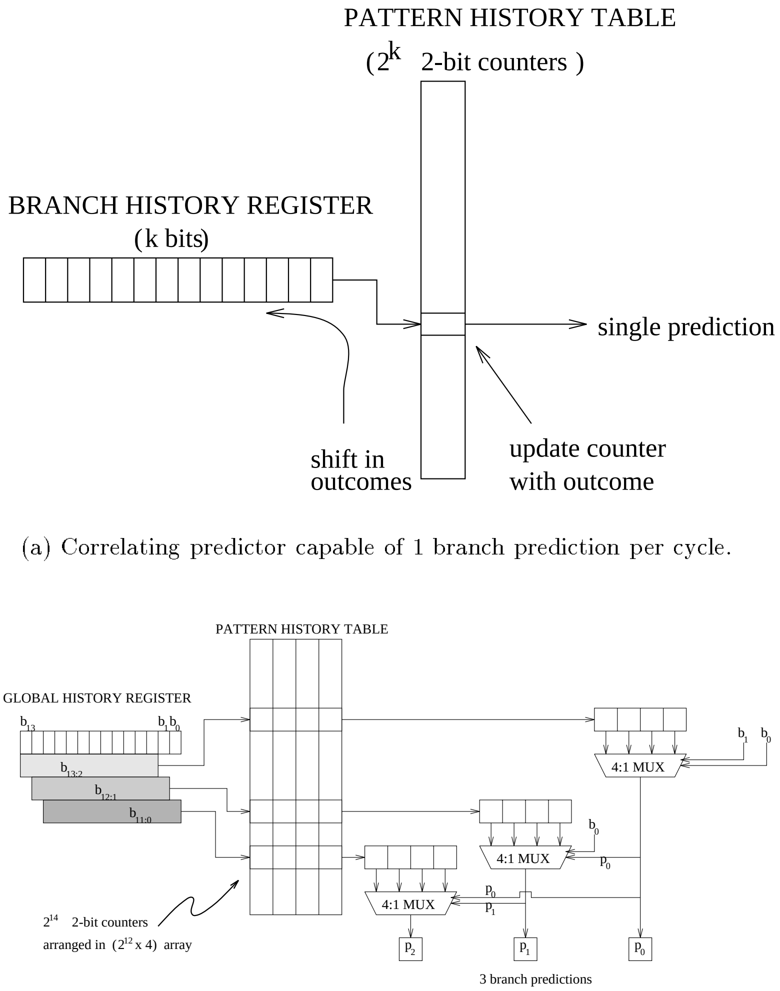
BTB logic combines the BTB hit information with the branch predictions to produce the next fetch address, and to generate trailing zeroes in the *valid instruction bit vectors* (if there is a predicted taken branch). The leading zeroes in the valid instruction bit vectors are determined by the low-order bits of the current fetch address. The masking logic is controlled by these bit vectors.

Both the interchange and shift logic are controlled by the low-order bits of the current fetch address. *This is a key point: the left-shift amount is known at the beginning of the fetch cycle, and has the entire cache access to fanout to the shifter datapath.* Further, if a transmission gate barrel shifter is used, instructions pass through only one transmission gate delay with a worst case capacitive loading of 15 other transmission gates on both input and output. In summary, control is not on the critical path, and datapath delay is minimal. Therefore, in our simulations we treat the core fetch unit as a single pipeline stage.

2.2 Adding the Trace Cache

The core fetch unit can only fetch contiguous sequences of instructions, i.e. it cannot fetch past a taken branch in the same cycle that the branch is fetched. The trace cache provides this additional capability. The trace cache together with the core fetch unit is shown in Figure 5.

The trace cache is made up of instruction traces, control information, and line-fill buffer logic. The length of a trace is limited in two ways – by number of instructions n and by number of basic blocks m. The former limit n is chosen based on the peak dispatch rate. The latter limit m is chosen based on n and the average number of instructions in a basic block. m also determines, or is constrained by, the number of branch predictions made per cycle. In Figure 5, $n = 16$ and $m = 3$. The control information is similar to the tag array of standard caches but contains additional state information:



(a) Correlating predictor capable of 1 branch prediction per cycle.

(b) Correlating predictor capable of 3 branch predictions per cycle.

Figure 4: Extending the predictor throughput.

- *valid bit*: indicates this is a valid trace.

- *tag*: the tag field identifies the starting address of the trace.

- *branch flags*: there is a single bit for each branch within the trace to indicate the path followed after the branch (taken/not taken). The m^{th} branch of the trace does not need a flag since no instructions follow it, hence there are only $m - 1$ bits instead of m bits.

- *branch mask*: state is needed to indicate (1) the number of branches in the trace and (2) whether or not the trace ends in a branch. This is needed for comparing the correct number of branch predictions against the same number of branch flags, when checking for a trace hit. This is also needed by the branch predictor to know how many predictions were used. The first $\lceil log_2(m+1) \rceil$ bits encode the number of branches. One more bit indicates if the last instruction in the trace is a branch; if true, the branch's corresponding branch flag does not need to be checked since no instructions follow it.

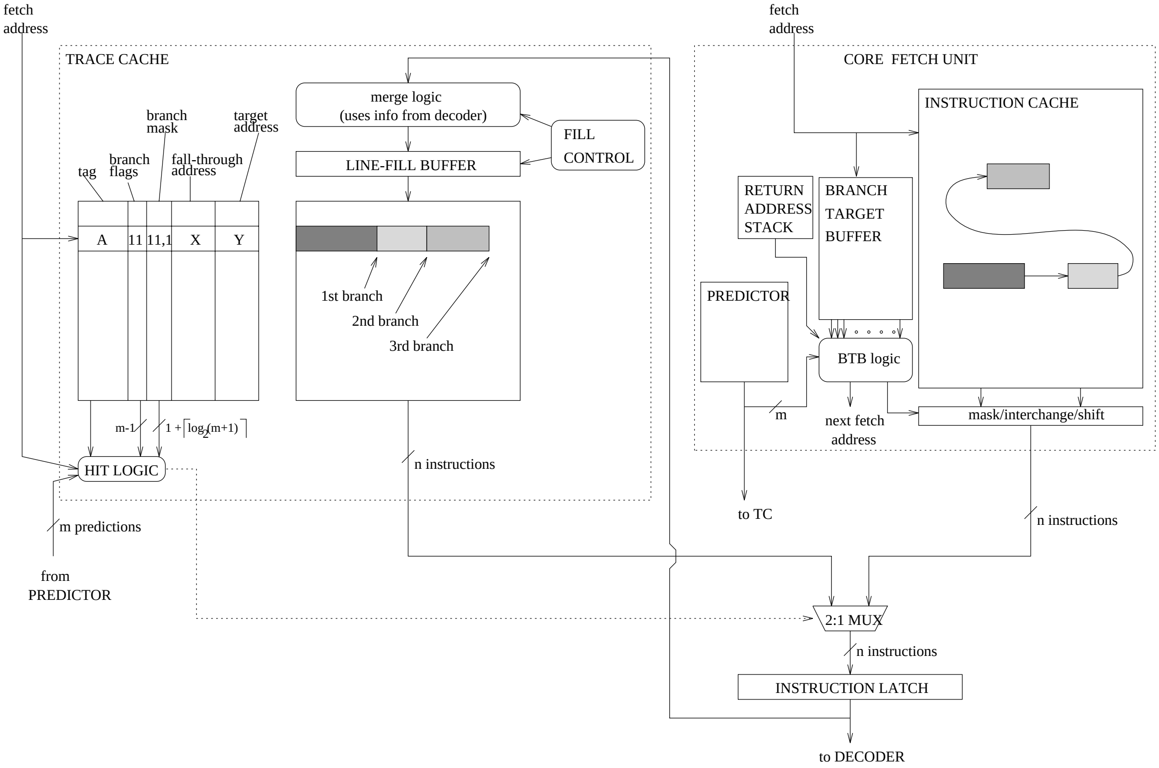


Figure 5: The trace cache fetch mechanism.

- *trace fall-through address*: if the last branch in the trace is predicted not taken, this address is used as the next fetch address.
- *trace target address*: if the last branch in the trace is predicted taken, this address is used as the next fetch address.

The trace cache is accessed in parallel with the instruction cache and BTB using the current fetch address. The predictor generates multiple branch predictions while the caches are accessed. The fetch address is used together with the multiple branch predictions to determine if the trace read from the trace cache matches the predicted sequence of basic blocks. Specifically, a trace cache hit requires that (1) the fetch address match the tag and (2) the branch predictions match the branch flags. The branch mask ensures that the correct number of prediction bits are used in the comparison. On a trace cache hit, an entire trace of instructions is fed into the instruction latch, bypassing the instruction cache.

On a trace cache miss, fetching proceeds normally from the instruction cache, i.e. contiguous instruction fetching. The line-fill buffer logic services trace cache misses. In the example in Figure 5, three basic blocks are fetched one at a time from the instruction cache, since all branches are predicted taken. The basic blocks are latched one at a time into the line-fill buffer; the line-fill control logic serves to merge each incoming block of instructions with preceding instructions in the line-fill buffer. Filling is complete when either n instructions have been traced or m branches have been detected in the trace. At this point the contents of the line-fill buffer are written into the trace cache. The branch flags and branch mask are generated during the line-fill process, and the trace target and fall-through addresses are computed at the end of the line-fill. If the trace does not end in a branch, the target address is set equal to the fall-through address.

There are different classes of control transfer instructions – conditional branches, unconditional branches, calls or direct jumps, returns, indirect jumps, and traps – yet so far only conditional branches have been discussed. The complex alternative for handling all of these cases is to add additional bits to each branch flag to distinguish the type of control transfer instruction. Further, the line-fill buffer must stop filling a trace when a return, indirect jump, or trap is encountered, because these control transfer instructions have an indeterminite number of targets, whereas the predictor can only predict one of two targets. Lastly, the branch mask and the hit logic are made slightly more complex since unconditional branches and calls should not be involved in prediction (the outcome is known).

We simplify these complications in two ways. First, the trace cache does not store returns, indirect jumps, or traps at all; the line-fill buffer aborts a fill when it detects any of these instructions. Second, unconditional branches and calls can be viewed as conditional branches that are extremely predictable; from this point of view, they can be grouped into the conditional branch class and not be treated any differently. With these two simplifications, the trace cache has only to deal with conditional branches.

The size of a direct mapped trace cache with 64 lines, $n = 16$, and $m = 3$ is 712 bytes for tags/control and 4 kilobytes for instructions (comparable in area to the correlated branch predictor, 4kB). This configuration is used in the experiments which follow.

2.3 Trace Cache Design Space

The trace cache depicted in Figure 5 is the simplest design among many alternatives. It is the implementation used in simulations of the trace cache. However, the design space deserves some attention:

- *associativity*: The simplest trace cache is direct mapped. However, it can be made more associative to reduce conflict misses. This will be at the expense of access time and replacement complexity.

- *multiple paths*: A downside of the simple trace cache is that from a given starting address, only one trace can be stored. It might be advantageous to be able to store multiple paths eminating from a given address. This can be thought of as another form of associativity – *path associativity*. Adding path associativity could reduce thrashing between traces that start at the same address.

- *partial matches*: An alternative to providing path associativity is to allow *partial hits*. If the fetch address matches the starting address of a trace and the first few branch predictions match the first few branch flags, provide only a prefix of the trace. This is in place of the simple "all or nothing" approach we use. The additional cost of this scheme is that intermediate basic block addresses must be stored for the same reason that trace target and fall-through addresses are stored. Also, a new issue is introduced: there is the question of whether or not a partial hit be treated as a miss.

- *other indexing methods*: The simple trace cache indexes with the fetch address and includes branch predictions in the tag match. Alternatively, the index into the trace cache could be derived by concatenating the fetch address with the branch prediction bits. This effectively achieves path associativity while keeping a direct mapped structure, because different paths starting at the same address now map to consecutive locations in the trace cache. There is a severe complication, however, because the number of branches in a trace could be less than

$m-1$. This implies the index should not use all branch prediction bits to access the trace cache, yet this is not known in advance. A possible solution is to write the same trace to two or more consecutive locations, so that either location gives a hit.

- *fill issues*: While the line-fill buffer is collecting a new trace, the trace cache continues to be accessed by the fetch unit. This means a miss could occur in the midst of handling a previous miss. The design options in order of increasing complexity are: ignore any new misses, delay servicing new misses until the line-fill buffer is free, or provide multiple line-fill buffers to support concurrent misses. Another issue is whether to fill the trace cache with speculative traces or to wait for branch outcomes before committing a trace to the cache.

- *judicious trace selection*: There are likely to be traces that are committed but never reused. These traces may displace useful traces, causing needless misses. To improve trace cache hit rates, the design could use a small buffer to store recent traces; a trace in this buffer is only committed to the trace cache after one or more hits to that trace.

- *victim trace cache*: An alternative to judicious trace selection is to use a victim cache [20]. A victim trace cache may keep valuable traces from being permanently displaced by useless traces.

3 Other High Bandwidth Fetch Mechanisms

In this section we analyze the organization of two previously proposed fetch mechanisms aimed at fetching and aligning multiple noncontiguous basic blocks each cycle. The analysis compares these mechanisms against the trace cache, with latency being the key point for comparison.

3.1 Branch Address Cache

The branch address cache fetch mechanism proposed by Yeh, Marr, and Patt [2] is shown in Figure 7. There are four primary components: (1) a branch address cache (BAC), (2) a multiple branch predictor, (3) an interleaved instruction cache, and (4) an interchange and alignment network. The BAC extends the BTB to multiple branches by storing a tree of target and fall-through addresses as depicted in Figure 6. The depth of the tree depends on the number of branches predicted per cycle.

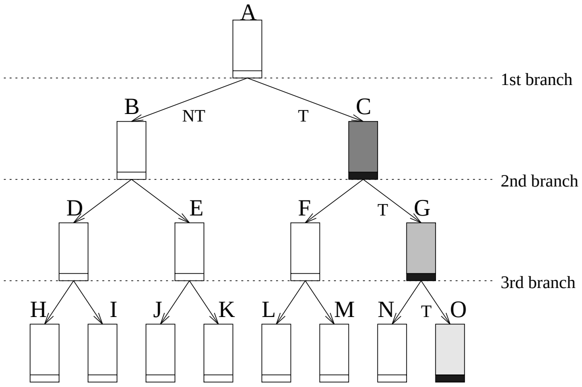


Figure 6: Each BAC entry stores a portion of the control flow graph.

In Figure 7, light grey boxes represent non-control transfer instructions and dark grey boxes represent branches; the fields in the BAC correspond to the tree in Figure 6, as indicated by the address labels A through O. The diagram depicts the two-stage nature of the design. In the first stage, an entry containing up to 14 basic block addresses is read from the BAC. From these addresses, up to 3 basic block addresses corresponding to the predicted path are selected. In this example, the next 3 branches are all predicted taken, corresponding to the sequence of basic blocks {C,G,O}. In the second stage, the instruction cache reads the three basic blocks indicated by addresses from the BAC in parallel from its multiple banks. Since the basic blocks may be placed arbitrarily into the cache banks, they must pass through an alignment network to align them into dynamic program order and merge them into the instruction latch.

The two stages in this design are pipelined. During the second stage, while basic blocks {C,G,O} are being fetched from the instruction cache, the BAC begins a new cycle using address O as its index. In general, the last basic block address indexing into the instruction cache is also the index into the BAC.

If an address misses in the BAC, an entry is allocated for the portion of the control flow graph which begins at that address. Branch target and fall-through addresses are filled in the entry as paths through the tree are traversed; an entry may contain holes corresponding to branches which have not yet been encountered.

Though conceptually the design has two pipeline stages, possibly one or more additional pipeline stages are implied by having the complicated alignment network. The alignment network must (1) interchange the cache lines from numerous banks (with more than two banks, the permutations grow quickly), and (2) collapse the basic blocks together, eliminating unused intervening instructions. Though not discussed in [2], logic like the collapsing buffer [1] discussed in the next section will be needed to do this.

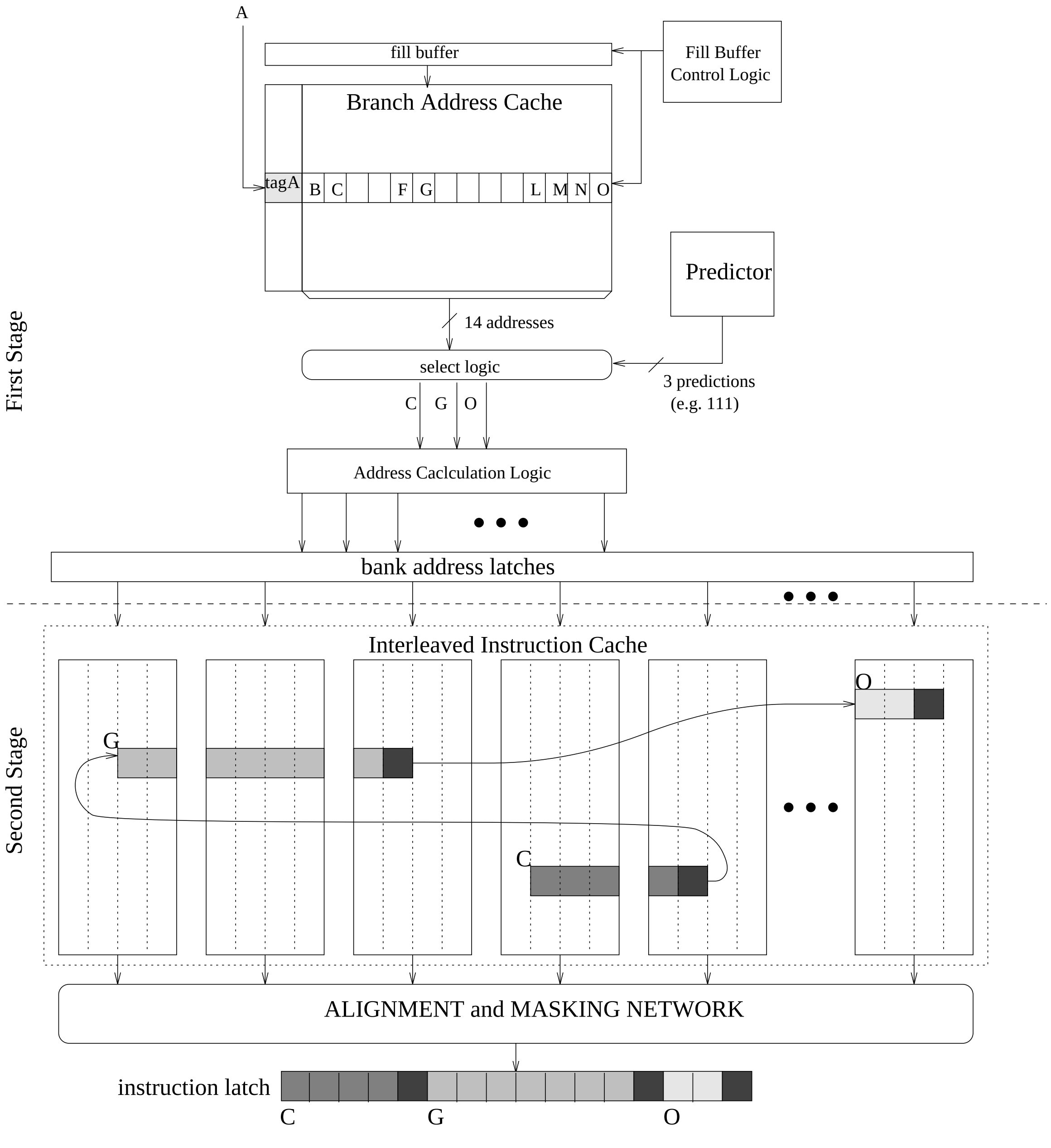


Figure 7: The BAC approach to instruction fetching.

3.2 Collapsing Buffer

The instruction fetch mechanism proposed by Conte, Mills, Menezes, Patel [1] is illustrated in Figure 8. It is composed of (1) an interleaved instruction cache, (2) an interleaved branch target buffer (BTB), (3) a multiple branch predictor, (4) special logic after the BTB, and (5) an interchange and alignment network featuring a *collapsing buffer*.

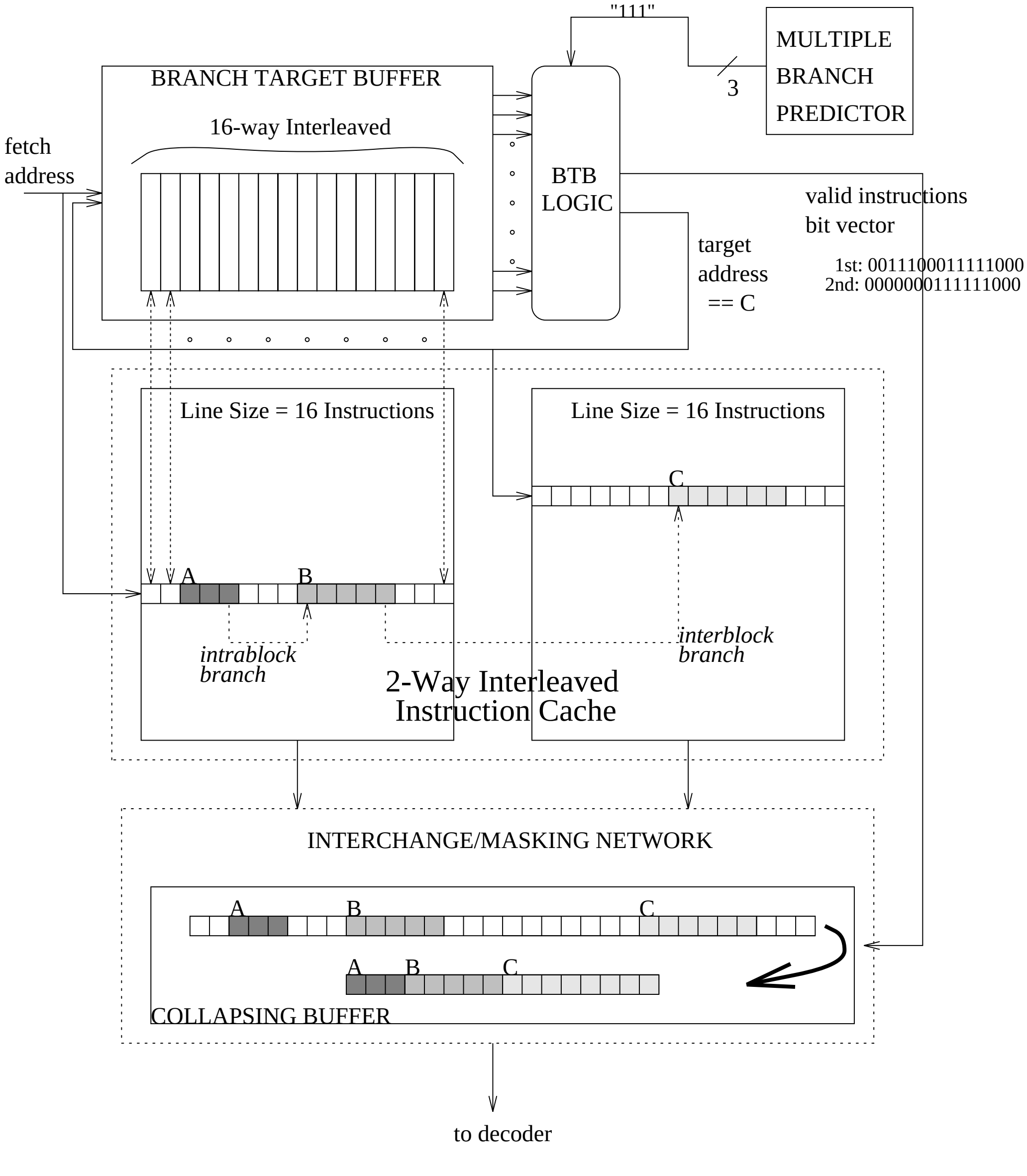


Figure 8: The instruction fetch mechanism proposed by Conte et al [1].

The hardware is similar to the core fetch unit of the trace cache (described in Section 3), but has two important distinctions. First, the BTB logic is capable of detecting *intrablock* branches – short hops within a cache line. Second, a single fetch goes through two BTB accesses. As will be described below, this allows fetching beyond one taken *interblock* branch – a branch out of the cache line. In both cases, the collapsing buffer uses control information generated by the BTB logic to merge noncontiguous basic blocks.

Figure 8 illustrates how three noncontiguous basic blocks labelled A, B, and C are fetched. The fetch address A accesses the interleaved BTB. The BTB indicates that there are two branches in the cache line, one at the instruction 5 with target address B, the other at the instruction 13 with target address C. Based on this branch information and branch predictions from the predictor, the BTB logic indicates which instructions in the fetched line are valid and produces the next basic block address, C.

The initial BTB lookup produces (1) a bit vector indicating the predicted valid instructions in the cache line (instructions from basic blocks A and B), and (2) the predicted target address C of basic block B. The fetch address A and target address C are then used to fetch two nonconsecutive cache lines from the interleaved instruction cache. This can be done only if the cache lines are in different banks. In parallel with this instruction cache access, the BTB is accessed again, using the target address C. This second, serialized lookup determines which instructions are valid in the second cache line and produces the next fetch address (the predicted successor of basic block C).

When the two cache lines have been read from the cache, they pass through masking and interchange logic and the collapsing buffer (which merges the instructions), all controlled by bit vectors produced by the two passes through the BTB. After this step, the properly ordered and merged instructions are captured in the instruction latches to be fed to the decoders.

This scheme has several disadvantages. First, the fetch line and successor line must reside in different cache banks. Bank conflicts can be reduced by adding more banks, but this requires a more complicated, higher latency interchange switch. Second, this scheme does not scale well for interblock branches; supporting additional interblock branches requires as many additional BTB accesses, all serialized. Third, the BTB logic requires a serial chain of n address comparators to detect intrablock branches, where n is the number of BTB banks. Most seriously, however, is that this fetch mechanism adds a significant amount of logic both before and after the instruction cache. The instruction fetch pipeline is likely to have three stages, as depicted in Figure 9: (1) initial BTB lookup and BTB logic, (2) instruction cache access and second BTB lookup, and (3) interchange switch, masking, and collapsing buffer. Because the BTB is used in both the first and second stages, a new fetch cycle cannot be initiated until the third stage unless the BTB is dual-ported and the BTB logic is duplicated.

The collapsing buffer takes only a single stage if implemented as a bus-based crossbar [1]. Note that the collapsing buffer is dependent on control information from the second BTB lookup, so control fanout logic from the third stage cannot be brought up into the second stage; i.e. there is truly a third stage.

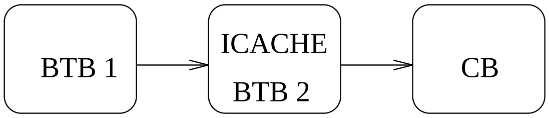


Figure 9: Stages in the CB approach.

4 Simulation Methodology

4.1 Processor Model

Our simulation model follows the basic structure shown in Figure 1 – a fetch engine and an execute engine decoupled via instruction issue buffers. Various fetch engines – trace cache, branch address cache, and collapsing buffer – are modeled in detail. The processor execution part of the model is constrained only by true data dependences. *We assume unlimited hardware resources – any instructions in the instruction buffers that have their data available may issue.* This is done to place as much demand on the fetch unit as any implementation will ever achieve, making instruction fetch the performance bottleneck whenever possible. In effect, unlimited register renaming and full dynamic instruction issue are assumed. Loads and stores are assumed to have oracle address disambiguation – loads and stores wait for previous stores only if there is a true address conflict. Also, the data cache always hits. The only hardware limitations imposed are the maximum size of the instruction buffer and the degree of superscalar dispatch. In all simulations, the size of the instruction buffer is 2048 useful instructions and the maximum fetch/dispatch bandwidth is 16 instructions per cycle. In summary, the amount of ILP exploited is limited by 5 factors:

- *maximum fetch/dispatch rate (16/cycle)*
- *maximum size of instruction window (2048)*
- *true data dependences in the program*
- *operation latencies*
- *performance of the fetch engine*

It is the last factor that we are interested in and which will vary between simulations.

The instruction pipeline is composed of 4 phases: fetch, dispatch, issue, and execution. The latency of the fetch phase is varied according to implementation, and the dispatch latency is fixed at 1 cycle. If all operands are available at or immediately after dispatch, instruction issue takes only 1 cycle; otherwise issue is delayed until operands arrive. Because of unlimited resources and unlimited register renaming, issue never stalls due to structural or register WAR/WAW hazards. After issue, execution takes a certain number of cycles based on the operation. Operation latencies are shown in Table 2. The minimum pipeline latency from fetch to completion is therefore 4 cycles, for a fetch unit latency of 1 cycle, no data dependence stalls, and a 1 cycle operation. The retire stage(s) of the pipeline are hidden due to unlimited result forwarding from the execute stage to the issue stage.

4.2 Workload

Six integer and six floating point benchmarks from the SPEC suite were used to evaluate the performance of the various fetch mechanisms. The benchmarks were compiled on a Sun SPARCstation 10/30 using "gcc -O4 -static -fschedule-insns -fschedule-insns2" for integer benchmarks and "f77 -O4 -Bstatic -fast -cg89" for floating point benchmarks. SPARC instruction traces were generated using *the Quick Profiler and Tracer* (QPT) [16] and then fed into the trace-driven processor simulator. Table 3 shows inputs for each of the benchmarks. Benchmarks were simulated for 100 million instructions.

The foremost problem with trace-driven simulation is that incorrect speculative execution cannot be simulated, since traces represent only the correct path of execution. This may lead to some

OPERATION LATENCIES (cycles)	
Integer ALU Operations	1
Loads	2†
Stores	1‡
Control Transfer Instructions	1
FP Add/Sub/Mult/Conv	3
FP Div	11/18
FP Sqrt	17/32
all other FP ops	1

† One cycle to compute address, one cycle to access the data cache.
‡ One cycle to compute address, the rest of the write is hidden; store data can always be bypassed to dependent loads when available.

Table 2: Operation execution latencies. Note that for loads and stores, data cache misses are not simulated. Floating point latencies for the most common operations are similar to the MIPS R10000 [15].

error. For example, cache structures do not see the pollution effects caused by fetching and executing instructions down the wrong path. Further, our execution engine does not see resource usage due to incorrectly speculated instructions (however, because of unlimited resource assumptions in the execution model, this latter factor will not produce performance much different from simulation of incorrect speculation).

BENCHMARK	input	BENCHMARK	input
eqntott	int_pri_3.eqn	doduc	doducin
espresso	bca.in	tomcatv	(internal)
compress	in	nasa7	(internal)
gcc	stmt.i	mdljdp2	mdlj2.dat
xlisp	li-input.lsp	swm256	swm256.in
sc	loada1	su2cor	su2cor.in

Table 3: Benchmarks and their inputs.

4.3 Performance Metric

For measuring performance we use instructions completed per cycle (IPC), which is a direct measure of performance and is almost certainly the measure that counts most. The harmonic mean is used to average the performance of benchmarks.

5 Results

Table 4 summarizes the trace cache (TC), collapsing buffer (CB), and branch address cache (BAC) fetch unit parameters used in all experiments. In the sections which follow, results for the three high bandwidth implementations are compared. Previously, these methods could not be compared because the studies used different simulation environments (instruction sets, benchmarks, compilers, performance metrics) and different hardware configurations (execution model, instruction cache, branch predictor, etc.).

As a base case for comparison, results are also presented for conventional instruction fetching. The core fetch unit of the trace cache (described in Section 2.1) is used as the base case. We will call the base case "sequential" (SEQ), since only sequential instructions can be fetched in a given cycle. To demonstrate the effect of branch throughput, two variations of SEQ are simulated: SEQ.1 is limited to one basic block per cycle, and SEQ.3 can fetch up to three contiguous basic blocks. Simulation parameters for SEQ are the same as those for TC in Table 4, but with the trace cache not present (and 1 or 3 branch predictions per cycle).

SIMULATION PARAMETER		INSTRUCTION SUPPLY MECHANISM		
		TC	CB	BAC
instruction fetch limit		16 instructions per cycle		
Multiple Branch Predictor	BHR	14 bits		
	PHT	2^{14} 2-bit counters (4 KB storage)		
	# pred/cycle	up to 3 predictions each cycle		
Instruction Cache	size	128 KB		
	associativity	direct mapped		
	line size	16 instructions	16 instructions	4 instructions
	interleave factor	2-way	2-way	8-way
	miss penalty	10 cycles		
Return Address Stack	depth	unlimited		
Branch Target Buffer	size	1024 entries	1024 entries	n/a
	associativity	direct mapped	direct mapped	
	interleave factor	16-way	16-way	
Trace Cache	size	64 entries	n/a	
	associativity	direct mapped		
	line size	16 instructions		
	# concurrent fills	1		
Branch Address Cache	size	n/a		1024 entries
	associativity			direct mapped
	# concurrent fills			1

Table 4: Fetch unit configurations.

The simulators only differ in fetch unit-specific hardware. Because of the fundamental differences between fetch units, it is difficult to maintain an area budget over all the schemes. Therefore, area differences must always be kept in mind when comparing the results. Here we comment briefly about area comparisons:

- The trace cache used adds under 5 kB of additional SRAM storage to the core fetch unit, SEQ.3. The SRAM is assumed to have 1 read port and 1 write port.

- Superficially, the CB fetch unit differs from SEQ only in datapath after the instruction cache. However, the BTB must be dual-ported to support pipelining the CB fetch unit. In the worst case (replicating cells), a dual-ported BTB has twice the area. The BTB used throughout takes up about 7 kB of SRAM storage; if we assume 1.5 to 2 times that for dual-porting, add another 3.5 to 7 kB of storage for the CB fetch unit.
- A large BAC (1024 entries) is used in the experiments because the BAC must perform the functions of both the TC and BTB. The BTB is a particularly important resource because it gets most of the base performance. A rough area estimate for the BAC is about 60 kB (464 bits per entry [2]).

The results are split into two sets. The first set assumes all fetch units have a latency of 1 cycle, in order to demonstrate each mechanism's ability to deliver bandwidth performance. The second set shows what happens when the extra pipe stages implied by CB and BAC are actually simulated.

5.1 Single-Cycle Fetch Latency

The first set of results, Table 5 and corresponding graphs in Figures 10 and 11, assumes a fetch unit latency of 1 cycle for all schemes. This is done to isolate the ability of the fetch mechanisms to supply instruction bandwidth.

Benchmark	SEQ.1	SEQ.3	BAC	CB	TC
eqntott	3.05	3.30	3.96	4.16	4.24
espresso	3.17	4.10	4.42	4.61	5.20
xlisp	2.63	3.10	3.29	3.43	3.57
gcc	2.16	2.32	2.24	2.47	2.50
sc	3.17	3.64	4.10	4.33	4.43
compress	3.28	3.72	3.82	4.02	4.13
doduc	4.22	4.37	4.15	4.48	4.48
tomcatv	10.5	11.9	12.4	13.9	14.2
nasa7	8.41	8.56	8.52	8.63	10.5
mdljdp2	6.03	7.47	7.09	8.24	8.36
swm256	8.88	9.26	9.19	9.55	9.60
su2cor	4.66	4.77	4.72	4.90	5.02

Table 5: Performance of instruction fetch mechanisms, assuming unit fetch latency for all.

The first observation is that SEQ.3 gives a substantial performance boost over SEQ.1. The graph in Figure 12 shows that fetching past multiple not-taken branches each cycle yields performance improvement above 7% for all of the integer benchmarks. Half of the integer benchmarks show a 15% or better performance improvement. Only two of the floating point benchmarks show similar improvement.

The second observation is that for integer benchmarks, fetching past taken branches is a big win. Adding the TC function to the SEQ.3 mechanism yields as much performance improvement

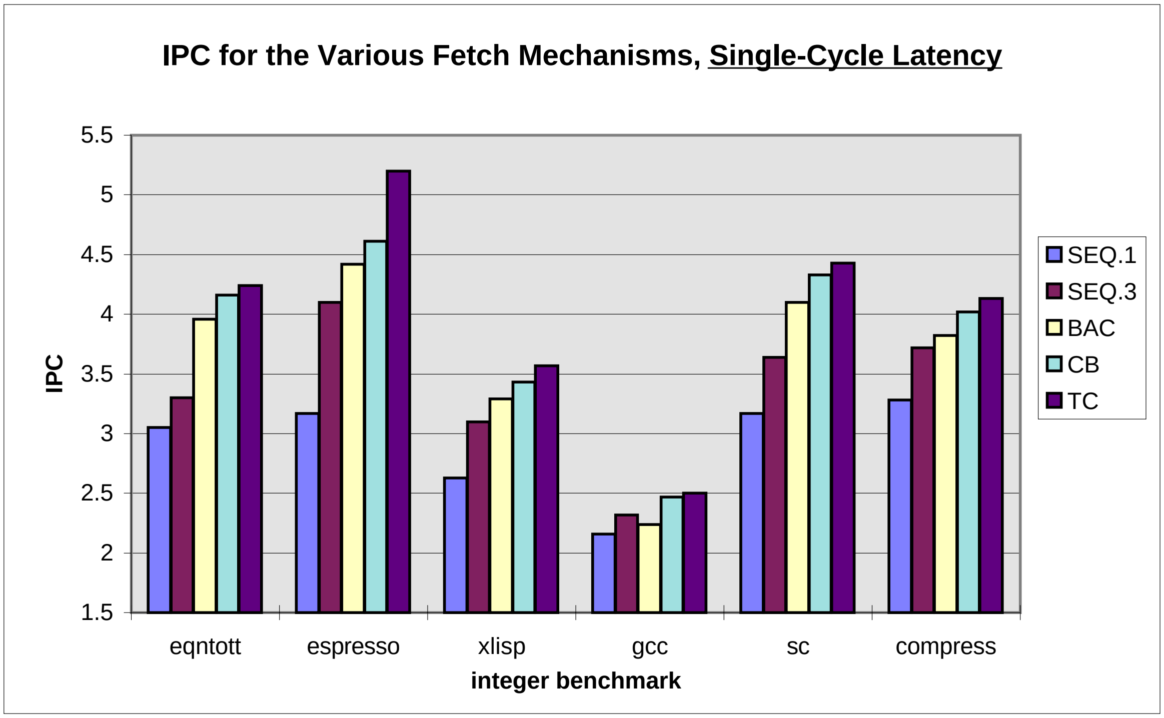


Figure 10: IPC for the various fetch mechanisms, assuming a fetch unit latency of 1 cycle for all designs. (integer benchmarks)

as extending SEQ.1 to multiple not-taken branches per cycle, and in some cases significantly more improvement. Somewhat surprisingly, half of the floating point benchmarks see substantial performance improvement (12% or more) with the TC.

The graph in Figure 13 shows the performance improvement that BAC, CB, and TC yield over SEQ.3 (SEQ.3 is used as the base instead of SEQ.1 because it is aggressive, yet not much more complex than SEQ.1). One might expect that under the single-cycle fetch latency assumption, the three approaches would perform similarly. However, TC enjoys a noticeable lead over CB. This is most likely because the original collapsing buffer was not designed to handle backward taken intrablock branches [1], whereas the TC can handle any arbitrary trace. The BAC performs worst of the three, and in some cases even performs *below* SEQ.3 – particularly in floating point. There are two explanations for this behavior:

- Instruction cache bank conflicts are the primary performance loss for BAC. Table 6 shows that BAC is comparable to TC if bank conflicts are ignored, again assuming single-cycle fetch latency for BAC.

- The BAC treats basic blocks as atomic units. As a result, a BAC entry will provide only as many basic block addresses as will fit within the 16 instruction fetch limit. The consequence is that given hits in both the TC and BAC, the BAC can never supply more instructions than the TC. This situation can be remedied by having a wider fetch datapath than the dispatch datapath, which amounts to prefetching.

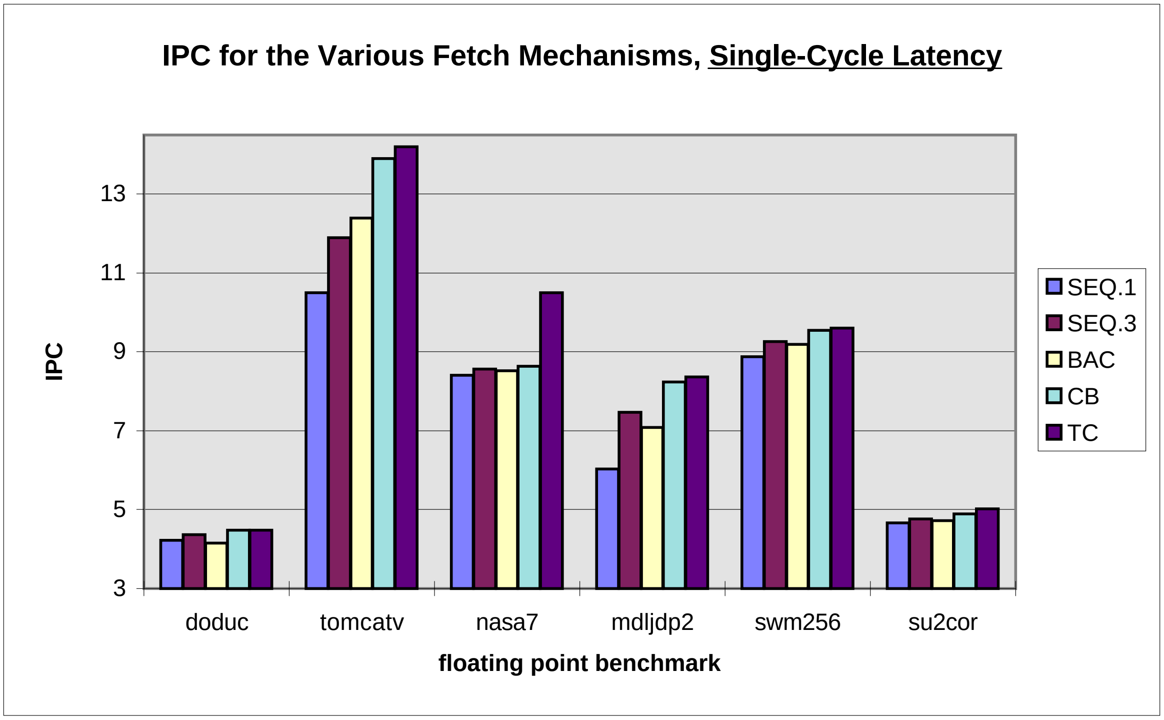


Figure 11: IPC for the various fetch mechanisms, assuming a fetch unit latency of 1 cycle for all designs. (floating point benchmarks)

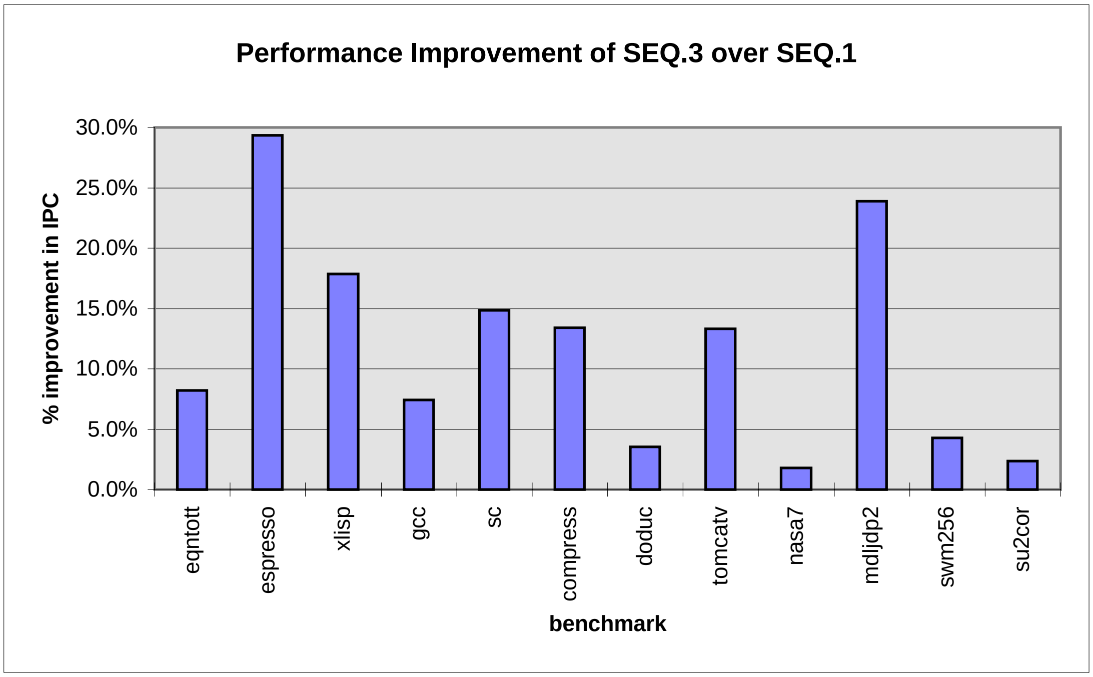


Figure 12: Performance improvement of SEQ.3 over SEQ.1.

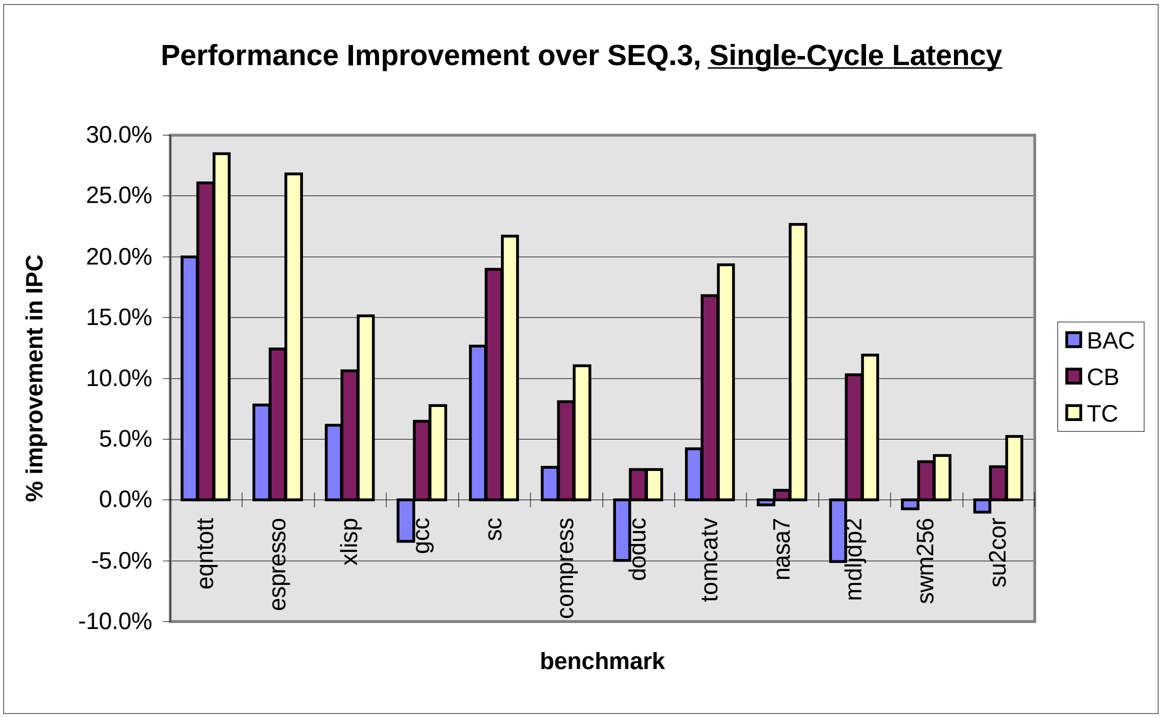


Figure 13: Performance improvement of BAC, CB, and TC over SEQ.3, assuming a fetch unit latency of 1 cycle for all designs.

Benchmark	BAC (conflicts)	BAC (no conflicts)	TC
eqntott	3.96	4.24	4.24
espresso	4.42	5.34	5.20
xlisp	3.29	3.46	3.57
gcc	2.24	2.36	2.50
sc	4.10	4.48	4.43
compress	3.82	3.93	4.13

Table 6: The performance of BAC is severely limited by instruction cache bank conflicts. This table shows that the performance of BAC is comparable to TC if bank conflicts are ignored, assuming unit fetch latency.

5.2 The Effect of Latency

The effect of fetch unit latency is quantified in Table 7 and corresponding graphs in Figures 14 and 15. Since both the CB and BAC schemes add stages before and after the instruction cache, we give the performance of these schemes as fetch latency is varied from 1 to 3 cycles. Figure 16 shows that the impact of latency is quite severe. CB and BAC fall well below the performance of TC. For all but 3 of the benchmarks, BAC with a latency of 2 cycles performs worse than SEQ.3. Likewise, for over half of the benchmarks, CB with a latency of 3 cycles performs worse than SEQ.3.

The effect of latency is more dramatic for integer code than floating point code. Still, only one benchmark – *tomcatv* – appears to be tolerant of fetch unit latency.

Benchmark	CB	CB (L2)	CB (L3)	BAC	BAC (L2)	BAC (L3)
eqntott	4.16	3.86	3.60	3.96	3.68	3.44
espresso	4.61	4.34	4.10	4.42	4.17	3.94
xlisp	3.43	3.17	2.95	3.29	3.08	2.89
gcc	2.47	2.25	2.07	2.24	2.07	1.92
sc	4.33	4.00	3.73	4.10	3.64	3.40
compress	4.02	3.83	3.65	3.82	3.65	3.49
doduc	4.48	4.33	4.19	4.15	4.01	3.87
tomcatv	13.9	13.8	13.7	12.4	12.3	12.2
nasa7	8.63	8.27	7.93	8.52	8.18	7.85
mdljdp2	8.24	8.01	7.80	7.09	6.91	6.73
swm256	9.55	9.44	9.31	9.19	9.06	8.92
su2cor	4.90	4.75	4.61	4.72	4.57	4.43

Table 7: Performance of the CB and BAC schemes with latency taken into account. L2 = 2 cycle fetch latency, L3 = 3 cycle fetch latency.

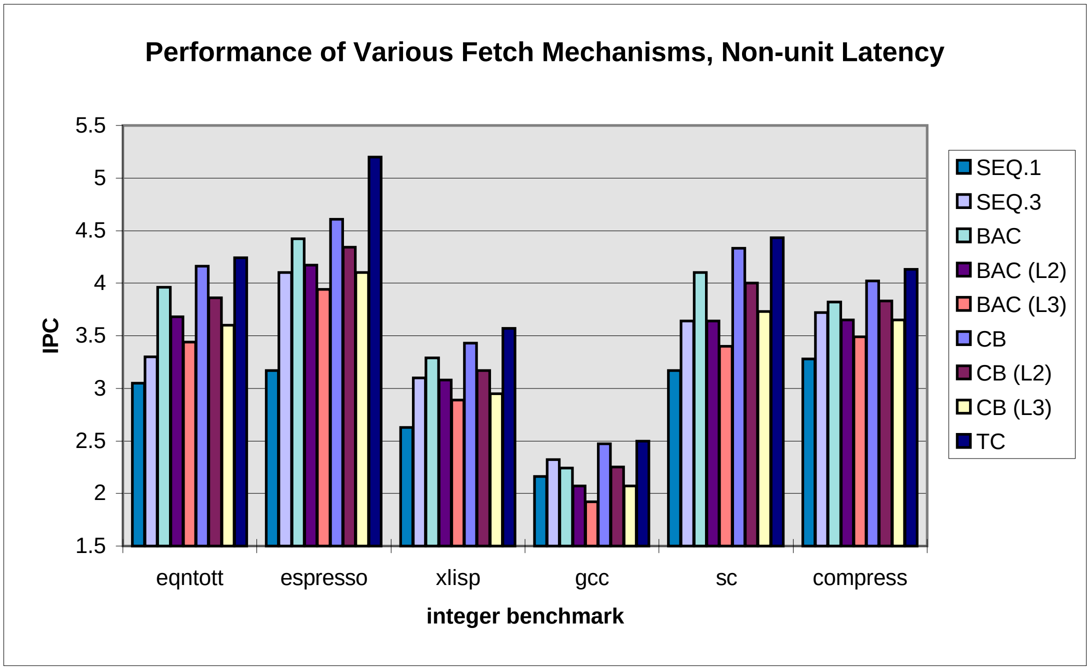


Figure 14: IPC for the various fetch mechanisms, using realistic latencies for CB and BAC. *L2* and *L3* stand for 2 cycle and 3 cycle latency, respectively. (integer benchmarks)

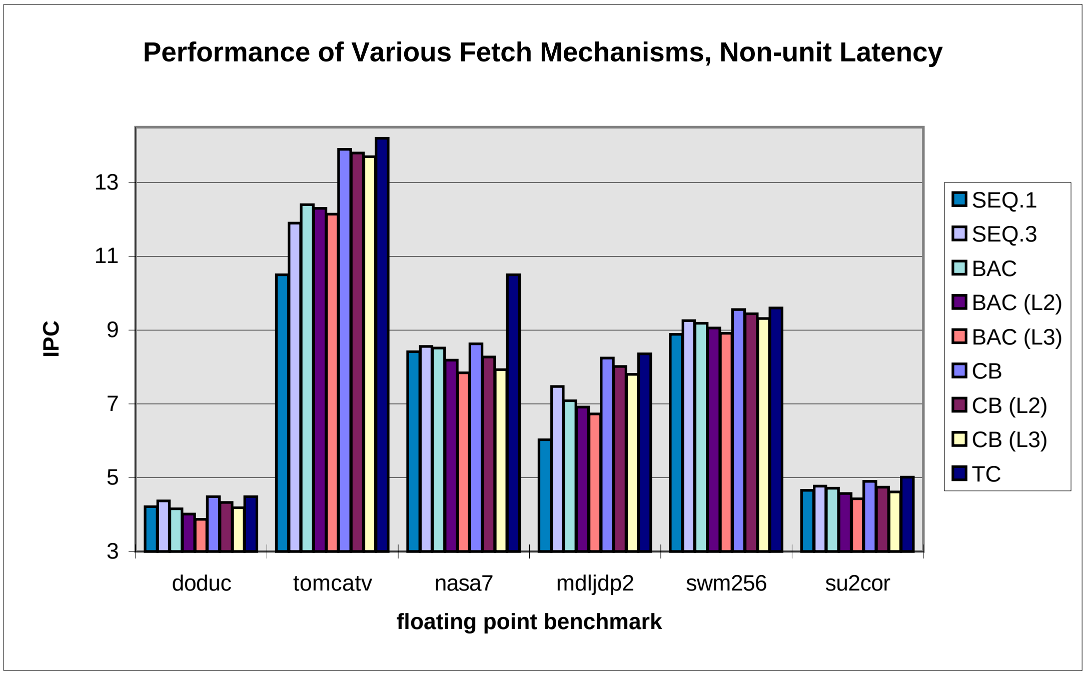


Figure 15: IPC for the various fetch mechanisms, using realistic latencies for CB and BAC. *L2* and *L3* stand for 2 cycle and 3 cycle latency, respectively. (floating point benchmarks)

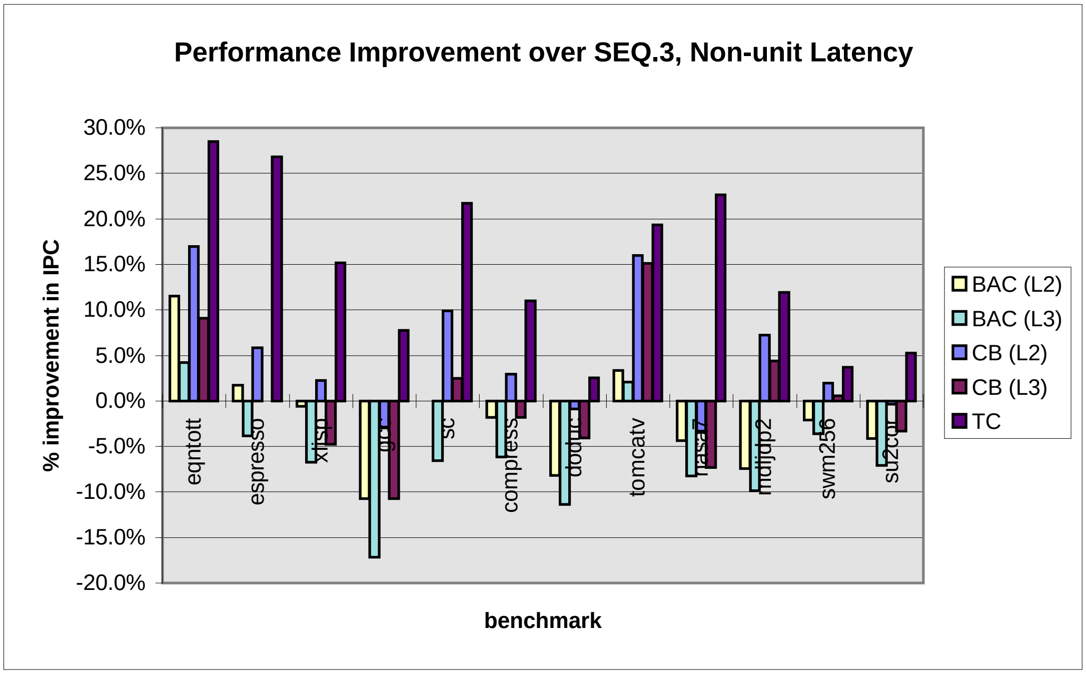


Figure 16: Performance improvement of BAC, CB, and TC over SEQ.3, using realistic latencies for CB and BAC. *L2* and *L3* stand for 2 cycle and 3 cycle latency, respectively.

5.3 Trace Cache Effectiveness

To determine how effective the trace cache is, we must establish the upper bound on its performance and measure how far short it falls from this bound. The bound is established by an "ideal" fetch model, defined as follows: as long as branch outcomes are predicted correctly and instructions hit in the instruction cache, up to 3 basic blocks or 16 instructions – whichever comes first – can be fetched every cycle.

Figure 17 shows that there is still performance to be gained by better instruction fetching. TC falls short of ideal performance due to trace cache and BTB misses. The trace cache used here has only 64 entries and is direct mapped; adding 2/4-way associativity or simply increasing the number of entries will narrow the performance gap between TC and ideal. Figure 17 provides incentive to explore the design space alternatives of Section 2.3 aimed at improving hit rate.

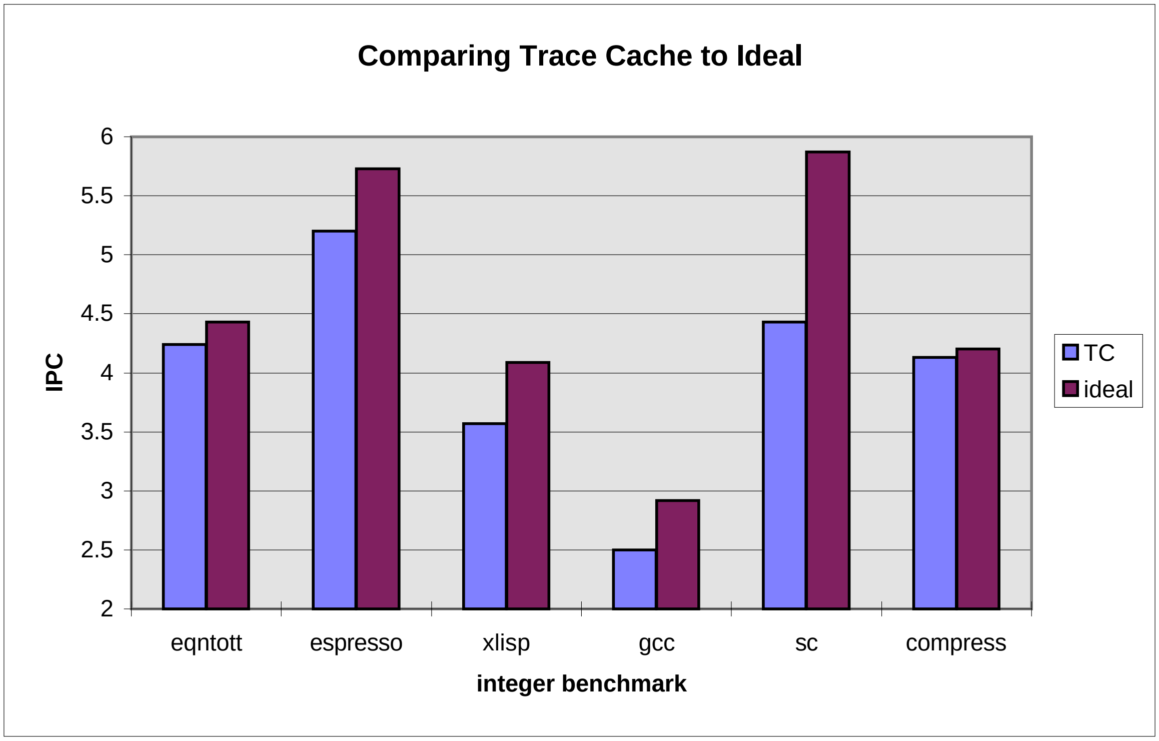


Figure 17: TC falls short of full performance potential due to trace cache misses and BTB misses.

Trace cache miss rate can be specified in two ways: in terms of traces (trace miss rate) and in terms of instructions (instruction miss rate). Trace miss rate is the fraction of accesses that do not find a trace present. Instruction miss rate is the fraction of instructions not supplied by the trace cache. Trace miss rate is a more direct measure of trace cache performance because it indicates the fraction of fetch cycles that benefit from higher bandwidth. However, instruction miss rate is also reported because it corresponds to cache miss rate in the traditional sense. Miss rates are shown in Table 8.

Benchmark	trace miss rate	instruction miss rate
eqntott	27.4%	7.80%
espresso	41.9%	21.1%
xlisp	64.0%	39.7%
gcc	70.7%	51.2%
sc	50.7%	27.7%
compress	17.8%	6.10%
doduc	66.8%	59.1%
tomcatv	61.2%	59.9%
nasa7	9.80%	2.50%
mdljdp2	14.1%	4.30%
swm256	79.0%	76.1%
su2cor	21.4%	5.80%

Table 8: Trace miss rate and instruction miss rate for a 64 entry direct mapped trace cache.

6 Conclusions

Trends in processor organization – wider dispatch/issue, larger instruction windows, and deeper speculation – expose new instruction fetch performance issues. These are branch throughput, noncontiguous instruction alignment, and fetch unit latency.

Experiments with real implementations yield the following results:

- For integer code, being able to fetch past multiple *not-taken* branches improves performance on average by 14% over a fetch unit that is limited to 1 branch prediction per cycle. For floating point, the average performance improvement is 7%.
- For integer code, being able to fetch past multiple *taken* branches improves performance on average by 17% over a fetch unit that can only fetch contiguous instructions (this result is for the trace cache implementation). For floating point, the average performance improvement is 9%.
- The trace cache consistently performs better than either of the other high bandwidth fetch mechanisms studied – collapsing buffer and branch address cache – *even if unit fetch latency is assumed across all three.* The primary reasons for this are (1) the original CB design does not handle backward taken intrablock branches and (2) the BAC design suffers from instruction cache bank conflicts.
- Simulations with more realistic latencies for the CB and BAC designs (based on stages before and after the instruction cache) clearly show the advantage of using a low latency approach. Their performance falls well below that of the trace cache, and in some cases a low latency, lower bandwidth fetch unit does better.

The combined effect of multiple branches per cycle and noncontiguous instruction fetching is an average improvement of 34% for integer benchmarks and 16% for floating point benchmarks, using the trace cache.

While a small trace cache performs well, comparison with the "ideal" noncontiguous instruction fetch model shows the potential for even higher performance – additional improvement in the range of 5% to 30%. This experiment motivates investigation of larger and/or more complex trace cache designs, such as path associativity, partial matches, judicious trace selection, and victim trace caches.

In conclusion, it is important to design fetch units capable of fetching past multiple, possibly taken branches each cycle. However, this additional bandwidth performance should not be achieved at the expense of longer fetch unit latency. The trace cache is successful in satisfying both of these requirements.

7 Acknowledgements

This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

Eric Rotenberg is funded by a Graduate Fellowship from IBM.

References

[1] T. Conte, et al. , "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[2] T-Y Yeh, D. Marr and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[3] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994.

[4] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," *to appear Micro-28*, Nov. 1995.

[5] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, January 1984.

[6] J. Losq, "Generalized History Table for Branch Prediction," *IBM Technical Disclosure Bulletin*, June 1982.

[7] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, January 1990.

[8] S-T Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[9] T-Y Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[10] T-Y Yeh, "Two-level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High-Performance Superscalar Processors," PhD Thesis, Department of Electrical Engineering and Computer Science, University of Michigan, 1993.

[11] T-Y Yeh and Y. N. Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," .

[12] E. Hao, P. Chang and Y. Patt, "The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited," *Proceedings of the 27th annual International Symposium on Microarchitecture*, December 1994.

[13] D. Kaeli and P. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991.

[14] D. Wall, "Limits of Instruction-Level Parallelism," *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[15] L. Gwennap, "MIPS R10000 Uses Decoupled Architecture," *Microprocessor Report*, October 1994.

[16] J. Larus, "Efficient Program Tracing," *IEEE Computer*, May 1993.

[17] S. Melvin, M. Shebanow and Y. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proceedings of the 21st Annual International Symposium on Microarchitecture*, December 1988.

[18] J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Symposium on computer Architecture*, May 1981.

[19] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceedings IEEE*, December 1995.

[20] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

A Logic Implementation Issues

This appendix serves as a repository for many of the issues that we encountered in our study of the various high bandwidth instruction fetch mechanisms. There is no significance to the order that things are presented, except that several subsections depend on the BTB description, which is presented first. Also, a given section may be relevant to one or more different fetch mechanisms.

A.1 Concerning the Branch Target Buffer

The branch target buffer differs from its original design in that it must locate branches within a *block* of instructions. The conventional branch target buffer takes a single word address and determines if there is a branch at that address, a scheme originally intended for single issue processors. Extending the branch target buffer to detect multiple branches in a block of instructions is fairly straightforward. Conte et al. [1] propose an interleaved branch target buffer where the number of banks is equal to the number of instructions in a cache line. In this way, all locations in the fetched cache line can be checked for branches in parallel. (The address of the first instruction in the cache line is used to index the first bank; the address of the second instruction indexes the second bank; and so on.)

Barring misses, the branch target buffer provides all of the branch information contained in a cache line. This information must be combined with multiple branch predictions to determine two things: which instructions in the fetched line are valid and what is the next fetch address. Matching predictions to their intended branches requires special logic after the branch target buffer. Conceptually this logic scans the branch information from the first bank to the last bank counting branches. This counting has an inherent ripple effect which is nontrivial for larger cache lines. The logic diagram in Figure 18 shows a possible implementation where banks are counted in groups of four to reduce the amount of ripple. If four is not sufficient, even more "lookahead" can be used.

We now present a detailed implementation of the BTB logic. The purpose of this logic is to combine multiple branch predictions with branch information from the BTB to determine which instructions in the fetched cache line are valid and to produce the next fetch address.

The implementation assumes a cache line size of 16 instructions and 3 branch predictions per cycle. The following notation and signals are used in expressions and logic diagrams:

- x: BTB bank number, also instruction position in the cache line. x ranges from 0 to 15.
- hit_x: Bank x of the BTB indicates a hit. In other words, there is a branch at instruction position x of the cache line.
- $ADDR_{5:2}$: Bits 2 through 5 of the fetch address point to the first valid instruction in the cache line.
- b_x: If set, there is a valid branch at instruction position x in the cache line.
- one_x: If set, there is *at least* one valid branch preceding the instruction at position x.
- two_x: If set, there are *at least* two valid branches preceding the instruction at position x.
- $three_x$: If set, there are *at least* three valid branches preceding the instruction at position x.
- p_0: First branch prediction.
- p_1: Second branch prediction.

- p_2: Third branch prediction.
- M_x: Mask bit for the instruction at position x in the cache line.
- SEL_x: If this signal is set, the target address from bank x of the BTB is used as the next fetch address.

The outputs of the logic are a bit vector indicating valid instructions and the next fetch address. The bit vector can be derived from the mask bits M_x. If there is a taken branch, the next fetch address is selected using the SEL_x signals. Here are the necessary expressions:

$$b_x = hit_x(x \geq ADDR_{5:2}) \tag{1}$$

$$M_x = (p_0 one_x + p_1 two_x + three_x) + (x \leq ADDR_{5:2}) \tag{2}$$

$$SEL_x = b_x(\overline{one_x} p_0 + one_x \overline{two_x}\,\overline{p_0} p_1 + two_x \overline{three_x}\,\overline{p_0 p_1} p_2) \tag{3}$$

Most of the complexity is in computing one_x, two_x, and $three_x$ for all x. Essentially this involves counting the number of valid branches from the leftmost bank to the rightmost bank. The ripple effect introduces too long a delay if counting proceeds one bank at a time. Figure 18 shows how 4 banks can be grouped together for counting branches, analogous to carry lookahead logic. The diagram shows how the ripple path (shaded region) sees only 2 additional gate levels every 4 banks.

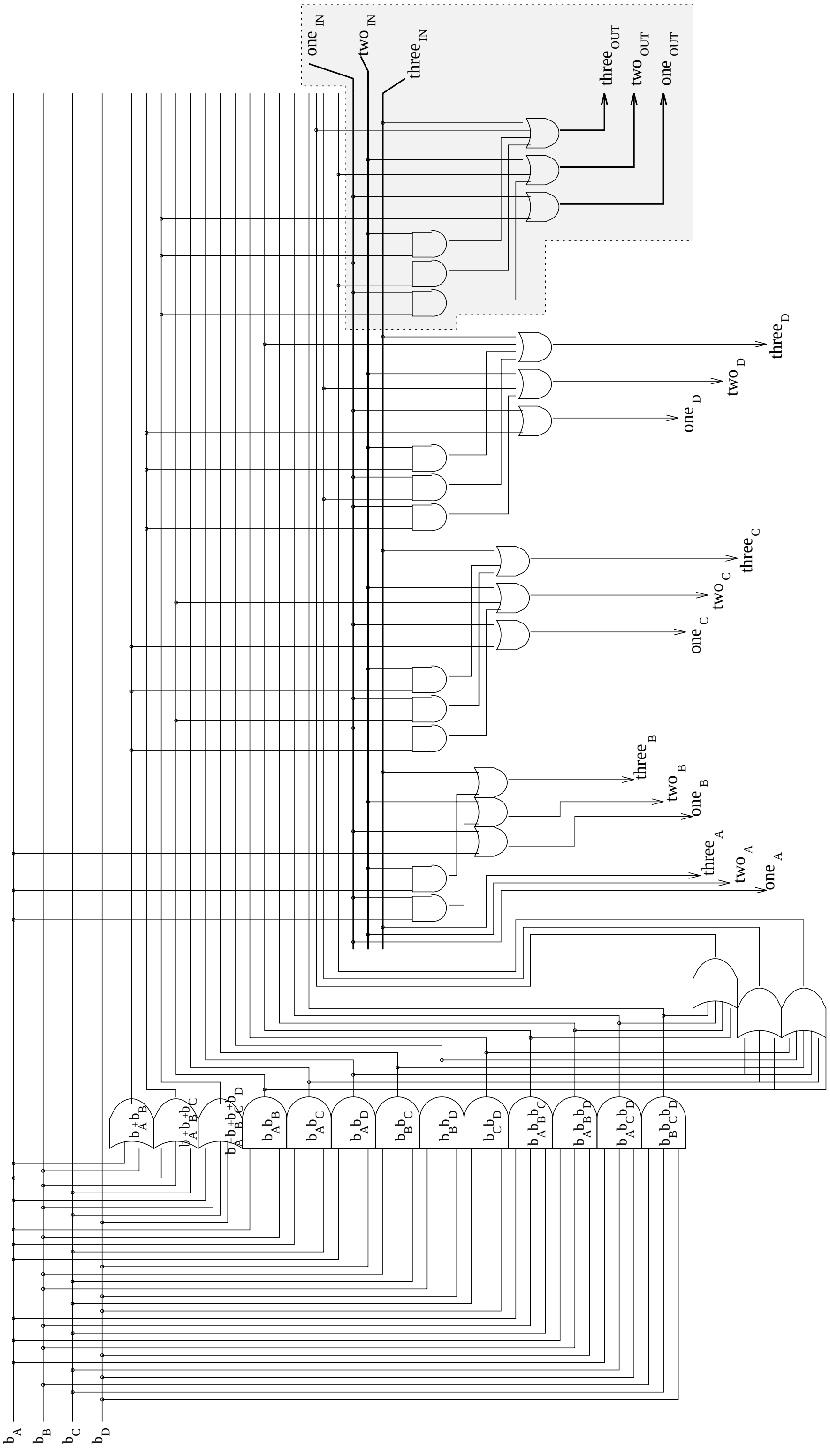


Figure 18: Counting the number of branches within a group of 4 BTB banks. The shaded region highlights the critical path. If there are 16 instructions in a cache line, and hence 16 BTB banks, 4 of these logic blocks are daisy-chained together.

A.2 Concerning the Multiple Branch Predictor

Predicting the next sequence of basic blocks to be fetched requires predicting the outcomes of multiple intervening branches. The predictor must also be highly accurate so that the fetch unit can speculate past many branches to form a large, accurate window of instructions for the execution engine. The multiple branch predictor used throughout this paper is that proposed by Yeh et al. [2]. Before describing the multiple branch predictor, it is necessary to briefly review the single branch predictor on which it is based – the correlation predictor [8].

The correlation predictor is shown in Figure 19. Correlation prediction uses the fact that the outcome of a branch tends to be influenced by the outcomes of branches preceding it. It is one of nine Two-level Adaptive Branch Prediction schemes [9] all of which use two levels of history to achieve high accuracy. The global branch history register (BHR) and the pattern history table (PHT) reflect the two levels of history. A BHR k bits in length keeps a record of the outcomes of the last k branches. As new branches are encountered, their outcomes are shifted into the least significant bit of the BHR. The PHT is a table of 2^k 2-bit counters, one counter for each possible pattern in the BHR. To predict the next branch, the BHR is used to index the PHT and a prediction is made based on the value of the corresponding PHT entry. In summary, the current prediction depends on (1) the pattern of k preceding branches (the BHR), and (2) the behavior of the branch the last few times that pattern was encountered (2-bit counter in the PHT). When the outcome of the branch is known, it is used to update the same 2-bit counter in the PHT (increment if taken, decrement if not taken).

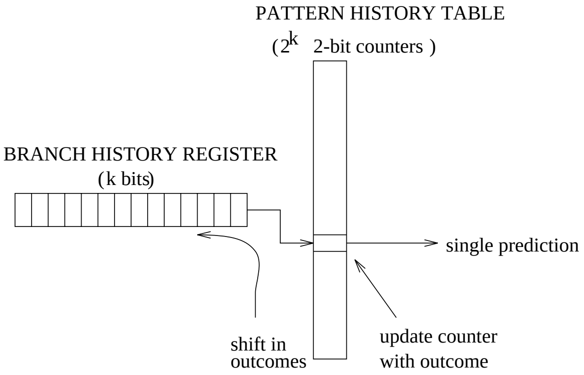


Figure 19: Correlation branch predictor.

Adapting the predictor for multiple branch predictions in one cycle requires changing both the physical layout of the PHT and the way that the structures are updated. Conceptually the PHT must be accessed multiple times, each time using successive shifts of the BHR as an index. It is impractical to assume that multiple sequential accesses of the PHT can be made in a single cycle. Figure 20 shows one possible implementation of the multiple branch predictor. There are two key features. First, the PHT has multiple read ports, one for each prediction. Second, the PHT is organized such that a *row* of counters is read from each port. This is because the low order bits of successive BHR indices are incomplete. These low order bits are only known after accessing the PHT; they feed a chain of multiplexors after the PHT to make final selections. Instead of updating the structures with branch *outcomes*, they must be speculatively updated with branch *predictions*. The BHR needs to shift in predictions because outcomes are simply unavailable. On the other hand, speculatively updating the PHT is not a strict requirement, but from a heuristic standpoint it is preferred over using stale history. In the event that a branch is mispredicted, the structures should be repaired. Repairing the BHR is easily done by maintaining a second, "golden" BHR which is updated with true branch outcomes; when a misprediction is detected, the golden BHR is loaded into the speculative BHR (rolling back the state). Repairing entries in the PHT is more complex; it would be worthwhile to study the performance impact of *not* repairing PHT entries.

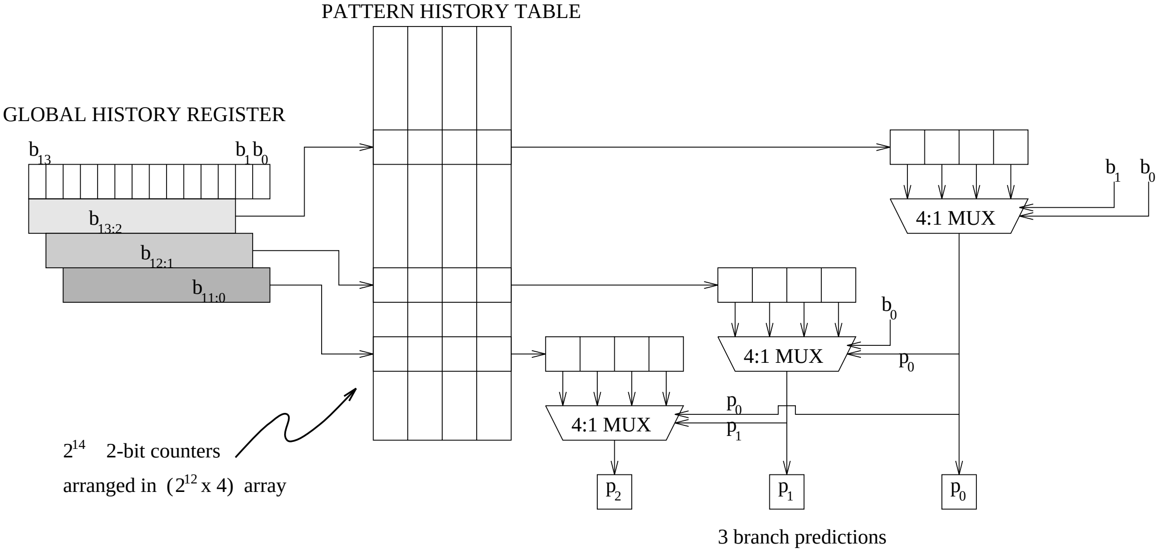


Figure 20: Correlation branch predictor adapted for multiple branch predictions in a cycle.

When fetching from the instruction cache, correct handling of branches requires coordination of the multiple branch predictor, branch target buffer, and decode logic after the instruction latch. The following rules provide this coordination.

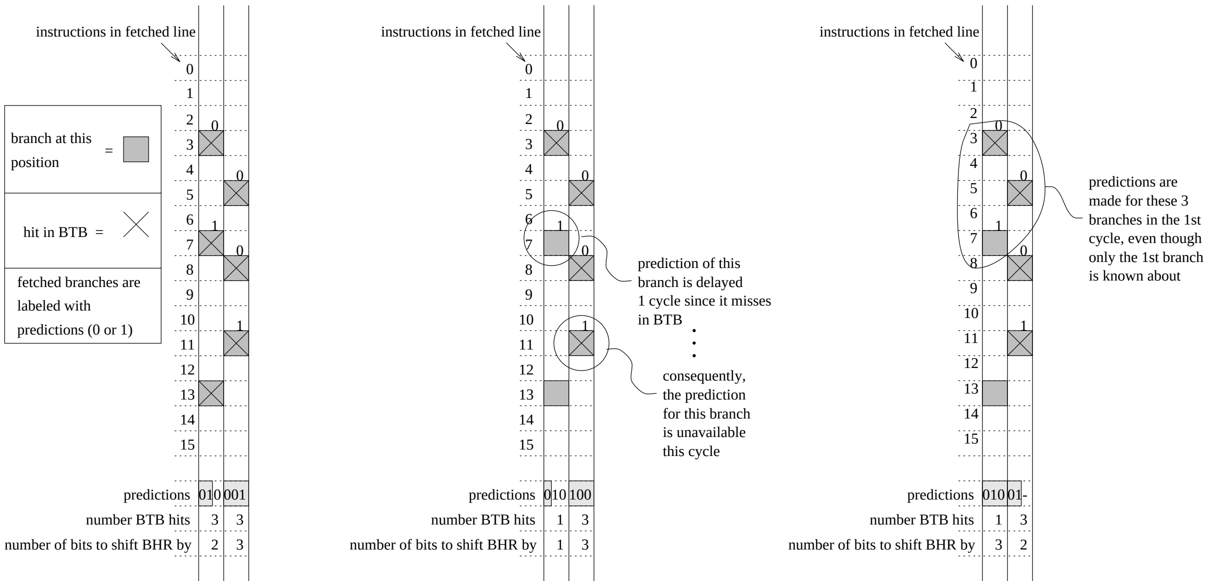
- If a branch misses in the branch target buffer, the branch is automatically predicted not taken. Since the branch is not detected until the decode stage (a cycle after fetching), the simplest alternative is to continue fetching sequentially. Even if the predictor predicted a taken branch, it is treated as not taken.

- The predictor itself does not know when to shift the BHR, and by how many bits, since it has no knowledge of branches in the fetch stream. The problem is illustrated in Figure 21. If all branches in the fetched line hit in the branch target buffer, there is no problem – the number of bits to shift the BHR by is equal to the number of branches up to and including the first taken branch (Figure 21(a)). However, if some branches do not hit in the branch target buffer, the predictor will not know about them until it is too late (the decode stage). As a consequence, the BHR shifts less bits than required, and could *fall behind* the fetch unit in subsequent cycles (Figure 21(b)). The solution is to always keep the BHR ahead (Figure 21(c)). That is, if the predictor is capable of p predictions per cycle, it should stay up to p branches ahead of the fetch stream.

- The p predictions must be paired with their respective branches. Again, there is no problem if all branches in the fetched line hit in the branch target buffer – the first prediction goes to the first branch, the second prediction to the second branch, and so on (Figure 22(a)). However, if any branch misses in the branch target buffer, predictions can get "skewed" (Figure 22(b)) – subsequent branches are assigned predictions intended for previous branches. There is no way to avoid the problem, although recovery steps can be initiated in the decode stage. However, to be consistent with the first rule above, prediction "skew" is simply tolerated.

- If more than p branches are detected in the fetched cache line, only instructions up to the p^{th} branch are supplied to the decoder. Fetching of instructions after the p^{th} branch is delayed until the next cycle. This rule is imposed so that the predictor does not fall behind (its throughput is limited to p predictions per cycle).

Notice that predecode information in the instruction cache pertaining to the number and location of branches can help the fetch mechanism a great deal.

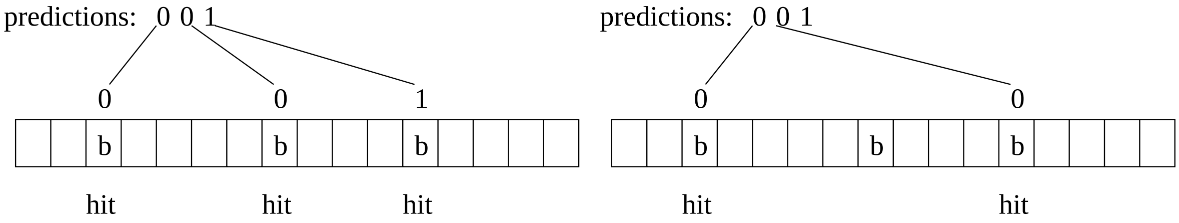

instructions in fetched line
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
branch at this position =
hit in BTB =
fetched branches are labeled with predictions (0 or 1)
predictions 010 001
number BTB hits 3 3
number of bits to shift BHR by 2 3
instructions in fetched line
prediction of this branch is delayed 1 cycle since it misses in BTB
consequently, the prediction for this branch is unavailable this cycle
predictions 010 100
number BTB hits 1 3
number of bits to shift BHR by 1 3
instructions in fetched line
predictions are made for these 3 branches in the 1st cycle, even though only the 1st branch is known about
predictions 010 01-
number BTB hits 1 3
number of bits to shift BHR by 3 2

(a) If all branches hit in the BTB, there is no problem determining how many predictions are made this cycle and therefore how many bits to shift the BHR by.

(b) In this example the second branch is undetected in the first cycle since it misses in the BTB. As a result, only 1 prediction is made in the first cycle and the BHR is shifted by only 1 bit. In the next cycle, the decoder indicates that there *was* a second branch, and in addition there are 3 new branches. This calls for 4 new branch predictions, but the predictor can only provide 3. *Not anticipating the second branch in the first cycle causes the predictor to fall behind.*

(c) This example shows the predictor keeping up to 3 branches ahead of fetching, to anticipate any branches which are undetected until the decode stage. The predictor makes 3 predictions and shifts the BHR by 3 bits in the first cycle, even though only 1 branch is detected by the BTB. Only 2 predictions are needed in the first cycle, so the third prediction is used in the second cycle.

Figure 21: Determining how many predictions to make and therefore how many bits to shift the BHR by.



(a) If all branches hit in the BTB, there is no problem pairing up predictions with their intended branches.

(b) If a branch misses in the BTB, predictions may be wrongly assigned to branches.

Figure 22: Assigning predictions to branches and the problem of prediction "skew".

A.3 Next PC MUX

The next fetch address is fed by a very large MUX. The following addresses are candidates for next fetch address:

- *PC feedback* – no change in the PC.
- *Sequential PC* – the PC incremented by the fetch bandwidth.
- *Recover PC* – feedback from the execution engine which redirects fetching.
- *RAS PC* – pop return address off the return address stack if a return is detected.
- *trace target* – if a trace cache exists, and if it hits, this address is the target of trace.
- *trace fall-through* – if a trace cache exists, and if it hits, this address is the fall-through of trace.
- *BTB branch targets* – if the BTB detects a taken branch, must select the appropriate branch target from 1 of 16 banks.

By far the critical path exists in the MUX tree used to select 1 of the 16 BTB target addresses, due to the complexity in generating the $SEL_{15:0}$ select signals (refer to Section A.1). The select signals are increasingly timing critical going from bit 0 to bit 15. The datapath in Figure 23 reflects this by skewing the MUX tree such that $SEL_{15:12}$ goes through minimal logic. The expected critical path is labelled, though other critical paths may be exposed in the effort to reduce the primary one.

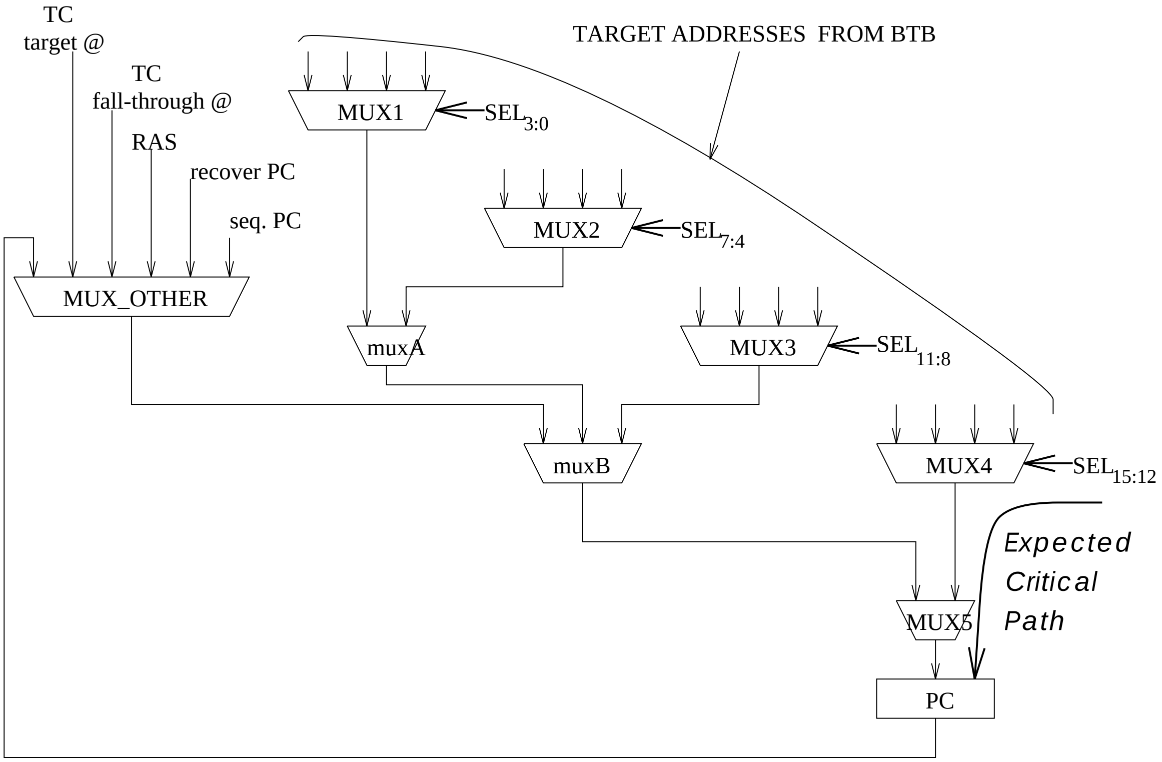


Figure 23: The next PC MUX logic.

A.4 Logic After the Instruction Cache

A.4.1 Base Instruction Cache

The base instruction cache does not provide any support for fetching across taken branches, yet it is high performance in one respect: it can always provide a full cache line worth of instructions or up to the first taken branch. In other words, it handles the problem of "falling off" the edge of a cache line by two-way interleaving the cache. With two cache banks, two adjacent cache lines can be accessed.

Three stages of logic are required after the base instruction cache. It will be shown that this logic adds much less delay than other cache designs which attempt to align noncontiguous blocks of instructions in the fetch path (these are treated in the following sections).

The three functions that must be performed are (1) masking off unused instructions, (2) possibly interchanging the output of the two cache banks, and (3) left-shifting the instructions into the final instruction latch by an arbitrary amount. The first two functions are trivial, and thus only the third function is discussed here.

Figure 24 shows the three functions that are performed. At first it might seem that the shifter function adds another full stage to the fetch unit latency. This is not the case for two reasons:

- Shifting left by an arbitrary amount can be done quite fast using a barrel-shifter. Figure 25 shows a 1-bit slice of the shifter. If there are 16 instructions in a cache line, the 1-bit slice requires 256 transmission gates. A full shifter requires 32 of these logic blocks (since there are 32 bits/instruction), for a total of 8,192 transmission gates. The delay is minimal: data signals pass through only 1 transmission gate, and in the worst case the signals see an input load of 15 other transmission gates and an output load of 15 other transmission gates. Both area and capacitive loading can be reduced by precharging the output bus and removing the p-transistors from the transmission gates.

- The delay through the shifter *datapath* is small. However, the real problem is with the control signals $S_0, S_1, S_2, \ldots, S_{15}$. Each signal feeds 16 transmission gates per bit, for 32 bits: this means a load of 512 gates. *Fortunately, the shift control signals are based solely on $ADDR_{5:2}$, the 4 least significant bits of the fetch address.* This means the control signals have nearly a full cycle to fanout to 512 gates, in parallel with the cache access. Assuming the technology supports driving 3-gates without repowering, a buffer fanout tree 5 levels deep is sufficient to drive 512 gates. Thus the control signals will have no problem reaching the shifter before the instructions arrive.

A.4.2 Collapsing Buffer

Figure 26 shows a bus-based crossbar implementation of the collapsing buffer [1]. Not shown are the control signals required to route instructions into the final instruction latch. Somehow the two *valid instruction bit vectors* generated from two passes through the BTB logic must be converted into crossbar switch signals. As will be shown, generating these control signals is nontrivial.

Before describing the control logic, here is some notation:

- $S_{i,j}$ – single bit control signal which routes instruction i of the input into instruction j of the output.

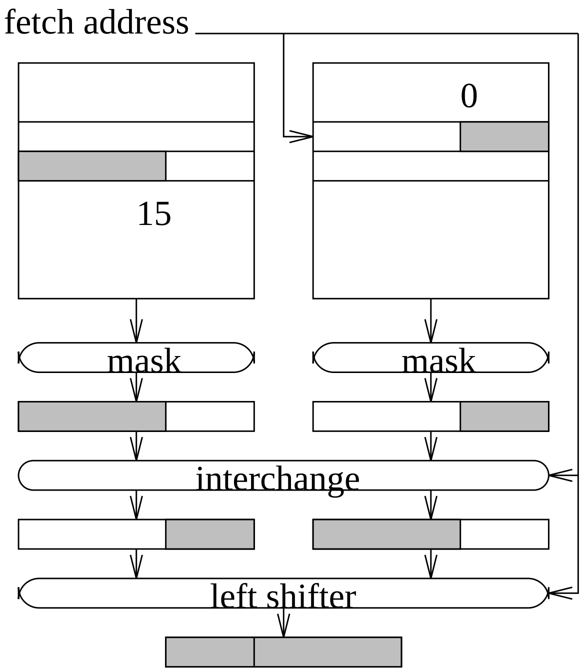


Figure 24: Demonstrating logic after the base instruction cache.

- v_i – i^{th} bit of the valid instruction bit vector. If set, there is a valid instruction at position i of the input.
- $one_i, two_i, three_i, \ldots, fifteen_i$ – there are *at least* one, two, three, etc. valid instructions before the i^{th} instruction.

Most of the complexity and critical timing is in generating the $one, two, three, \ldots$ signals. Similar signals are generated by the BTB logic, described earlier in Section A.1. The important thing to realize is that this logic is essentially keeping count of the number of instructions before a given position; counting is inherently serial and thus timing critical.

Once the critical $one, two, three, \ldots$ signals are generated, they can be used to generate the actual crossbar switch matrix as follows.

$$\begin{bmatrix} S_{0,0} & S_{1,0} & S_{2,0} & \ldots & S_{31,0} \\ & S_{1,1} & S_{2,1} & \ldots & S_{31,1} \\ & & S_{2,2} & \ldots & S_{31,2} \\ & & & \ddots & \vdots \\ & & & & S_{31,15} \end{bmatrix} = \begin{bmatrix} m_0 & m_1\overline{one_1} & m_2\overline{one_2} & \ldots & m_{31}\overline{one_{31}} \\ & m_1 one_1 & m_2 one_2\overline{two_2} & \ldots & m_{31} one_{31}\overline{two_{31}} \\ & & m_2 two_2 & \ldots & m_{31} two_{31}\overline{three_{31}} \\ & & & \ddots & \vdots \\ & & & & m_{31} fifteen_{31} \end{bmatrix} \tag{4}$$

After the routing control signals are generated, they are fanned out to the bus drivers.

The following latency problems can be identified with this fetch mechanism:

- First, generating the valid instruction bit masks is *even more* complicated than the logic discussed in Section A.1. The reason: handling intrablock branches requires a serial chain of comparators from one BTB bank to the next [1]. We have not attempted to merge the comparator logic in with our preexisting BTB logic design.

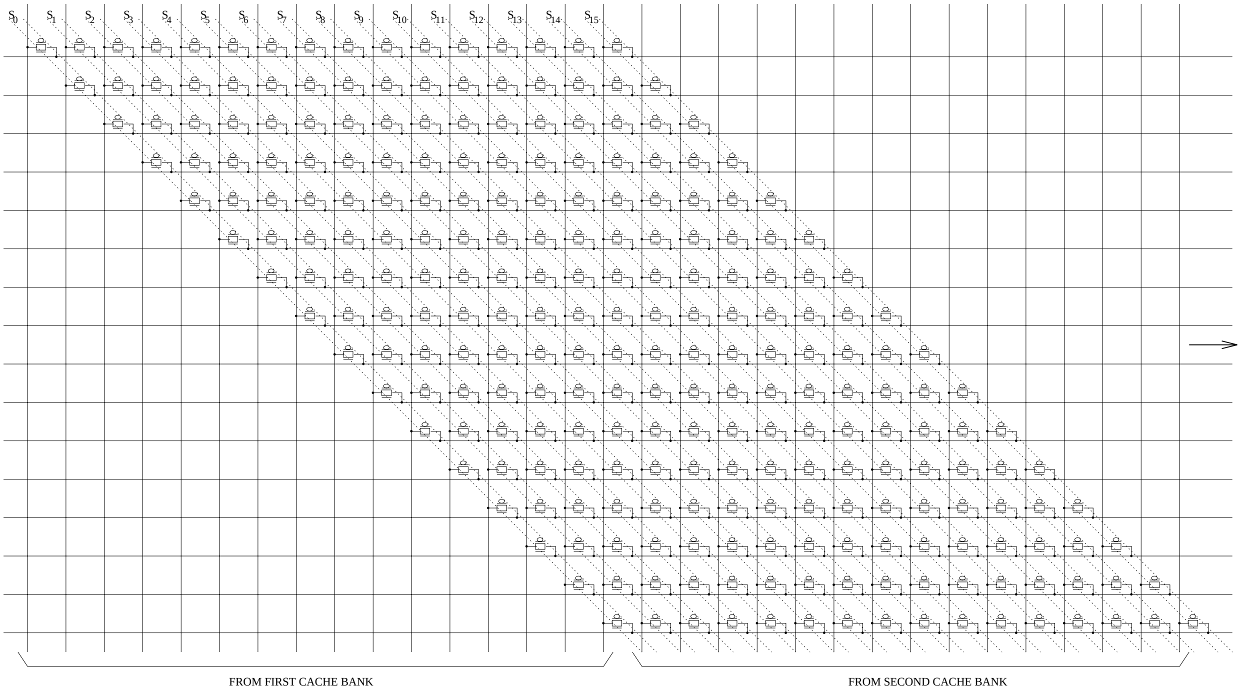


Figure 25: Fast left-shifter design for the base instruction caches. This is a 1-bit slice; hence there are 32 of these blocks, one for each bit in an instruction word. A complete shifter requires $(256)(32) = 8,192$ transmission gates. The 31 input bits (16 bits from first cache bank, 15 bits from second cache bank) run vertically, and the 16 output bits (which feed the output instruction latch) run horizontally. Both positive and negative shift control signals are shown as dotted diagonal lines. The shifter can shift instructions left anywhere from 0 to 15 positions in 1 transmission gate delay with reasonable loading.

Note that the logic in Section A.1 (which matches multiple predictions to their corresponding branches) does not exist in the original CB design of [1]. This is because simple 2-bit branch predictors [18] are embedded with every branch in the BTB. This simple branch predictor is a serious performance handicap, however.

- The collapsing buffer crossbar network is driven by the valid instruction bit vectors. That is, the collapsing buffer stage depends on the BTB stages. This dependence forces a distinct stage after the instruction cache.
- The collapsing buffer stage is a longer path than expected because the actual routing signals are not available at the beginning of the cycle. First, complex logic is needed to convert the valid instruction bit vectors into the switch signals. Then the switch signals must be fanned out to the bus drivers. Only then can the instructions propagate through the crossbar network.

A.4.3 Branch Address Cache

This section contains only comments rather than detailed logic descriptions, due in part to complexity, but foremost because it is unclear how to design the instruction cache to support this fetch mechanism.

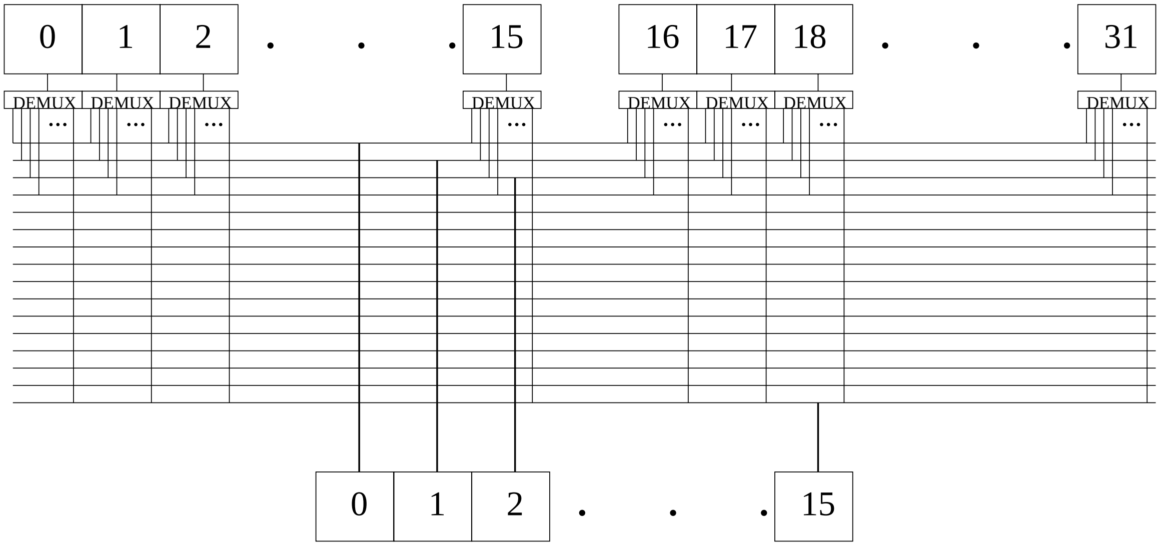


Figure 26: The collapsing buffer stage of the CB fetch mechanism, implemented as a bus-based crossbar.

The BAC requires logic similar to the collapsing buffer after the instruction cache, since useless instructions between branches and their targets must be removed. However, the logic is even more complex due to the large number of cache banks (8 are used in this paper and in [2]). Even without that hurdle, there are several major problems with this fetch unit:

- We do not have a *valid instruction bit mask*, nor has anything been mentioned in [2] about storing lengths of basic blocks. Somehow the logic after the instruction cache needs to know where all valid instructions are, yet only the starting location of each contiguous block is known. A BAC-like design in [4] does discuss providing length information, among other basic block *attributes*.

- It is not clear how to perform the interchange function. There are many more permutations than with just 2 cache banks.

- Even if the previous two problems are addressed (so that we have enough control information and the banks are presented in sequential order), we are back to the same collapsing buffer latency issues.

A.5 BAC Implementation Issues

A.5.1 Underflow and Overflow

The Branch Address Cache scheme as proposed in [2] introduces subtle issues which complicate implementation alternatives. The fetch unit has an upper limit on the number of instructions which can be supplied in a cycle. Two issues – here called *underflow* and *overflow* – arise from the interaction between the BAC and this instruction fetch limit.

Underflow is a scenario where (1) the fetch limit has not been reached, (2) it is *possible* to fetch more instructions in the same cycle, *BUT* (3) strict adherence to the spirit of the BAC dictates that fetching the additional instructions be delayed a cycle. (Hence the term underflow – providing less instructions than could be provided.) *Overflow* is a scenario where the BAC causes the fetch unit to *exceed* the instruction fetch limit. (Hence the term overflow – exceeding the hard limit.)

Refer to Figure 27 for the description of underflow. In cycle 1, there is one address A which is used to index both into the instruction cache and the BAC. Suppose that branch $b1$ is predicted not taken and branch $b2$ is predicted taken. The BAC indicates a hit and provides at least two addresses corresponding to the predicted path: B and Y. The problem has to do with branch $b1$. Since it is predicted not taken, two alternatives exist:

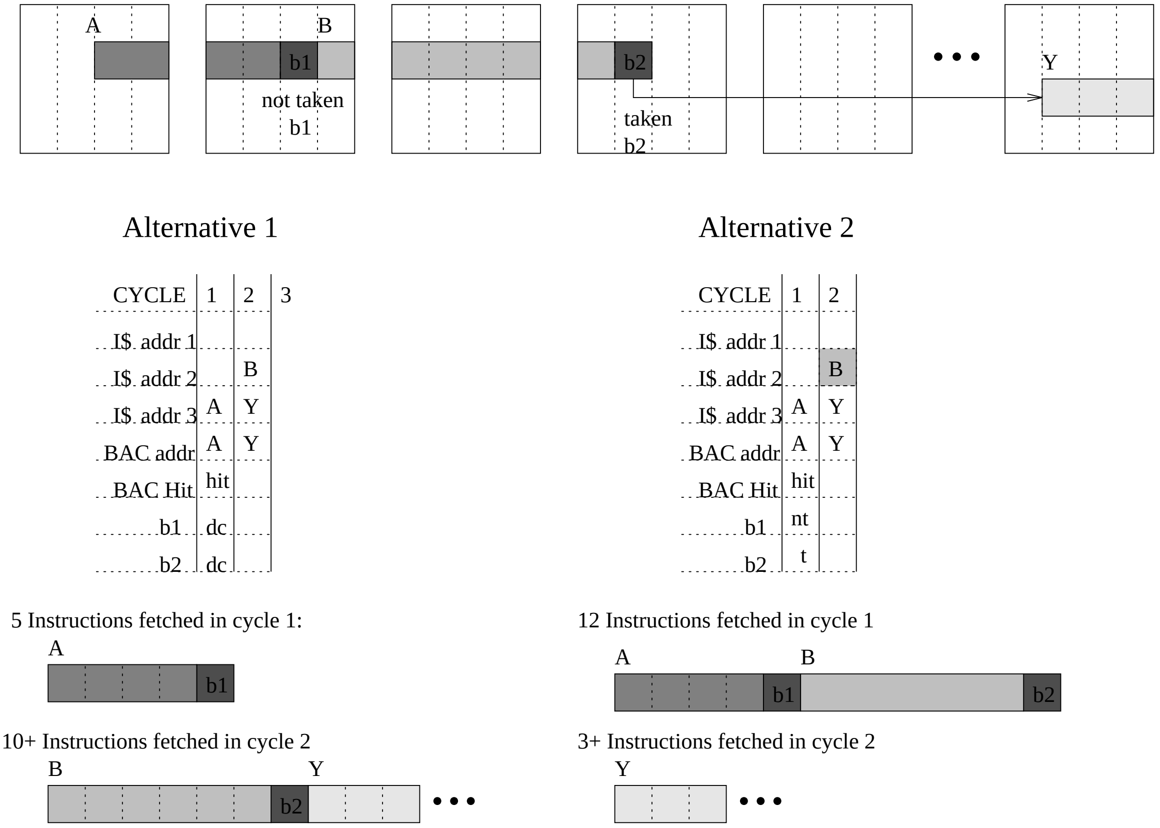
- *Alternative 1*: Fetch only the first basic block (A) of instructions in cycle 1. Wait until cycle 2 to fetch beyond branch *b1 even though branch b1 is predicted not taken*, using the address B provided by the BAC.

- *Alternative 2*: Fetch both basic blocks A and B in cycle 1, since branch *b1* is predicted not taken. *This requires that address B from the BAC not be used in cycle 2.*

Alternative 1 is the scheme of choice from a complexity point of view, since addresses provided by the BAC are always used (ignoring bank conflicts – more on this later). Alternative 2, on the other hand, introduces special cases which need to be handled differently. Specifically, the outcomes of branches as well as the number of addresses provided by the BAC have to be examined to make correct decisions. This leads to a less uniform design. The performance implications of using Alternative 1 over Alternative 2 are not clear: while Alternative 2 tends to provide more instructions in the first cycle, Alternative 1 "catches up" in the next cycle.

There are two conditions which characterize underflow. First, there must be a hit in the BAC. Second, the branch following the address which hit in the BAC must be predicted not taken. The issue is whether or not to use the fall-through address of the not taken branch, leading to the two alternatives previously described.

The problem of overflow is depicted in Figure 28. Suppose that the fetch unit is limited to a maximum bandwidth of 16 instructions per cycle. As in the previous example, during cycle 1 there is one address A which is used to index both into the instruction cache and the BAC. The BAC indicates a hit and provides three addresses corresponding to the predicted path: B, C, and Y. The basic blocks starting at these addresses are fetched in cycle 2. However, because the 16th instruction lies *within* basic block Y, part of that basic block is not fetched in cycle 2; these instructions are marked with the lighter shading in the diagram. The last address Y is used as an index into the BAC during cycle 2. This is where a potential problem exists. If the index Y hits in the BAC, then the next fetch address (e.g. X) will skip over the last few instructions of basic block Y. There are two alternative solutions to this problem:


A
B
b1
not taken
b1
b2
taken
b2
Y
Alternative 1
Alternative 2
5 Instructions fetched in cycle 1:
A
b1
10+ Instructions fetched in cycle 2
B
Y
b2
12 Instructions fetched in cycle 1
A
B
b1
b2
3+ Instructions fetched in cycle 2
Y

Alternative 1

CYCLE	1	2	3
I$ addr 1			
I$ addr 2		B	
I$ addr 3	A	Y	
BAC addr	A	Y	
BAC Hit	hit		
b1	dc		
b2	dc		

Alternative 2

CYCLE	1	2
I$ addr 1		
I$ addr 2		B
I$ addr 3	A	Y
BAC addr	A	Y
BAC Hit	hit	
b1	nt	
b2	t	

Figure 27: Underflow.

- *Alternative 1*: on a hit, the BAC may only provide n addresses if it is guaranteed that n full basic blocks can be fetched within the bandwidth limit (ignoring bank conflicts). The one exception to this rule is for $n = 1$: in this case, a full basic block need not be guaranteed. In the above example, indexing into the BAC with address A will yield only two basic block addresses: B and C. Basic block Y is not provided since it does not fit within the 16 instruction fetch limit.

- *Alternative 2*: The above restriction does not apply. Therefore, the hardware must detect overflow of the last basic block and fetch the leftover instructions.

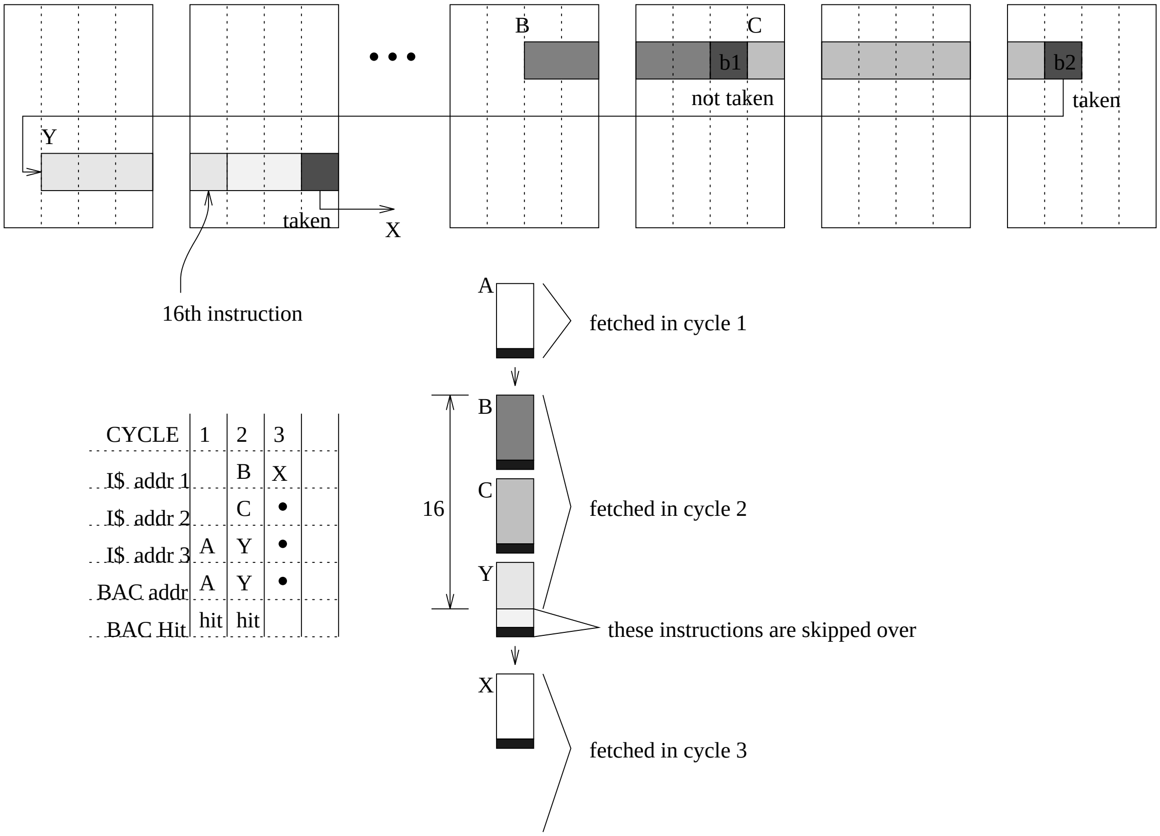
Alternative 2 seriously complicates the design since it deviates from basic block boundaries. In the example, the fetch unit has to generate an *extra* address to pick up the leftover instructions of basic block Y, which could introduce new bank conflicts as well as aggravate the overflow problem in cycle 3. Alternative 1 is a cleaner design, but may yield lower performance.

A.5.2 Content and Fill Issues

Each entry in the original BAC design stores the following information:

- Address tag.

- Three bits of state per branch in the tree, 1 valid bit and 2 bits identifying the branch type (conditional, unconditional, or return).



CYCLE	1	2	3	
I$ addr 1		B	X	
I$ addr 2		C	•	
I$ addr 3	A	Y	•	
BAC addr	A	Y	•	
BAC Hit	hit	hit		

Figure 28: Overflow.

- Target and fall-through addresses of each branch in the tree.

This information is enough to fetch a large trace of instructions from the cache to the decoders. However, the complexity of logic between the BAC and the bank address latches, and of the control logic driving the alignment network, may impact cycle time. If cycle time does become a concern, the design could be changed in several ways. First, at no extra hardware cost, bank conflicts can be determined ahead of time and only once – at the time branch addresses are entered into the BAC. In other words, any basic block addresses provided by the BAC are guaranteed to not conflict. Second, additional control information could be stored in each entry – also determined at the time the entry is filled. The extreme is to store precalculated bank addresses and alignment network control signals for each path through the tree. In the spirit of horizontal microcode, the information is fanned out directly to the datapath control points.

Another implementation issue is the number of concurrent fills supported. If an address misses in the BAC, an entry is allocated for the tree beginning at that address; the entry is filled with branch target and fall-through addresses as branches in the predicted path are encountered. While an entry is filling, the fetch unit continues to access the BAC. If another miss occurs before the first has completed, the hardware can either start another fill immediately, delay servicing the new miss until the first has completed, or ignore the miss altogether. Supporting multiple concurrent fills requires as many fill buffers.

A.6 Trace Cache Fill Logic

This section presents the major part of the fill logic required to accumulate traces for the TC. The design is intentionally incomplete; the intent is to give a feel for the kind of functions the logic must perform. Also, the datapath is not necessarily laid out for optimal timing. Figure 29 shows the design, which assumes a TC line size of 16 instructions and up to a maximum of 3 basic blocks.

The logic assumes the following latched inputs from the decoder each cycle:

- *incoming instruction latch* – holds up to 16 new instructions fetched from the instruction cache.
- *PC* – the starting address of the incoming contiguous block of instructions.
- *b* – the total number of branches in the incoming instruction latch.
- *p1, p2, p3* – the predicted directions of any branches in the instruction latch.
- *T1, T2, T3* – the target addresses of any branches in the instruction latch.
- *W1, W2, W3* – the size of basic block 1, the size of basic blocks 1 and 2 together, and the size of basic blocks 1, 2, and 3 together, respectively.

The central control – labelled as a PLA – monitors the progress of the fill and maintains relevant latches. Specifically, the following functions are performed:

- Reset all control and data latches when a signal indicates to start a new fill.
- Maintain the current length of the fill buffer, L.
- Maintain the current number of branches in the fill buffer, B.
- Update the *branch mask bits*.
- Update the *branch flags*.
- Detect when either (1) the number of branches equals or exceeds 3, or (2) the number of instructions equals or exceeds 16 (overflow bits). At this point, the “fill complete” signal is asserted and all latches are frozen, until the fill buffer information is committed to the TC.
- In the cycle that filling completes, the PLA controls what values are loaded into the *trace target address* and *trace fall-through address* registers.

In Figure 29, control is distributed via dotted lines from the PLA to control points in the datapath. The value of register L directly controls the shifting of new instructions into the appropriate position of the fill buffer.

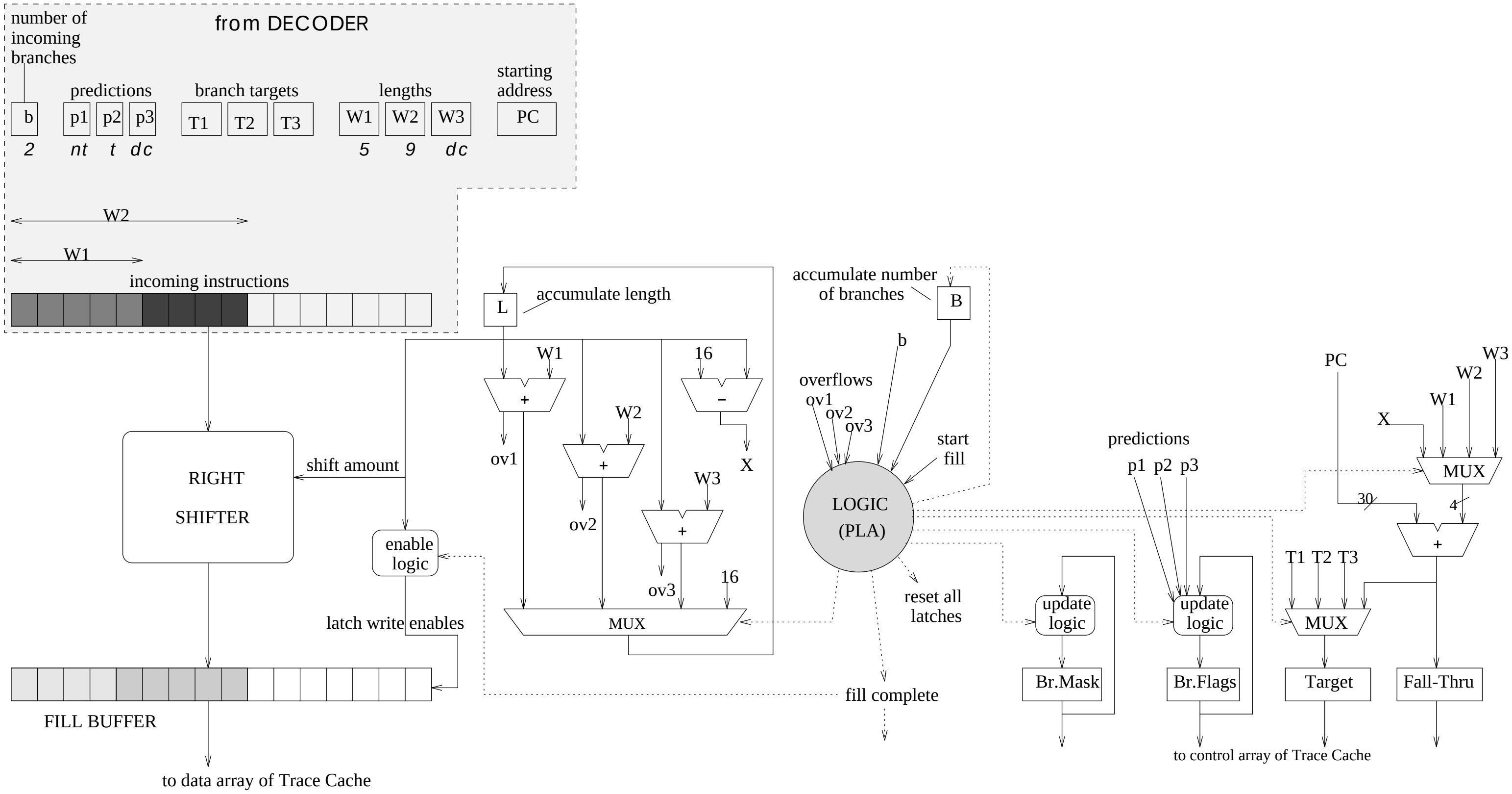


Figure 29: Major portions of the TC fill logic.