eXtended Block Cache

Stephan Jourdan, Lihu Rappoport, Yoav Almog, Mattan Erez, Adi Yoaz, and Ronny Ronen

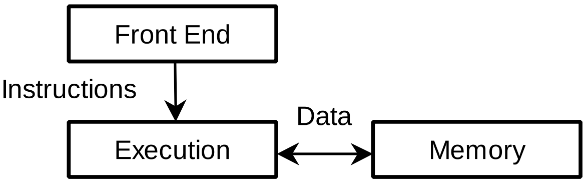
Intel Corporation
Intel Israel (74) Ltd., P.O. Box 1659, Haifa 31015, Israel.

Abstract

This paper describes a new instruction-supply mechanism, called the eXtended Block Cache *(XBC). The goal of the XBC is to improve on the Trace Cache (TC)* hit rate, *while providing the same* bandwidth*. The improved hit rate is achieved by having the XBC a nearly redundant free structure. The basic unit recorded in the XBC is the* extended block *(XB), which is a* multiple-entry single-exit *instruction block. A XB is a sequence of instructions ending on a conditional or an indirect branch. Unconditional direct jumps do not end a XB. In order to enable multiple entry points per XB, the XB index is derived from the IP of its ending instruction. Instructions within the XB are recorded in reverse order, enabling easy extension of XBs. The multiple entry-points remove most of the redundancy. Since there is at most one conditional branch per XB, we can fetch up to* n *XBs per cycle by predicting* n *branches. The multiple fetch enables the XBC to match the TC bandwidth.*

1. Introduction

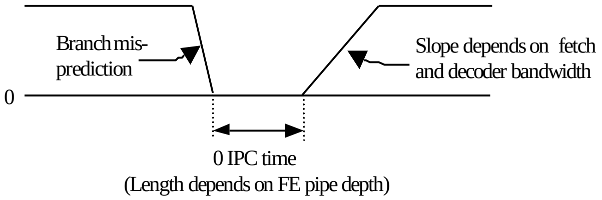
A processor may be viewed as composed of three blocks: *frontend*, *memory* and *execution*:



The goal of the frontend is to supply valid decoded instructions to the execution core, with *low latency* and *high bandwidth*. The frontend needs to predict which instructions to fetch next, decode them, and move them on to the renaming unit.

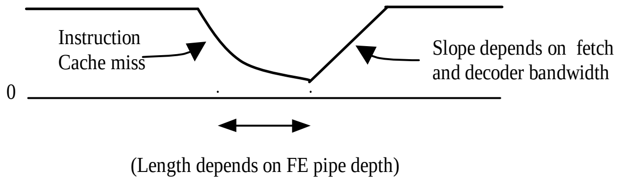
Program execution may be viewed as being comprised of three alternating phases: *steady state*, *transition*, and *stall*. Steady state execution is defined as execution of instructions in the absence of *disruptive events*. A disruptive event is either (i) a branch misprediction, (ii) an instruction cache miss, or (iii) a data cache miss. TLB misses has a similar effect as cache misses. There are other disruptive events, such as exceptions, which cannot be avoided and are not dealt with here. A stall phase is the period of time in which the machine is halted from execution (due to a long-latency disruptive event). A Transition phase is the period of time immediately following or preceding steady state execution. Here is a closer look at each of the three disruptive event types:

- **Branch mispredictions** cause the instruction window to be flushed. This lead to a steep degradation in IPC (the branch may be advanced due to out-of-order execution) followed by a stall, until instructions from the correct path are fetched and decoded.

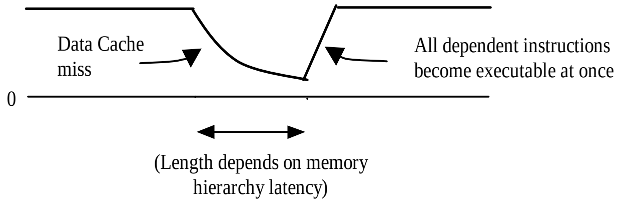


  In this case, the frontend is required to supply valid instructions to be executed as fast as possible.

- **Instruction cache misses** cause the instruction window to be gradually emptied - retired instructions exit, but it takes time until new instruction are fetched and decoded to fill it.



- **Data cache misses** cause the instruction window to be filled with instructions that are dependent on the missing data, which in turn cause a degradation in IPC. Once the missing data is retrieved, all dependent instructions may be executed.



Executing instructions out-of-order hides part of the penalty associated with the stalls incurred by the disruptive events. Instructions that are independent of the missing data, may pass the dependent instructions and still be executed.

[Mich99] describes analytically some of these transitions. As a rule of thumb, the CPU spends about 50% of the time in steady state execution, 30% in transition phase, and is stalled for 20% of the time. During steady state execution we need a high-bandwidth front-end structure, while latency is less important (hidden by pipelining). However to minimize the time spent in transition, a fast ramp-up is required, which implies low latency. This shows the importance of a high-bandwidth low-latency front-end structure.

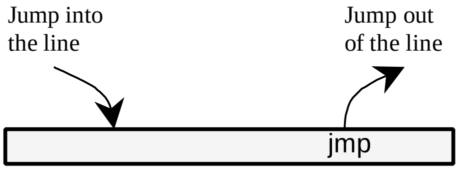
The rest of the paper is organized as follows. In Section 2 we describe four of the known frontend alternative: instruction cache, decoded cache, trace cache, and block-based trace cache. In Section 3 we describe the XBC in detail, including the definition of a XB and the XBC structure and algorithms. In Section 4 we bring experimental results, comparing the XBC to the TC, and exploring some of the XBC design alternatives. Finally, we conclude in Section 5.

2. Frontend Alternatives

2.1 Instruction Cache

The traditional frontend solution is based on an *instruction cache* (IC). The basic data unit of an IC is the cache line, which is a sequence of consecutive undecoded instructions, typically 16 to 64 bytes long. The IC is simple to implement, and has a high hit rate. However, it fails to meet both the low latency and the high bandwidth requirements.

The IC bandwidth is low since a program may jump into and out of a cache line. Thus, the bandwidth of a single-ported IC is limited to a single block of consecutive instructions. A multi-ported IC is needed in order to provide more than one block per cycle. Several complete proposals have been introduced to increase the limited bandwidth of an IC using techniques like multiple branch predictions while multi-porting the IC [Yeh93, Cont95, Dutt95, Sezn96].



The IC latency is high since instructions must be decoded before being supplied to the execution core. This is especially true for IA32 instructions, which have a non-uniform format. Moreover, IA32 instructions are *variable length* instructions, which makes parallel decoding difficult, and further reduces bandwidth. A pre-decoded cache holding instruction boundaries reduces latency and improves bandwidth.

2.2 Decoded Cache

In order to facilitate out-of-order execution and increase parallelism, the execution core of most current IA32 processors does not process the original instructions. Instead, during the decode stage IA32 instructions are translated into fixed-length RISC-like *micro-instructions* (uops). Thus, the decode stage latency is increased further to include both variable length decoding and instruction translation, and caching or buffering the output of the decoder may be beneficiary.

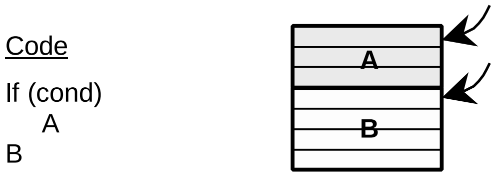
However, since the number of uops in each IA32 instruction varies, an addressing problem arises. In the IC the indexing into the cache is done by the instruction-address. In the decoded-cache if we wish to use such a simple index, fragmentation is likely to occur, since each line must provide enough space for the maximum number of uops in each instruction.

One solution to this problem was presented in [Intr92]. The mechanism here deals with storing variable-length instructions and not uops, but is also applicable to the second case. The idea is to keep a single instruction in each line of the decoded-cache and use the instruction-address as an index. This solution is very costly in terms of the tag array requirements. To adapt it to a decoded cache that stores uops, each line must now contain a fixed amount of uops. We will not go into details as to how such a cache may be constructed, but in essence it is similar to the idea of the block-cache structure of [Blac99].

To summarize, the decoded-cache resolves some of the latency problems associated with the IC, however it still suffers from the bandwidth problems of the IC, and its hit rate is slightly reduced (due to fragmentation).

2.3 Trace Cache

The trace cache (TC) aims to reduce the decoding latency and increase the supplied bandwidth. The basic unit of the TC is a *trace*, which is a sequence of dynamically executed instructions, which may originally reside in non-contiguous portions of the program memory. A trace starts with any instruction, and ends when one of the following trace end-conditions is met: size limit (number of instruction per trace), instructions such as indirect branches and returns, and a maximum number of conditional branches. The last condition is required due to the limited branch prediction bandwidth (next-trace prediction partially circumvents this limitation [Jaco97]). There are several variations of the TC, most TC schemes build traces which are single-entry multiple-exit. This causes *redundancy* in the TC (several copies of an instruction exist in the TC). For example look at the following code:



```
Code
If (cond)
   A
B
```

This code produces two possible traces: (i) B and (ii) AB. Both are stored in the TC. This causes the instructions (or uops) in B to be recorded twice (once for each entry point). Since traces can start on any instruction, B may be recorded in many more copies due to alignment discrepancies. Instruction redundancy is the average number of times each uop appears in the TC.

The original TC model was proposed in industry [Pele94], but was not well known in academia. The basic TC model that has been adopted by academia [Rote96, Frie97] is organized as a set-associative cache where each line holds a single trace of up to 16 instructions with a maximum of 3 branches. One limitation of these models is the lack of *path associativity*, which means that the TC cannot hold two traces that start with the same IP (since a trace is defined by its starting instruction alone). The model proposed in [Jaco97] allows for path associativity and also circumvents the branch limit. This is done by defining traces according to the starting IP and according to an encoding of the path.

A TC-based frontend operates in one of two modes*: build mode* or *delivery mode*. While in build mode, the frontend fetches IA32 instructions from the instruction cache, decodes them into uops and sends the uops to execution. In parallel, the frontend also builds traces, and stores them in the TC. Once a trace is built, the frontend performs a TC head lookup for the next instruction. Upon a hit, the frontend switches to delivery mode. In this mode, uops are fetched directly from the TC. The TC delivers much higher bandwidth than a regular IC with lower latency. However this is done at the expense of the hit rate due to both the fragmentation and redundancy. A low hit rate implies that a high percentage of instructions are fetched in build-mode, without utilizing the high bandwidth and the low latency of the TC.

2.4 Block-based Trace Cache

Recently the block-based trace cache (BBTC) [Blac99] was suggested as a variation of the conventional trace cache concept. The BBTC records traces of block pointers instead of instructions. The pointers are then used to access a decoded cache to retrieve the instruction blocks. The BBTC shifts the redundancy from instructions to block pointers, which reduces the negative impact of redundancy. However, fragmentation is likely to be larger due to the finer storage granularity.

The BBTC can be seen as a special case of the fetch mechanism previously introduced in [Dutt98]. This fetch mechanism records tree-like sub-graph elements comprised of block pointers. A path in the tree is predicted and the pointers are used to access a multi-ported instruction cache to retrieve the instructions. The differences lie in the recording structure (tree versus path) and the type of cache.

3. XBC: Structures and Algorithms

Our goal is to design an instruction supply mechanism, which improves on the TC hit rate, while achieving the same bandwidth and latency. In order to achieve this goal, we design a new frontend structure, called the *eXtended Block Cache* (XBC). The better hit rate of the XBC is achieved by having it redundancy free, and with lower fragmentation.

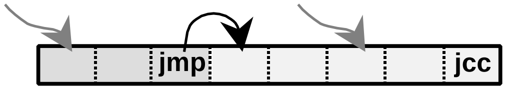
In this section, we describe the XBC structure and algorithms. The XBC is comprised of three main units:

1. The cache structure itself, holding the XBs.
2. The XBTB, which predicts the next *n* XBs in the XBC.
3. The Fill Unit, XFU, which builds XBs into the XBC.

3.1 The Extended Block

The basic unit saved in the XBC is the extended block (XB), which is a sequence of instructions, ended by a conditional or an indirect branch.

A XB features multiple entry points and a single exit— exactly the opposite of a trace. A XB is entered either at its head, or anywhere in the middle. Since there are no conditional or indirect branches within a XB, **once a XB is entered, it can only be exited at its end**. The tag and the index of a XB are derived from the IP of its ending instruction. This enables each XB to have multiple entry points.



Unconditional branches, which redirect the flow towards a single location, do not end XBs. On the other end, both conditional and indirect branches (including returns) end XBs since they may branch to multiple locations.

In order to keep the XBC latency small, as in the TC, the instructions in the XB are also kept in some decoded form (e.g., uops). To summarize, the XB end conditions are: conditional branches, indirect branches, return instructions, and a quota limit of 16 uops per XB.

Our goal is to supply more than a single XB per cycle. Since conditional and indirect branches end a XB, we need to predict only a single branch per XB. This implies that with a prediction bandwidth of *n* predictions per cycle, we can predict and fetch *n* XBs per cycle.

In order to maximize the XBC bandwidth for a given prediction bandwidth, longer XBs may be built using *conditional branch promotion* [Pate98]. This scheme enables the promotion of a conditional branch to be treated as an unconditional branch, once it is found that the branch is monotonic (≥ 99% biased). When we find that a XB ends with a branch which is highly biased, we promote the XB, and join it with the following XB. Figure 1 shows the length distribution of four types of instruction blocks, all limited by a maximum length of 16 uops: basic block – ends with any jump, XB, XB with promotion, and dual XB (two consecutive XBs). The average length of a basic block in IA-32 code is 7.7 uops, of a XB – 8.0 uops, of a XB with promotion – 10.0 uops, and of a dual XB – 12.7 uops.

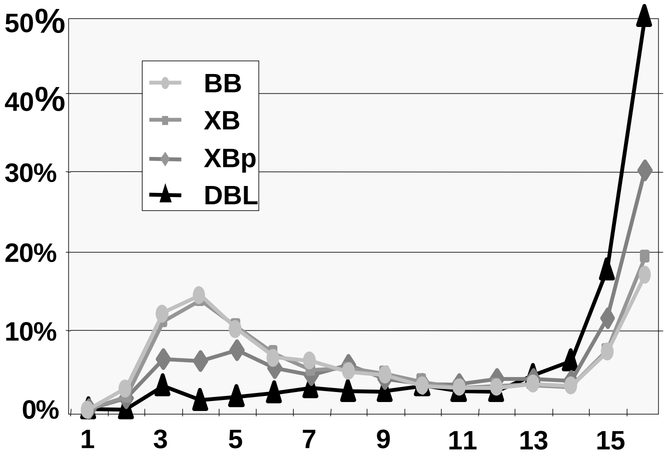


Figure 1. Length Distribution

3.2 The XBC Cache Structure

The XBC is organized as a banked cache. Each bank has its own decoder, so in a given cycle a different set may be accessed in each bank. Instructions may be received from all banks in parallel in the same cycle. In addition, each bank may be implemented as a true set-associative cache. The bank structure supports the storing of variable length XBs (while minimizing fragmentation), and fetching multiple XBs per cycle.

The XBC supports variable length XBs, while minimizing fragmentation by allowing a XB to spread over one or more banks. We use a 4-bank structure with a total of 16 uops, which is the maximum fetch width.

The XBC supports fetching multiple XBs per cycle by enabling instructions to be received from all banks in the same cycle. If two consecutive XBs are stored in non-overlapping banks, both can be fetched in the same cycle. If a bank conflict does occur, only part of the second XB can be fetched.

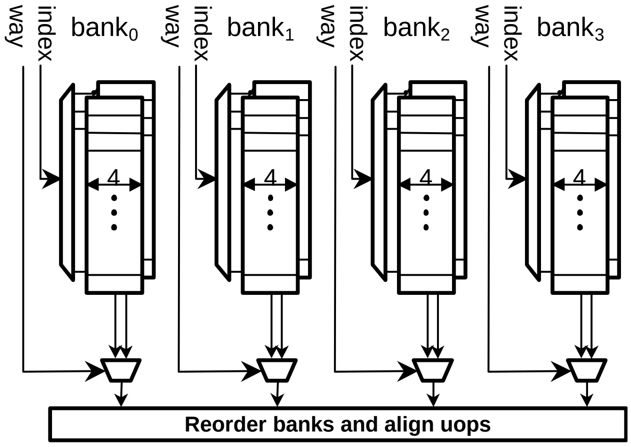


Figure 2. The XBC Data Array.

The average XB length (without promotion) is 8.5 uops. As a result, a 16-uop set is ≤2 XBs wide. This implies that a XBC set can hold less than 2 XBs on average. If two such XBs are mapped to the same set, thrashing will occur. The solution is to use 2-way set-associative banks. Figure 2 shows the XBC data array.

The XBC tag array (directory), shown in **Figure 3**, holds tags, and performs the tag match. Tags are virtual, so no address translation is needed prior to the XBC access.

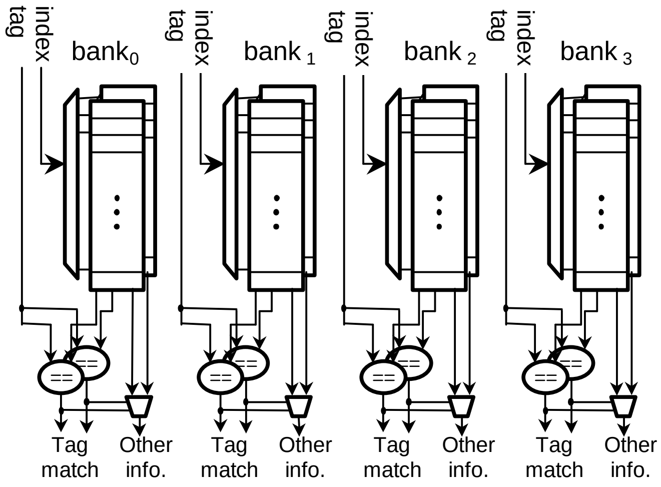


Figure 3. The XBC Tag Array

Each bank has a single decoder for accessing the two ways within the bank. The tag of *bank*$_i$, *tag*$_i$, is matched with the tags of both *bank*$_i$ ways. The two tag-match signals control the way multiplexer, and are also used to validate the access information supplied by the XBTB.

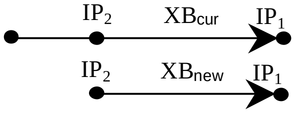
Notice that both the XB set and the XB tag are derived from the IP of the XB ending instruction. A bank that holds the end of a XB is called *the primary bank*. The XB tag is used for tagging all the banks holding the current XB. All the banks which hold a XB are numbered, according to their order: the primary bank is numbered 00, the preceding bank is numbered 01, etc. The number-field is part of the XBC tag array.

3.3 XB Build Algorithm

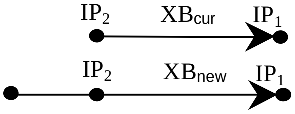
In this section we describe the XB build algorithm. As will be shown, this algorithm prevents redundancy in the XBC.

If the XBC lookup fails, the XFU starts building a new XB, XB$_{new}$. As the decoder supplies uops, the XFU inserts the uops into the fill buffer, until one of the end-of-XB conditions is reached. At this stage, the XFU performs a XBC lookup for XB$_{new}$. If no match occurs, XB$_{new}$ is stored in the XBC, an entry is allocated for XB$_{new}$ in the XBTB, and the XBTB entry of the previously executed XB is updated to point to XB$_{new}$. Since XB$_{new}$ end-IP uniquely identifies it, none of XB_{new}'s uops can appear in an existing XB, complying with the no-redundancy requirement.

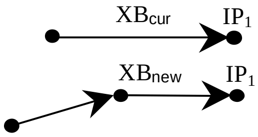
There are, however, three cases in which XB_{new} tag may match the tag of an existing XB, XB_{cur}:



(1)



(2)



(3)

For example, let us look at the following code:

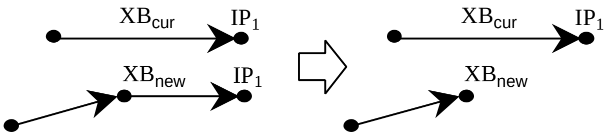
```
        A …. Goto Lc
L:      B
Lc:     C
        D
```

The three cases are: (1) XB_{cur} = BCD and XB_{new} = CD, (2) XB_{cur} = CD and XB_{new} = BCD, and (3) XB_{cur} = BCD and XB_{new} = ACD.

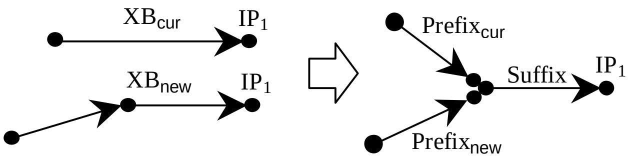
In the first case XB_{cur} contains XB_{new}, in the second case XB_{new} contains XB_{cur}, and in the third case, XB_{new} partially overlaps XB_{cur}—they have the same suffix (and thus the same XB tag), but a different prefix.

In the first case, there is no need to update the XBC, just to update the XBTB. In the second case, XB_{cur} is extended with the uops from XB_{new} prefix. If there are enough free uops in the bank holding XB_{cur}, the new uops are simply added to XB_{cur} on its head. If there are not enough free uops, another bank needs to be allocated for the XB, in which case, the replaced bank needs to be evicted from the XBC.

In the third case (same suffix, different prefix), there are two possible solutions complying with the no-redundancy requirement. The first solution is to store in the XBC the prefix of XB_{new} as an independent XB. That is, although this prefix ends with an unconditional jump, it defines a XB:



The drawback of this solution is that we get two short XBs instead of a single long XB, reducing bandwidth. This solution also needs more entries in the XBTB. In the second solution, XB_{cur} and XB_{new} are viewed as a single "complex XB", with two different prefixes:



The complex XB is stored in (at least) three banks containing $Prefix_{cur}$, $Prefix_{new}$, and Suffix. A XB is now identified by a tag and by a mask vector. The mask vector defines the banks that needs to be accessed. For example, assume that $Prefix_{cur}$ is stored in $bank_2$, $Prefix_{new}$ in $bank_0$, and Suffix in $bank_3$:

$bank_0$	$bank_1$	$bank_2$	$bank_3$
$Prefix_{cur}$	*f*	Suffix	$Prefix_{new}$

If we need to access XB_{new}, the mask vector is 0011. The order field will mark $bank_2$ with 00, and $bank_3$ with 01. Since $Prefix_{cur}$ and $Prefix_{new}$ are never needed together, when set-associative banks are implemented, it is better to put them in different ways of the same bank.

3.4 Storing Uops in Reverse Order

When an existing XB is extended, it can only be extended at its beginning. In order to extend an existing XB we need to move all its uops, to make room for the new uops. Since the existing uops move, the pointers in the XBTB become stale (see in Section 3.5).

In order to solve this problem, and to simplify the extension of existing XBs, we store the uops of a XB in reverse order, i.e. starting from the last instruction to the first instruction. In this way, when an existing XB is extended, its uops need not be moved.

Notice that since the XB IP is derived from the IP of the ending instruction, extending the XB does not change the XB IP.

3.5 The XBTB

In the current structure the XBTB and the XBC are tightly coupled: the XBC can be accessed only via the XBTB. That is, the information needed to track the next XB in the XBC is not found in the XBC itself. Once a branch is resolved, we know the IP of the target instruction of the branch. Since the XBC is a multiple-entry structure, the XB_IP is derived from the end-branch of the XB (as opposed to a TC, which is single entry, and therefore a trace tag is derived from the IP of the entry-point of a trace). This means that we cannot lookup the target IP of a branch in the XBC. Therefore, in case of a XBTB miss, or a mis-fetch, we must switch to build mode.

Once in build mode, it is possible to switch back to delivery mode only when the XB that is currently being built is already in the XBC. In that case there is both a XBC hit and a XBTB hit. Once there is a XBTB hit, the XBTB provides the next XBs in the XBC, so we can switch back to delivery mode.

The XBTB provides the information required for locating the next XB, XB_{next}. This information includes:

– XB_IP: The IP of XB_{next} (the IP of XB_{next} ending instruction); it defines both XB_{next} index and tag.

– BANK_MASK: a masking vector, indicating the banks in which XB_{next} resides.

– OFFSET: The number of uops counted backward from the end of XB_{next}; it defines where to enter XB_{next}.

Since we want to fetch more than a single XB per cycle from the XBC, the XBTB provides pointers to the next two XBs in each cycle. The XBTB is comprised of the following units (Figure 4):

1. XBTB: keeps information on all XBs that are stored in the XBC, and predicts the next XB for XBs that are ended by a conditional branch.
2. XBP: predicts the direction of conditional branches.
3. XiBTB: predicts the next XB for XBs that are ended by an indirect branch that takes more than a single target.
4. XRSB: predicts the next XB for XBs that are ended by a return instruction.

In an instruction cache, the fall through instruction follows the current instruction, and thus need not be predicted. However within the XBC, the fall through XB does not follow the current XB. Therefore, for a XB which ends with a conditional branch, the XBTB must hold pointers both to the taken path XB, XB_{tkn}, and to the fall through XB, XB_{nt}. The XBP provides the prediction for the taken/not-taken direction of conditional branches. This prediction is used for selecting between XB_{tkn} and XB_{nt}.

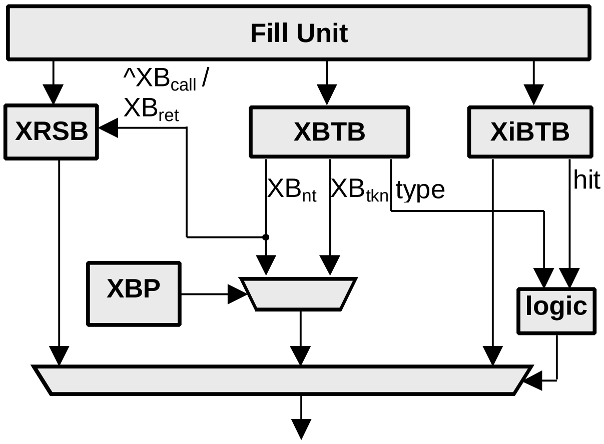


Figure 4. XBTB Sub-Units: XBTB, XBP, XiBTB, XRSB.

The XRSB predicts the next XB for XBs that end with a return instruction. A XBTB entry associated with a XB ended by the corresponding call, XB_{call}, records both a pointer to the first XB in the procedure, XB_{func}, and a pointer to the XB following the return instruction, XB_{ret}. When the call is executed for the first time, a new XBTB entry in allocated for XB_{call}. A pointer to XB_{call} XBTB entry is pushed into the XRSB, so later on, its XBTB entry can be updated with a pointer to XB_{ret}. Once XB_{func} is built, the taken pointer in XB_{call} XBTB entry is updated to point to XB_{func}. Once the function returns, the XRSB is popped to get XB_{call} XBTB entry. When XB_{ret} is built, the not-taken pointer in XB_{call} XBTB entry is updated to point to XB_{ret}. Next time the call is executed, the XBTB hits for XB_{call}. The taken pointer provides XB_{func}, which is used for accessing the XBC, and the not-taken pointer provides XB_{ret}, which is pushed into the XRSB. Once the function returns, the XRSB is popped to retrieve XB_{ret}.

Although there is no need to save the next XB for XB_{ret} in the XBTB (since it is provided by the XRSB), there is still a need to allocate a XBTB entry for XB_{ret} to record its type as return.

3.6 Accessing The XBC

In this section, we describe the interaction between the XBC and the XBTB. Figure 5 shows the entire XBC structure.

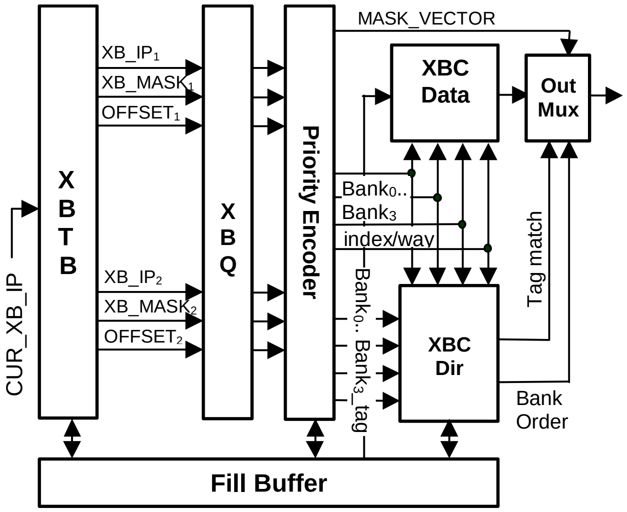


Figure 5. XBC Structure: the Whole Picture.

First, the current XB_IP is supplied to the XBTB. The XBTB performs a lookup according to XB_IP, and returns XB_IP, BANK_MASK, and OFFSET, that are needed to locate the next two XBs.

While the XBTB provides 2 XBs per cycle (on a XBTB hit), the XBC provides a variable amount of XBs due to both the variable length nature of the XBs, and due to bank conflicts. Thus, we need to decouple the XBTB from the XBC, as in [Rein99]. This is done by the XBQ.

A priority encoder receives the information from the XBTB, and determines (i) which set should be accessed in each bank in the XBC and with what tag, and (ii) what is the combined mask vector (for both XBs).

Example:

Assume that the first XB, XB_1, resides in $bank_0$ and in $bank_3$ of set 23, and that the second XB, XB_2, resides in $bank_2$ and in $bank_3$ of set 15. Further assume that $OFFSET_1 = 5$, $bank_0$ holds the prefix of XB_1, $OFFSET_2 = 7$, and $bank_2$ holds the prefix of XB_2:

	$bank_0$	$bank_1$	$bank_2$	$bank_3$
Set # 15	*f*	*f*	XB_2	XB_2
Set # 23	XB_1	*f*	*f*	XB_1
MASK_VECTOR	0 0 0 1	0 0 0 0	0 1 1 1	1 1 1 1

XB_MASK_1 is 1001, and XB_MASK_2 is 0011. Since XB_1 is the first of the two, it has priority over XB_2. As a result, $bank_0\_index = 23$, $bank_1\_index = \phi$, $bank_2\_index = 15$, and $bank_3\_index = 23$. Since $bank_2$ holds the prefix of XB_2, we can fetch it. The suffix of XB_2, which resides in $bank_3$, cannot be fetched in the current cycle (due to the bank conflict with XB_1). The combined mask vector from the priority encoder is a 16-bit vector, derived both from the combined mask vector described, and from the offset information.

The bank set numbers are provided both to the XBC data array and to the XBC directory. The bank data array is speculatively accessed, without waiting for the tag match. The data from the XBC data array is moved into the OUT_MUX, which also gets the mask vector from the priority encoder, and the tag match and order field from the directory. The OUT_MUX masks and reorders the uops. A tag mismatch (XBC miss), requires a switch to build mode. A tag mismatch occurs when following a pointer from the XBTB to a XB that was evicted from the XBC.

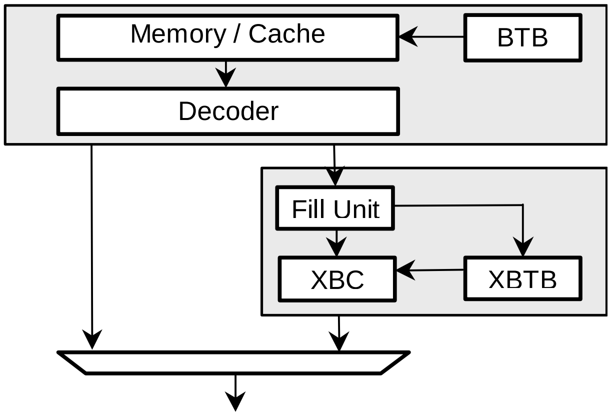


Figure 6. A XBC-Based Frontend.

Let us look at a XBC-based frontend (Figure 6). The upper part of this frontend is a traditional IC based frontend. The BTB points to the next instruction to be fetched from the IC, the appropriate cache line is fetched, the decoder decodes the relevant instructions within the line, and the decoded instructions, or uops are delivered to the execution. In parallel to delivering the uops to execution, the uops are processed by the fill unit and grouped into XBs. The fill unit stores the new XBs in the XBC and updates the XBTB. This mode of operation is referred to as *build mode*.

Once we finish building a XB into the XBC, we lookup the next XB in the XBTB. If there is a XBTB hit and a XBC hit, we switch to *delivery mode*, in which uops are supplied by the XBC. As long as we hit the XBTB and the XBC we remain in delivery mode.

3.7 Reordering and Aligning the Uops

Once we get the 4 lines from the banks, we first need to reorder them according to the order of both the XBs and the banks within the XBs. This is accomplished by a first mux layer (Figure 7).

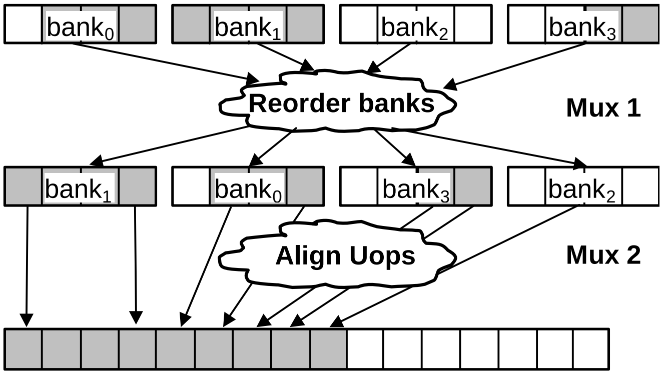


Figure 7. Reordering and Aligning the uops.

Now that we have the lines ordered, we are still not done since some of the lines may be only partially full. Therefore, we need to align the uops. This is done by a second mux layer. A careful design of this mechanism accomplishes the reordering and alignment in just one cycle.

3.8 Branch Promotion

In order to identify highly biased branches, each XBTB entry is augmented with a 7 bit counter, which indicates how biased the static branch that ends the XB is. Each time the branch is taken the counter is incremented by 1, and each time the branch is not-taken, the counter is decremented by 1. If the counter value is ≤1, it implies that out of the last $2^7 = 128$ times that the branch was executed, it was taken at most once, so it is at least 99.2% biased to not-taken. If the counter value is ≥126, the branch is at least 99.2% biased to taken.

If we decide to promote a branch, the XB ended by the branch, XB_0, and the XB usually following the branch, XB_1 are combined into a single XB, XB_{comb}. XB_{comb} ends with XB_1, thus its XB_IP is equal to the XB_IP of XB_1. This implies that XB_1 remains in its place, while XB_0 is copied to the same set as XB_1 and extends XB_1. XB_0 original location

LRU value is reduced to minimum. XB_0 XBTB entry is not discarded when XB_0 is promoted, and its not-taken pointer is updated to point at XB_{comb}. Notice that XB_0 may be pointing into XB_1, rather than to its beginning. In this case XB_{comb} is actually a complex XB. This also implies that in order to get the full benefit of branch promotion, the XBC needs to support complex XBs.

At any given time, we hold a pointer to the XBTB entry of the previously executed XB, XB_{-1}. When XB_0 is promoted, XB_{-1} is updated to point to XB_{comb}.

There may still be other XBTB entries that point to the original location of XB_0. If such an entry is accessed, and XB_0 has not been evicted yet from its original location, XB_0 uops are supplied from there. The XBTB entry is then updated to point to XB_{comb}, using the information in XB_0 XBTB entry. If, however, XB_0 has been evicted from its original location, XB_0 XBTB entry provides the new location of XB_0, XB_{comb}. Although this incurs a one-cycle penalty, it saves a switch to build mode.

The XBTB entry of a promoted XB serves two other purposes. First, the taken pointer field of this entry points to the non-frequent path. This saves a switch to build mode if the promoted branch takes the non-frequent path. Second, the counter field keeps gathering statistics on the promoted branch. Each time the promoted branch takes the other path, the counter is incremented. If the promoted branch starts misbehaving, it is de-promoted.

Branch promotion was suggested in the context of the TC in [Pate98]. Notice that in the TC a trace is built with no knowledge on the behavior of the branches within the newly built trace. As a result, many of the traces are wrongly built, in the sense that it would be better to build the trace along other paths than those originally chosen. Using branch promotion the way we use it here can be viewed as a more careful way to build traces.

3.9 Set Search

If a XB is evicted and then placed again, it is placed in the same set but not necessarily in the same banks. If different banks are used, the XBTB entries that point to the old location of the XB are erroneous. This problem can be solved with only a small penalty (cycle loss, but no switch to build), by searching the entire set. On a XBTB hit, XBC miss, the entire set is searched for the XB. If the XB is found, a new mask vector is calculated according to the offset.

3.10 XB Placement and Replacement

We use a LRU replacement policy among all the lines in a given set. The LRU policy also makes sure that we do not evict a line other than a head line (the first line) of a XB. There is no point in retaining the head line while evicting another line, since if we enter the XB in the head line, we will get a miss when we reach the evicted line. If, however, a head line is evicted, if we need to enter the XB in its middle, we may still avoid a miss. Since a non-head line is always accessed after a head line is accessed, its LRU will be higher, so it will not be evicted before the head line.

We employ a smart build-mode placement algorithm: when building a new XB into the XBC, the new XB is placed in banks such that it does not have any bank conflict with the previous XB (if possible). LRU ordering is maintained by switching the LRU line with the desired (non-conflicting) line before the new XB is placed. Set-search repairs the XBTB.

We also employ a dynamic (delivery mode) placement algorithm. If we find repeating bandwidth losses due to bank conflicts, the conflicting lines are moved to non-conflicting banks. For each XB we hold a counter which is incremented each time some of the uops of the XB are deferred to the next cycle due to a bank conflict (that is, there was an unutilized bank). If the counter reaches a predefined threshold, the conflicting lines are switched with other lines in non-conflicting banks. A line can be switched with another line, only if its LRU is higher, or if both gain from the switch.

4. Results

Results were obtained using a trace-driven stand-alone frontend simulator which models frontend structures based on either an IC, a TC, or a XBC. The frontend bandwidth is restricted by the renamer BW which is set to 8 uop/cycle.

Traces consist of 30 million consecutive x86 instructions translated into uops. Traces are representative of the entire execution, and they record both user and kernel activities. Results are reported for 21 traces grouped into 3 suites:

– SPECint95 (8 traces),

– SYSmark32 for Windows 95 (8 traces),

– and Games (5 traces of popular games).

We chose not to use the SpecFP benchmarks since the frontend bandwidth of both structures is very high for these traces.

The XBC was simulated for various sizes and associativity with a fixed 8K-entry XBTB. The TC was simulated as a 4 way set-associative cache, where each line holds a single trace of up to 16 uops with a maximum of 3 branches. We simulated a 16-bit history GSHARE predictor [McF93] for both the XBC and the TC.

Figure 8 shows both the XBC and the TC uop bandwidth for the same cache size (32K uops). Bandwidth is not affected much by total size, since it is defined only for hits (uops from delivery mode). As can be seen from the graph, the difference between the XBC and TC bandwidth is negligible.

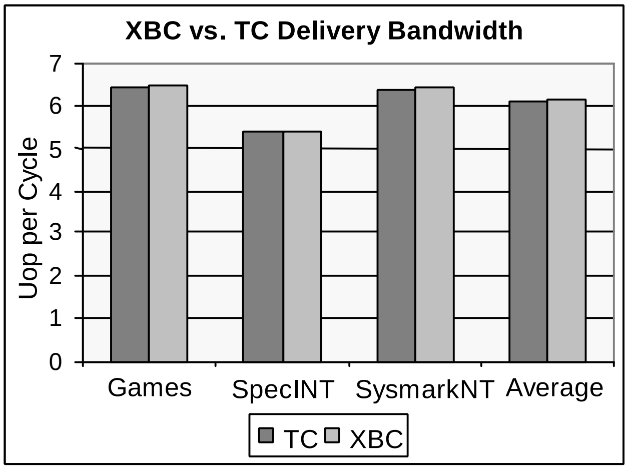


Figure 8. XBC versus TC bandwidth

Figure 9 compares the uop miss rate (percent of uops brought from the IC) of both structures as their size is varied. As expected the XBC has a much lower miss rate due to the reduced redundancy. The difference is most pronounced for the smaller cache size and diminishes as the number of uops stored increases. However, the reduction in the number of misses is ~29% for all cache sizes. That is, the XBC provides superior hit ratio for about the same bandwidth, regardless of the cache budget. In order to match the XBC hit rate, the TC should be enlarged by more than 50%.

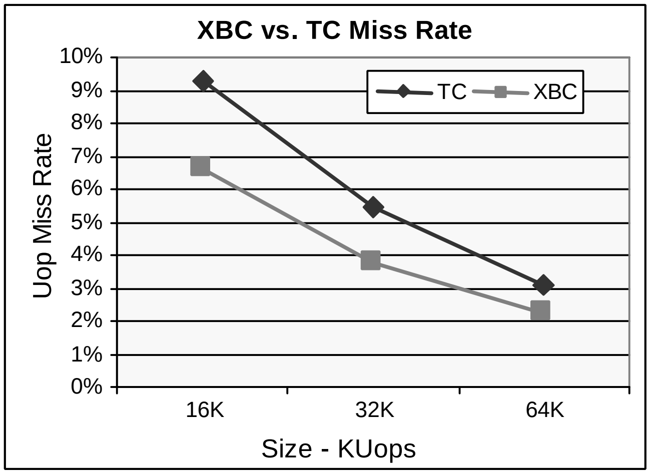


Figure 9. XBC vs. TC Miss Rate for Various Sizes

Figure 10 shows the average miss rate of the XBC and the TC as a function of the associativity. Both structures exhibit the well-known curve for increasing associativity. Moving from a direct-mapped implementation to a two-way set associative cache reduces the misses by about 60%. The improvement is smaller when increasing the associativity to 4. As in branch prediction, the reduction in the number of misses is very important due to the high penalty for fetching from the IC.

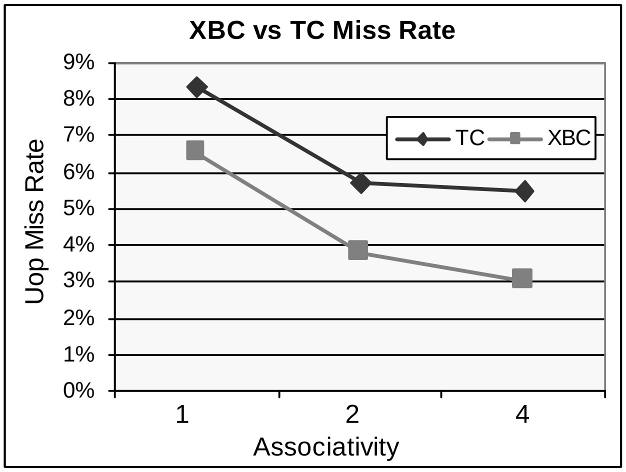


Figure 10. Miss Rate vs. Associativity

5. Conclusions

In this paper we presented the XBC, which is a new instruction supply mechanism. We showed that the XBC provides a better hit rate than the Trace Cache, while achieving the same instruction bandwidth with the same number of branch predictions per cycle performed. In order to match the XBC hit rate, a TC should be enlarged by more than 50%. The XBC incorporates both novel ideas such as ending IP indexing, reverse-order uop recording, and a two dimensional way-bank structure, as well as a few recently published ideas like branch promotion and decoupling.

Acknowledgements

We would like to thank Uri Weiser for his contribution in the initial definition of the XBC structure. We also want to express our gratitude to Moty Mehalel for his help with the reorder and alignment mux design options and the timing aspects.

Bibliography

[Blac99] B. Black, B. Rychlik, and J. Shen, "The Block-based Trace Cache," in Proceedings of the *26th International Symposium on Computer Architecture*, May 1999.

[Cont95] T. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," in Proceedings of the *22nd International Symposium on Computer Architecture*, June 1995.

[Dutt95] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," in Proceedings of the *28th International Symposium on Microarchitecture*, November 1995.

[Frie97] D. Friendly, S. Patel, and Y. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism," in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[Intr92] G. Intrater and I. Spillinger, "Performance Evaluation of a Decoded Instruction Cache for Variable-Length Computers," in Proceedings of the *19th International Symposium on Computer Architecture*, May 1992.

[Jaco97] Q. Jacobson, E. Rotenberg, and J. Smith, "Path-Based Next Trace Prediction," in Proceedings of the *30th International Symposium on Microarchitecture*, December 1997.

[McF93] S. McFarling, "Combining Branch Predictors," *Technical Note TN-36, DEC WRL*, June 1993.

[Mich99] P. Michaud, A. Seznec, and S. Jourdan, "Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors," in Proceedings of the *International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[Pate98] S. Patel, M. Evers, and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," in Proceedings of the *25th International Symposium on Computer Architecture*, June 1998.

[Pele94] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line*," U.S. Patent Number 5,381,533, Intel Corporation*, 1994.

[Rein99] G. Reinman, B. Calder, and T. Austin, "A Scalable Front-End Architecture for Fast Instruction Delivery," in Proceedings of the *26th International Symposium on Computer Architecture*, May 1999.

[Rote96] E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," in Proceedings of the *29th International Symposium on Microarchitecture*, December 1996.

[Sezn96] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-Block Ahead Branch Predictors," in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[Yeh93] T. Yeh, D. Marr, and Y. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," in Proceedings of the *7th International Conference on Supercomputing*, July 1993.