Two Level Bulk Preload Branch Prediction

James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, Anthony Saporito
IBM Systems and Technology Group
bonannoj, collura, dlipetz, brprasky, saporit @us.ibm.com Ulrich.Mayer@de.ibm.com

Abstract

This paper describes the large capacity hierarchical branch predictor in the 5.5 GHz IBM zEnterprise EC12 microprocessor. Performance analyses in a simulation model and on zEC12 hardware demonstrate the benefit of this hierarchy compared to a smaller one level predictor.

Novel structures and algorithms for two level branch prediction are presented. Prediction information about multiple branches is bulk transferred from the second level into the first upon detecting a perceived miss in the first level. The second level does not directly make branch predictions.

Access to the second level is limited when it is unlikely to be productive. The second level is systematically searched in an order that is likely to provide hits as early as possible.

On the workloads analyzed in the simulation model, measurements show a maximum core performance benefit of 13.8%. On the two workloads analyzed on zEC12 hardware 3.4% and 5.3% system performance improvements are achieved.

1. Introduction

Dynamic branch prediction reduces the performance degrading effect that conditional and redirecting instructions have on program flow. A highly accurate branch direction and target predictor is essential for good performance. However, performance of very large workloads is often limited by the capacity of the predictors more than the accuracy of the dynamic prediction algorithms. This paper presents a solution to the capacity problem.

The design described in this paper includes history-based predictors providing both the direction and target address of branches asynchronously, and most often ahead of instruction fetching and delivery. In steering both instruction fetch and delivery, the predictors attempt to minimize penalties from instruction cache misses, incorrect branch paths, and target redirects.

All prediction structures have a finite size, and while from a capacity and precision perspective bigger is better, real world practice shows that access latencies, silicon area, and power consumption limit designers from using overly large structures [10, 14].

This paper describes the two level hierarchical branch predictor implemented in the IBM zEnterprise EC12 [13]. This branch prediction hierarchy achieves the performance benefit of a very large capacity predictor with minimal impact on latency and power. The design is a semi-exclusive two level structure where the large second level predictor is used to back fill the smaller and lower latency first level predictor. Predictions affecting program flow are only made from the fast first level structure which is located close to the instruction fetch and delivery pipeline. There is more flexibility in the placement of the second level structure since it is decoupled from the first level. The second level predictor is only powered up and accessed when content is perceived as missing from the main first level predictor.

The second level predictor has its own steered search and hit pipeline capable of providing history on a large number of tagged branches that correlate to the current instruction address space. When employed, the second level predictor is capable of searching either a small or large amount of address space. The small search space option is used for sporadic capacity based first level content gaps, and the large option for more wide spread gaps normally associated with cold code. An additional small BTB is used to prevent bulk second level transfers from polluting the main first level predictor. This additional structure also serves as a first level predictor victim buffer.

The performance benefits of this design, as compared to a smaller one level predictor, are explored using information gathered from both trace driven simulated models and from an actual IBM zEnterprise EC12 machine.

978-1-4673-5587-2/13/$31.00 ©2013 IEEE

2. Background

Branch target buffers (BTBs) have been around for decades [5]. Sussenguth describes a mechanism for looking up addresses and redirecting the flow of instructions to prevent sequencing issues [7]. Studies have shown that a branch prediction scheme with a BTB has considerable advantages. Modern processors realize a high performance benefit from branch prediction, especially as lower cycle times have necessitated increased pipeline depths [3].

Significant advantages may be obtained when the BTB is as large as possible, with all other factors constant [4]. Theoretically, to maximize performance, BTBs would be large enough to retain all the branches in a working application set, yet maintain low access latency and remain on-chip [2], near the instruction fetch and branch prediction logical structures, typically located in the timing-critical part of the processor pipeline [6]. In addition to being sized to contain a large number of branches, the more complete the branch information stored per BTB entry, the better the performance, provided the content reduces aliasing [16].

A large BTB tends to be more accurate, but the increased size adds to the latency of accessing the BTB. Slower BTB access increases instruction sequence redirection penalties [10]. It is becoming increasingly difficult to complete a BTB search in one or two cycles because of decreasing cycle time and increasing storage structure size [9]. Jimenez et. al. in [14] showed that latency constraints must be considered when designing branch prediction logic. As microchip manufacturing technologies shrink, the delay to access content from the branch prediction logic potentially deteriorates, since transistors and wires may not equally scale. Attempts have been made to mask this latency. One way is to pipeline the branch predictor. Jimenez [19] and Seznec et al. [26] describe examples of pipelined direction predictors.

One encounters a tradeoff between a very large cache to store a large amount of branch information, or metadata, versus a small storage element that has fast access time [9]. However, a small BTB that has a fast access time is subject to aliasing with older entries being overwritten by newer ones [10]. Since the BTB typically stores only a portion of the branch's virtual address as a tag, aliasing can occur among branches within a thread and among branches in different threads [4,5,16].

A balance must be made among the number of entries, content per entry, access latency and accuracy of the prediction on various workloads. Several variations of multiple level branch prediction mechanisms have been proposed to help alleviate the inherent shortfalls that a traditional branch prediction implementation possesses [6, 8, 9, 10, 11, 12]. The solutions are analogous to a memory hierarchy, where a small amount of data is accessed quickly (e.g., a L1 cache) and large amount of data is accessed slowly (L2, L3, main memory etc.). It was suggested by [16] that multilevel BTBs may be able to strike a balance between the large number of entries needed in the BTB and the desire to have a large amount of metadata per entry.

Sometimes the levels are accessed in parallel and the slower, larger level can override a prediction from the smaller, faster level [9]. Alternatively, after not finding a branch in the first level, it is looked for in the second level which can then provide a prediction [10, 11]. Other approaches use a larger, slower second level to preload entries into the smaller, faster level [6, 8, 12].

Virtualization of prediction storage structures has recently emerged as another approach to address the latency, size and cost issues of traditional predictors. Predictor virtualization was suggested in [17] as a means to reduce the resource a conventional prediction system would consume while still maintaining a high degree of accuracy. Burcea et al. suggest virtualizing large prediction structures by exploiting the existing memory hierarchy in place of dedicated structures. A small physical storage structure would cache the working set of prediction data close to the timing critical hardware, while the remaining inactive metadata resides in memory.

Recognizing the potential of virtualization, Emma et al. in a patent application filed in 2003 [1] describe a system to preload branch prediction data from a larger, higher latency virtual cache table into a smaller, lower latency non-virtual dedicated branch prediction structure. The perceived capacity of the branch prediction structure is increased without adding additional branch prediction structural capacity.

Burcea et al. propose a virtual BTB, called a phantom BTB [2]. They also provide a solution that accommodates large workload footprints without dedicating large chip resources for a conventional BTB, expanding upon the work invented in [1]. This work relies on temporal correlation within the first level BTB miss stream to guide the phantom BTB in prefetching branch information from the L2 cache.

Aasaraai et al. applied the concept of prediction virtualization to branch direction predictors [18].

3. Two level bulk preload branch prediction

This section describes bulk preload branch prediction developed for the IBM zEnterprise EC12. It introduces new structures and algorithms for implementing a two level hierarchical BTB. The zEC12 is a machine with big-endian 64-bit addressing. Therefore bit 0 is the most significant and bit 63 is the least significant.

3.1. Branch prediction hierarchy

The branch prediction hierarchy consists of several structures. These are depicted in Figure 1 along with arrows representing the flow of information between them. They are implemented as either SRAM arrays or register files.

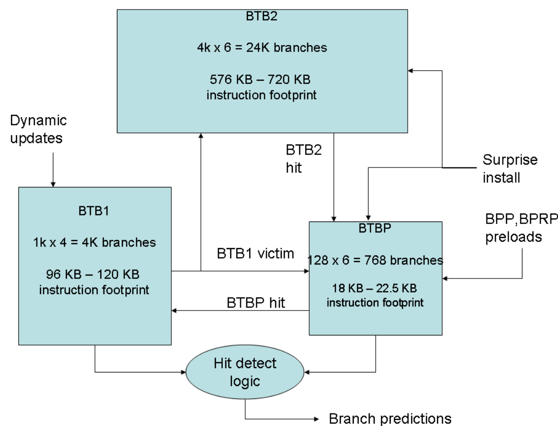

Figure 1. Branch prediction hierarchy

The first level Branch Target Buffer (BTB1) is a tagged cache of branch prediction information. It is indexed and tagged with branch instruction addresses. In addition to tag information, each BTB1 entry contains a 2-bit bimodal Branch History Table (BHT) direction prediction and a target address used for predicted taken branches. The BTB1 contains 4k branches, is organized as a 1k x 4-way set associative cache, and is implemented as an SRAM array. Instruction address bits 49:58 are used to index into the array. Therefore, each row in the BTB1 covers 32 bytes of instruction space.

The Branch Target Buffer Preload Table (BTBP) is also a Branch Target Buffer (BTB). Each BTBP entry contains the same type of content as the BTB1. It is read in parallel with the BTB1 to make branch predictions. Together, the BTB1, the BTBP and some auxiliary structures mentioned below are considered the first level branch predictor. Any branch not predicted by the first level predictor is called a surprise branch and its direction (taken or not-taken) is guessed based on a tagless 32k entry one-bit BHT, its opcode and other instruction text fields. The BTBP contains 768 branches and is organized as a 128 x 6-way cache. Instruction address bits 52:58 are used to index into the array. Therefore, like the BTB1, each row in the BTBP covers 32 bytes of instruction space. The BTBP is implemented as a register file with multiple write ports to support the many sources of writes into the branch prediction hierarchy: surprise installs from statically guessed branches, branch preload instructions, BTB2 hits, and BTB1 victims. The BTBP serves as a filter for the BTB1. Branch prediction information is initially written into the BTBP from various sources. Content is moved into the BTB1 upon making a branch prediction from the BTBP. At that time the replaced BTB1 entry (the BTB1 victim) is moved into the BTBP and the second level Branch Target Buffer (BTB2).

The BTB2 is also a BTB containing the same type of information as the BTB1 and BTBP. The BTB2 is written upon surprise installs into the branch prediction hierarchy and with entries evicted from the BTB1 upon transferring BTBP predictions into the BTB1. The BTB2 contains 24k branches and is organized as a 4k x 6-way cache, and is implemented as an SRAM array. Instruction address bits 47:58 are used to index the BTB2. Therefore, like the BTB1 and BTBP, each row in the BTB2 covers 32 bytes of instruction space.

An estimate of the instruction footprint covered by each of these structures can be made. This is based on average instruction length, average number of branch instructions, and average number of ever-taken branches which get installed into the BTBs. It is estimated that each BTB entry covers 24 - 30 bytes of instruction address space. Therefore, the first level predictor consisting of the BTB1 and BTBP is estimated to cover a footprint of 114 KB - 142.5 KB.

Auxiliary structures called the Pattern History Table (PHT) and Changing Target Buffer (CTB) are used as part of the first level branch predictor for branches exhibiting multiple directions and targets. They are indexed based on the path taken to get to a branch and are tagged with branch instruction address bits. Information is also maintained in the BTB1 and BTBP to control whether or not the PHT and/or CTB are allowed to be used for a particular branch. They are the same size and similar configuration as in the IBM zEnterprise 196 [15]. The PHT contains 4,096 entries and is indexed based on the direction of the 12 previous predicted branches and the instruction addresses of the 6 previous taken branches. The CTB contains 2,048 entries and is indexed based on the instruction addresses of the 12 previous taken

branches. These predictors are similar to the tagged ppm-like predictors described by Michaud [25].

3.2. Branch prediction search process

The first level branch predictor is searched asynchronously from instruction fetching and decode. This type of design is called an asynchronous lookahead branch predictor. Upon a restart condition, such as a mispredicted branch, both instruction fetching and branch prediction start at the same instruction address. The branch prediction logic searches for the first branch at or after this starting search address in parallel to instruction fetch logic fetching instructions sequentially. Upon finding a branch in the first level predictor, it predicts whether the branch is taken or not-taken. If it is predicted taken, it also predicts a target address. The branch predictor then redirects itself either to the target of the predicted taken branch or sequentially after the predicted not-taken branch and looks for the next branch. The sequential redirect allows for greater PHT accuracy. Information about the predicted branch is sent to the instruction fetch and decode logic to redirect fetching to the predicted target stream in the case of a dynamically predicted taken branch, and to apply the direction prediction to the predicted instruction at the time of instruction decode. Some of the branch prediction information is also stored until completion time of the branch and used at that time to update the branch prediction structures. Until table updates take place, speculative BHT and PHT updates are applied to predictions.

Usually branch prediction operates ahead of instruction fetch and decode. It is therefore able to effectively steer instruction fetching and minimize or completely eliminate the target redirect penalty for predicted taken branches. Because the estimated instruction footprint of the first level branch predictor (114 KB – 142.5 KB) is greater than the size of the first level instruction cache (64 KB), branch predictions occur for branches and their targets not in the first level instruction cache. Such predictions initiate instruction fetches that preload the necessary instructions into the first level instruction cache, often before the decode logic reaches that point. Therefore, the asynchronous lookahead branch predictor reduces or completely hides the first level instruction cache miss penalty.

Table 1. First level branch prediction search pipeline

Cycle	Search process	Re-index for prediction	Re-index sequential
b0 (x)	Index arrays with search address x.		
b1	Access arrays.		b0 (x+1)
b2	Start hit detection.	If under FIT control, re-index (b0) with FIT-supplied index for expected branch prediction.	b0 (x+2)
b3	Finish hit detection. Select prediction information.	If not under FIT control, re-index (b0) assuming taken prediction from MRU column.	
b4	Broadcast prediction info for taken prediction from MRU column.	If necessary, re-index (b0) for not-taken prediction or taken prediction not from MRU column.	
b5	Broadcast prediction info for 1st not-taken prediction or taken prediction not from MRU column.	If necessary, re-index (b0) for second not-taken prediction.	
b6	Broadcast branch prediction info for 2nd not-taken prediction.		b0 (x+3)

The branch prediction pipeline consists of the 7 cycles described in Table 1. Branch prediction throughput and search rate are variable. The branch prediction pipeline is able to predict branches as fast as one prediction every cycle. This fastest case is a loop

consisting of a single taken branch. When this case does not apply, branch predictions are possible every other cycle with the assistance of a 64 branch Fast Index Table (FIT) which accelerates branch prediction re-indexing on a 64 branch subset of the BTB1. When the FIT does not apply, it is possible to predict one taken branch every 3 cycles when those taken branches are in the most recently used (MRU) BTB1 column. Otherwise it is possible to predict one taken branch every 4 cycles. Not-taken predictions are possible at the fastest rate of 2 predictions every 5 cycles since each row searched in the BTB1 is allowed to make up to 2 not-taken predictions simultaneously. When this case does not apply, one not-taken prediction is possible every 4 cycles. When the search pipeline is not re-indexing itself for a predicted branch, it proceeds sequentially. If no branch predictions are found, the average search rate is 16 bytes per cycle. This is actually 3 cycles at 32 bytes per cycle followed by 3 cycles of 0 bytes per cycle, because on the 4th 5th and 6th cycles the logic is speculatively re-indexing the level one structures assuming the 1st, 2nd, and 3rd searches will find a predicted taken branch in the most recently used columns.

3.3. Maximizing exclusivity between BTB1 and BTB2

The BTB1 and BTB2 are maintained in such a way to approximate an exclusive hierarchy. Ideally, from a capacity standpoint, the BTB1 and BTB2 would be truly exclusive of each other. This would maximize the number of unique entries in the tables. In practice, the design is "semi-exclusive;" there are potentially duplicates in the BTB1 and BTB2. Techniques are employed to minimize duplicate entries across the levels.

A truly exclusive BTB hierarchy would invalidate or replace BTB2 entries whenever they are written into the BTB1. It may or may not include a BTBP. Without a BTBP, BTB2 hits would be transferred directly into the BTB1. At that time the BTB1 victim would be written into the BTB2. If the victim and the hit map to the same BTB2 row, it is trivial to have the victim overwrite the hit. If however they map to different BTB2 rows, then two write operations would be necessary. The first write would invalidate the hit and the second would write the victim into the BTB2. With a BTBP, BTB2 hits would be written into the BTBP. Upon BTBP predictions they would be written into the BTB1 and the BTB1 victim would be written into the BTB2 and BTBP. At that time exclusivity would be guaranteed by either replacing the BTB2 hit with the BTB1 victim or by explicitly invalidating the BTB2 hit. This would require the BTBP to remember the BTB2 column corresponding to the BTB2 hit. There is therefore a relatively high cost to guarantee exclusivity: additional BTB2 writes and storage in the BTBP for the BTB2 column.

A truly exclusive design would also avoid duplication upon surprise installs. Upon encountering a surprise branch, it would be necessary to check whether it already exists in the BTB1. If so it would be prevented from being written into the BTB2.

Due to the high cost of perfect exclusivity, a semi-exclusive design has been chosen. When an entry is copied from BTB2 to BTBP, it is made LRU in the BTB2. Upon moving content from the BTBP to BTB1, the content that is evicted from the BTB1 is written into the LRU column in the BTB2 and made MRU. By making BTB2 entries LRU upon BTB2 hits, it is likely that they will be replaced by subsequent BTB1 evictions and/or surprise installs. This avoids an extra write to the BTB2 structure for invalidation but does require an additional write the BTB2 LRU structure. However, the LRU can be a separate, smaller structure than the BTB2 array itself. This approach also does not require the BTB2 column to be stored in each BTBP entry.

Compared to an inclusive design, an exclusive configuration has the advantage of the BTB2 holding the most recently updated branch prediction information. Upon eviction from the BTB1, any information that has been learned about that branch's behavior is written into the BTB2. In an inclusive design, as branch prediction information is used and updated, a mechanism would be needed to update the information in the corresponding BTB2 entry. Some action would also be necessary to ensure or encourage inclusivity. For example, upon using a BTB1 or BTBP entry to predict a branch, the corresponding BTB2 entry could be made most recently used. Inclusive designs would therefore either update BTB2 entries leading to higher write activity and more power usage than an exclusive design, or forgo such updates to save power at the expense of worse prediction accuracy once stale content from the BTB2 is ultimately used.

3.4. Definition of a miss in the first level predictor

In an asynchronous lookahead branch predictor, a first level predictor miss, also called BTB1 miss, is detected after searching the BTB1 and BTBP for a predefined number of searches without finding any predictions. An example of detecting a BTB1 miss is shown in Table 2. It shows the process when the BTB1 miss limit is 3 searches, up to 96 bytes. The

three search limit is easier to show in the table than the actual setting of 4 searches, 128 bytes, used in the performance studies.

Table 2. BTB1 miss detection as part of first level prediction search process

Cycle	0	1	2	3	4	5
start 0x102	b0 BTB1 BTBP index	b1 BTB1 BTBP access	b2 start hit	b3 finish hit detect 1st miss		
search+1 0x120		b0	b1	b2	b3 2nd miss	
search+2 0x140			b0	b1	b2	b3 3rd miss BTB1 miss reported at starting search address 0x102

Simulation shows that reporting a BTB1 miss after 4 searches without predictions, up to 128 bytes, provides the best results on the studied workloads (Figure 6). Defining a BTB1 miss in this manner allows the miss to be detected and reported to the BTB2 logic very early in the pipeline. It is however by its nature speculative since a lack of predictions from the first level predictor does not necessarily mean that they are being missed for capacity reasons. It could be the case that the code does not contain any branches in the address range being searched. This would be the case in a long unrolled loop.

Alternative ways of defining BTB1 misses are possible. One such way would be to define a BTB1 miss whenever an actual branch instruction is encountered in the decode stage of the pipeline without having been dynamically predicted by the first level branch predictor. Furthermore, only certain types of encountered branches might be detected. For example, only those that are statically guessed taken based on the opcode and instruction text. This alternative method of defining BTB1 misses could be used instead of, or in addition to, the initial definition of a BTB1 miss presented above.

This alternative way of defining a BTB1 miss need not occur at the decode stage of the pipeline. Misses could be triggered at any stage of the processor pipeline from decode until completion. There is a trade-off between an earlier more speculative indication of a BTB1 miss and a later less speculative indication of a BTB1 miss.

3.5. Filtering BTB2 transfers based on instruction cache miss

The definition of a BTB1 miss is speculative. Therefore, it is beneficial to filter out BTB1 misses that are less likely to be actual first level BTB capacity misses. This is done by determining whether or not a detected BTB1 miss also has a corresponding first level instruction cache miss. BTB1 misses that also have corresponding instruction cache misses, in the same 4 KB block, are considered very likely to be BTB1 capacity misses. BTB1 misses that do not correspond to instruction cache misses are considered less likely to be capacity misses.

Filtered BTB1 misses can either be prevented from accessing the BTB2 or limited to a partial BTB2 search. In the implemented design, filtered BTB1 misses are limited to a 4-row BTB2 search (128 bytes), rather than the full 128-row BTB2 search (4 KB).

This approach is effective since the footprint of the first level predictors (BTB1+BTBP) is estimated to be between 114 KB and 142.5 KB which is greater than the size of the 64 KB first level instruction cache. Usually if a branch is displaced from the BTB1 and BTBP for capacity reasons it will also have been displaced from the instruction cache. Therefore, cases of a detected BTB1 miss and corresponding instruction cache hit are less likely to be due to limited BTB1 capacity and more likely to be false speculative perceived misses in code containing no branches at all.

3.6. BTB2 search process

Three BTB2 search trackers are implemented to remember information about BTB1 misses and instruction cache misses; and to initiate read accesses to the BTB2 structure. Each tracker represents one 4 KB block of address space (instruction address bits 0:51). It stores the following information:

- Block instruction address
- BTB1 miss validity
- Instruction cache miss validity
- Information about the priority and search status for each of the 128 32 byte BTB2 rows which make up the block being tracked.

Trackers are initiated upon BTB1 misses and instruction cache misses. The newly detected miss is compared against existing trackers. If a BTB1 miss matches an existing tracker with a valid BTB1 miss, it is ignored. If a BTB1 miss matches an existing tracker with only a valid instruction cache miss, it validates the BTB1 miss part of the tracker. If a BTB1 miss doesn't match any valid trackers it attempts to initiate a new

tracker. Instruction cache misses are handled in an analogous way.

A queue of 8 recently seen BTB1 misses and a separate queue of 4 recently completed BTB2 searches are maintained. Incoming misses which match these queues are ignored. This avoids duplicate transfers into the BTBP.

When a miss needs to initiate a new tracker, it does so by utilizing any completely invalid tracker. If no such tracker exists, it replaces any tracker in the state of only having a valid instruction cache miss and an invalid BTB1 miss. If no trackers are available for replacement, the incoming miss indication is ignored.

Trackers with both a valid BTB1 miss and a valid instruction cache miss are fully active trackers. Such trackers initiate reads to all 128 BTB2 rows in the 4 KB block. All branch prediction information read from the BTB2 with matching tags are called BTB2 hits and are written into the BTBP. The order of the reads is determined by priority logic which is described in the next section.

Trackers with only a valid BTB1 miss indication and an invalid instruction cache miss indication initiate partial BTB2 searches. Specifically, the 128 byte section of code corresponding to BTB1 miss address bits 0:56 in the BTB2 (4 rows) is searched. By the time this partial search completes, if the instruction cache miss validity bit in the tracker is still invalid, the tracker is completely invalidated. Trackers with only an instruction cache miss valid indication and an invalid BTB1 miss indication do not initiate any BTB2 searches.

Upon a BTB1 miss, the fastest the BTB2 search can be started is in the b10 cycle. This is 7 cycles after the miss is detected in the b3 cycle of the search process. The BTB2 search itself takes 8 cycles. Accesses are pipelined such that one BTB2 row is searched each cycle once searching is underway. Therefore, a full 4 KB bulk transfer takes 128 + 8 = 136 cycles.

3.7. BTB2 search steering

When transferring content from the BTB2, a 4 KB block of sequential addressing space is searched. Transferring all content sequentially from this block, even if the start point is based on the entry point into the block, is not the most efficient way to transfer branches from the BTB2 into the BTB1. Code executed in the 4 KB block is likely to encounter multiple taken branches and as such not be only sequential in nature. The goal is to transfer those sections in the 4 KB block that will be used first based on where the 4 KB block was entered. To determine an ordering of BTB2 return into the BTBP, an ordering table is tracked as a function of instruction checkpoint.

Given a 128 byte sector size, there are 32 sectors within a 4 KB block. The 4 KB block is divided into four 1 KB quartiles. Each quartile contains eight 1-bit sector markings and three markings to denote a reference to the other quartiles within the block. As a function of instruction checkpoint, sector ordering is tracked as follows. When a different 4 KB block is entered, the given quartile of entry is defined as the demand quartile. All sectors within the 4 KB block that have an instruction complete get the sector bit set to a '1'. If another quartile is entered from within the block the associated quartile bit within the demand quartile is also set to a '1'. This process continues for the given block until another block is entered. At this point, this information is stored and tracked in the ordering table array. The ordering table array tracks information as chunks of 128 bytes, which are called sectors. The table contains 512 entries and is 2-way set associative. Each entry represents a 4 KB block; therefore the table covers a 2 MB instruction footprint. Upon returning to the given block, information from the tagged ordering table array is retrieved and updated with any new paths which are traversed by the program code.

In parallel to initiating a search in the BTB2, the 4 KB pattern block is looked up in the tagged ordering table array. Given a table hit, content is provided for defining the return ordering from the BTB2. First, sectors from the demand quartile marked active are searched and transferred. Second, quartiles which are denoted as referenced from the demand quartile are searched in the BTB2 given the sector bits are active. Third, sector bits that are active but not in the demand or demand referenced quartiles are transferred from the BTB2 to the BTBP. After these transfers are performed, the same priority is repeated for those 128 byte sectors which do not have the sector bit set to a '1'. If there is not a table hit, content is returned in sequential order beginning with the demand quartile.

In the case of multiple 4 KB blocks being addressed by the BTB2 in parallel, the BTB2 will be prioritized to handle all 128 byte regions in the demand quartiles with the sector bits active. Ordering will then proceed through the remaining priorities for all of the multiple 4 KB blocks of data being requested by the BTB2.

4. Study methodology

The performance provided by the two level bulk preload branch predictor was studied in a C++ simulation model of the zEnterprise EC12 microprocessor using traces of several large commercial workloads. It was also studied on a zEC12 machine running two workloads. Relevant characteristics of the zEC12 are shown in Table 5.

Part of the microprocessor design process includes developing a C++ performance model of the microprocessor. This model includes the performance-relevant aspects of the microarchitecture such as caches, execution units, pipeline depths, and bypasses. It models multiple levels of the instruction and data cache hierarchy. For the studies in this paper, finite models of the first level caches are used. The second level caches and beyond are considered infinite. Therefore, upon any first level cache miss, a second level cache hit is assumed.

Wrong path execution is modeled. Whenever the processor goes down a mispredicted path the model simulates what the hardware would encounter down this path using information in the trace about which instructions are located at the wrong path instruction addresses. In some cases, if the model is unable to determine what instructions are at the wrong path addresses, it substitutes filler instructions which are either no-ops or instructions from some previously-encountered portion of the trace.

Traces collected from running various workloads and benchmarks are maintained. These traces are constructed to be representative of the performance of the entire workloads of interest even though they are often only portions of an entire benchmark in order to minimize trace size and therefore also simulation time.

For this study three configurations were analyzed. The configurations of the major branch prediction structures in each of the simulated configurations are shown in Table 3.

Simulating these configurations allows one to see the performance benefit of adding the BTB2 (configuration 2) compared to the performance potential of an unrealistically large low-latency BTB1 (configuration 3). The percent improvement in Cycles per Instruction (CPI) of configurations 2 and 3 compared to configuration 1 was calculated after running the performance studies. The results are presented in the subsequent "Results" section of this paper for the traces determined to be large footprint workloads.

The trace pool was analyzed to identify workloads with a large branch instruction footprint. More specifically, any trace with more than 5,000 unique taken branch instruction addresses is a good candidate for showing improvement from additional branch prediction capacity.

Table 3. Simulated Configurations

Name	BTBP	BTB1	BTB2
1. No BTB2	768 (128 x 8)	4k (1k x 4)	0 (Disabled)
2. BTB2 enabled	768 (128 x 6)	4k (1k x 4)	24k (4k x 6)
3. Unrealistically large BTB1	768 (128 x 6)	24k (4k x 6)	0 (Disabled)

Table 4. Large footprint traces

Trace name	Number of unique branch instruction addresses	Number of unique taken branch instruction addresses
1. Z/OS LSPR CB84	15,244	10,963
2. Z/OS LSPR CICS/DB2	40,667	27,500
3. Z/OS LSPR IMS	29,692	19,673
4. Z/OS LSPR CB-L	25,622	16,612
5. Z/OS LSPR WASDB+CBW2	114,955	51,371
6. Z/OS Trade6	115,509	56,017
7. TPF airline reservations	11,160	9,317
8. Z/OS AppServ benchmark	26,340	16,980
9. Z/OS DBServ benchmark	38,655	20,020
10. Z/OS DayTrader AppServ	67,336	30,165
11. Z/OS DayTrader DBServ	34,819	22,217
12. zLinux Informix	16,810	11,765
13. zLinux Trade6	69,847	31,897

The large footprint benchmarks expected to benefit from the BTB2 for which results are subsequently presented are listed in Table 4 along with counts of all and ever-taken unique branch instructions addresses.

Traces 1 through 5 are workloads from the IBM Large System Performance Reference (LSPR) [20] running on the Z/OS operating system. Trace 5 includes a mix of two of the LSPR workloads time sliced on one processor. Trace 6 is the Trade 6 workload [21] on Z/OS. Trace 7 is of a workload running an airlines reservation system under the TPF operating system [22]. Trace 8 is a commercial application server. Trace 9 is a commercial database

server. Traces 10 and 11 are the DayTrader benchmark [23]. Traces 12 and 13 are the Informix [24] and Trade6 workloads running on the Linux operating system.

Table 5. zEnterprise EC12 chip configuration

L1 Cache	Instruction cache 64KB (4-way) Data cache 96KB (6-way)
L2 Cache	Instruction cache 1 Meg (8-way) Data cache 1 Meg (8-way)
L3 Cache	48 Meg on-chip
L4 Cache	384 Meg off-chip
I-TLB1	4K & 1 Meg pages: 64 x 2
D-TLB1	4K pages: 256 x 2 1M pages: 32 x 2 2G pages: 1 x 8
TLB2	128 x 4 CRSTE; 256 x 3 PTE / CRSTE
Issue Queue	32 x 2
Completion Table	30 x 3 micro-ops
Physical Regs	80 general registers, 64 floating point
Issue bandwidth	7 (2 LSU, 2 FXU, 2 Branch, 1 Float)

5. Results

This section presents and analyzes the results from both the simulation model and from runs on the actual hardware. The BTB2 design was analyzed against a wide variety of environments to show the variation in potential benefit. Each individual result should not be considered as representative of all workloads within that specific environment.

5.1. BTB2 in zEnterprise EC12

Figure 2 shows the results of the performance studies in the C++ performance model. It shows the improvement in CPI from the two level branch predictor with the 24k BTB2 enabled (configuration 2) as well as the theoretical improvement possible from the unrealistically large 24k one level branch predictor (configuration 3). These improvements are with respect to the baseline configuration with the BTB2 disabled (configuration 1).

The bottom bars present the performance benefit of enabling the 24k (4k x 6-way) BTB2. The top bars present the performance benefit of an unrealistically large 24k (4k x 6-way) low-latency BTB1. The numbers on the right are the BTB2 effectiveness, which is defined as the ratio of the improvement from adding the BTB2 compared to the improvement from adding the unrealistically large BTB1.

For the traces and configurations studied, the maximum benefit of the BTB2 is 13.8% on the DayTrader DBServ trace. This compares to a benefit of 20.2% on this same trace with a large BTB1. BTB2 effectiveness compared to the large BTB1 varies from 16.6% to 83.4% with an average of 52%.

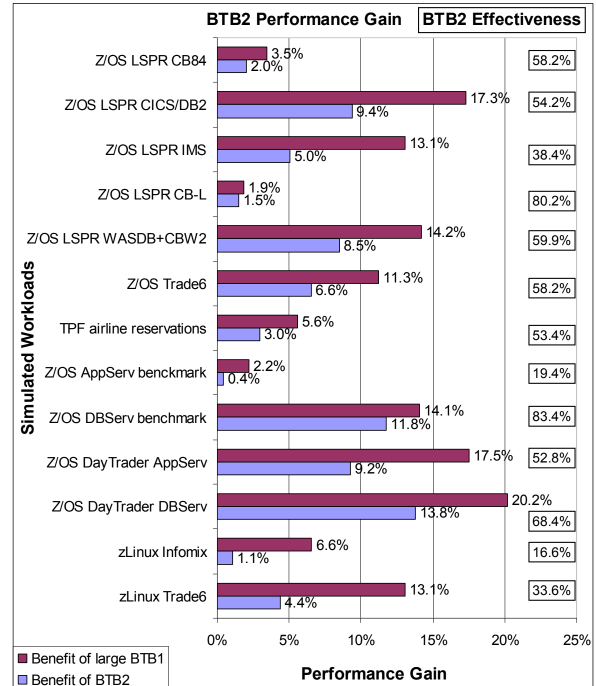

Figure 2. Benefit of BTB2, large BTB1 from C++ simulation model

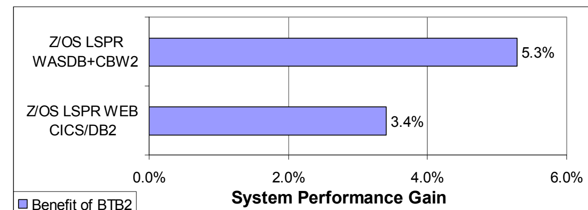

Figure 3. Benefit of BTB2 on zEC12 hardware

Figure 3 shows the performance benefit of enabling the BTB2 on zEC12 hardware. The WASDB+CBW2 workload was run on a single core. It can be seen that the system performance improvement measured in actual hardware (5.3%) on this workload is less than the performance improvement seen in the simulation model (8.5%). This is expected because only the first level instruction and data caches were modeled as finite in the simulation. The Web CICS/DB2 workload which was run on 4 cores is a new version of the single core CICS/DB2 trace. A 3.4% system performance improvement was measured on this workload

demonstrating the benefit of the BTB2 in a multi-core environment.

The BTB2 reduces the penalty from surprise branches which often incur a target redirect or misprediction penalty. It also reduces the instruction cache miss penalty due to more predicted taken branches initiating and overlapping instruction fetches earlier than if they were encountered as surprise branches.

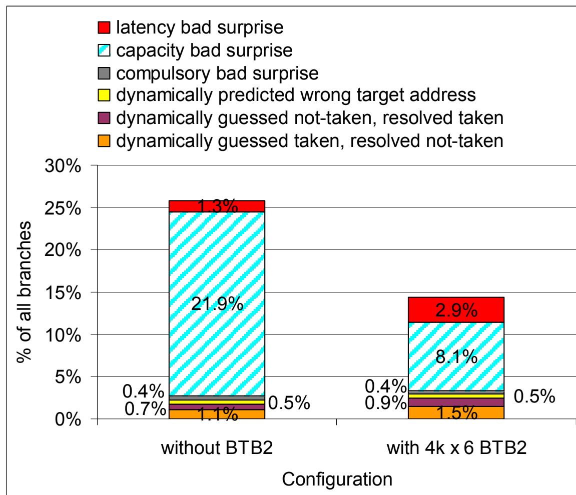

Figure 4. Effect of BTB2 on bad branch outcomes from C++ model of Z/OS DayTrader DBServ

Figure 4 shows the effect of the BTB2 on bad branch outcomes on the z/OS DayTrader DBServ trace from the C++ simulation model. This figure provides insight into why such a large performance improvement is achieved. As previously reported, the BTB2 improves performance of this workload by 13.8%.

Bad branch outcomes are those that incur a performance penalty. Specifically they consist of dynamically mispredicted branches and surprise branches which are guessed or resolved taken. These bad surprise branches are classified as compulsory (first time that branch is seen), latency (surprise because a prediction wasn't available in time either due to prediction falling behind decode, or due to latency for writing surprise branches into the prediction tables), or capacity (branch was seen before, and not categorized as missed due to latency). Dynamically mispredicted branches are guessed taken and resolved not-taken, guessed not-taken and resolved taken, or guessed taken and resolved taken with wrong target address prediction.

Figure 4 shows that without the BTB2, 25.9% of all branch outcomes are classified as being bad. Most of these (21.9% of all branch outcomes) are capacity bad surprise branches. Adding the BTB2 reduces the number of capacity bad surprise branches to 8.1%. The total number of bad branch outcomes is reduced to 14.3%. In large footprint workloads such as this, a large portion of the branch penalty is due to branch prediction capacity rather than branch direction and target prediction algorithms. This is the reason why the multi-level branch predictor described in this paper is so effective at improving performance.

5.2. Other simulated configurations

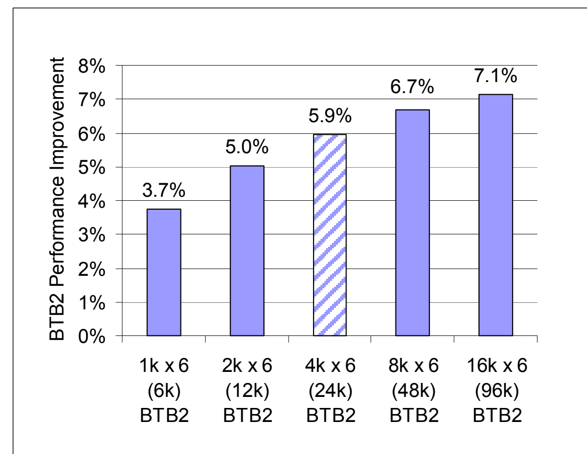

Figure 5. Various BTB2 sizes

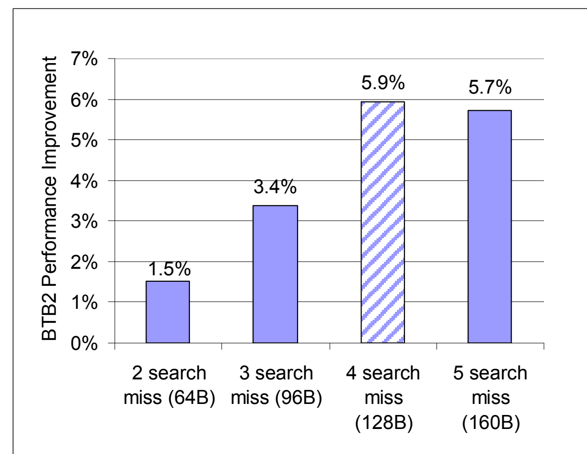

Figure 6. Various definitions of BTB1 miss

Additional configurations were simulated in the C++ model to investigate the effects of varying parameters of the design. The figures in this section show the benefit of the BTB2 compared to the configuration without the BTB2. They are the average of the 13 traces.

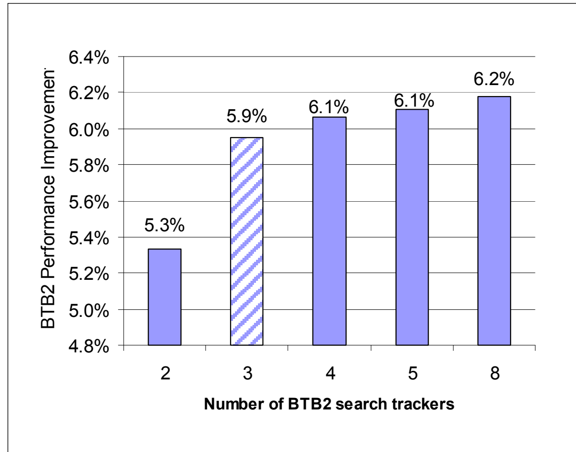

Figure 7. Various numbers of BTB2 trackers

Figure 5 shows the effect of varying the size of the BTB2 demonstrating the performance opportunity of a larger BTB2. Figure 6 shows the effect of varying the definition of a BTB1 miss. Figure 7 shows the effect of varying the number of BTB2 search trackers. These results support the choices made for the hardware implementation, shown in the charts with stripes.

6. Future work

Further gains in performance and performance/watt are being addressed through capacity and efficiency focus. A multi-level BTB allows for designing the first level in high-speed SRAM and the BTB2 in a higher density memory technology. Through optimal technology usage, the multi-level BTB design will support a greater number of predictions per square millimeter than a single level BTB designed solely in SRAM. Understanding the trade-offs between SRAM and eDRAM may be analyzed for defining an optimal design point which consists of SRAM for the BTB1 and eDRAM for the BTB2.

Improving the transfer rate will improve the efficiency of the ratio between a multi-level BTB hit rate and a large BTB1 only hit rate. The transfer rate is a function of bus utilization which is a function of the amount of address space covered per BTB2 congruence class. The current results apply an index which covers 32 bytes of instruction address space per 6-ways of associativity. This allows 6 predictions to be stored into the BTB2 for every 32 bytes of sequential virtual address space. By increasing the virtual address space per congruence class to 64 or 128 bytes, the amount of tag matching branches per search will increase at the cost of not being able to store all branches within a sequential code stream into the BTB2 because of overflowing a congruence class. The trade-off is not only an indexing bit range selection but also about which branches are to be tracked by the BTB2. An analysis may be made to determine how relative and indirect branches should be weighted for acquiring space in the BTB2. An additional analysis may be made to determine how branch target displacement ranges should prioritize which branches are stored into the BTB2.

Current work transfers only those branches within a 4 KB block. While the block size can be altered, being able to process more than a single block will increase the first level BTB hit rate. In a single BTB2 block search, there may be more than a single branch which redirects to another unique block. Without careful selection, the number of blocks to transfer can exponentially exceed the available bandwidth. An algorithm for multi-block transfers is an area of interest.

Investigating alternative ways of defining a BTB1 miss is a topic for additional study. Such research would explore the differences between detecting misses early in the pipe with high speculation as described in this paper versus later in the pipe with less speculation.

7. Conclusions

This paper describes the design and performance results of the two level bulk preload branch predictor used in the IBM zEnterprise EC12.

The novel structures and algorithms comprising this branch prediction hierarchy are presented as a solution to improve performance of large workloads sensitive to BTB capacity.

This design approximates the performance benefit of a very large BTB without negatively affecting the BTB1's access time, throughput, area, or power consumption. Simulated performance results of 13 traces show that the design achieves a maximum performance benefit of 13.8%. On average 52% of the benefit of an unrealistically large single level predictor of comparable capacity is attained. Measurements on actual hardware of two LSPR workloads show 5.3% and 3.4% system performance improvements.

The measured benefit of this design underlines its value. The design yielding half the value of its theoretical ceiling provides an impetus for continued research, refinement, and development.

8. Acknowledgements

We would like to thank the review committee and our colleagues Ioana Baldini, Jane Bartik, Steven Carlough, Andrea Harris, David Hutton, Gary King, Jim Mitchell, Michael Rihn, Eric Schwarz, Kevin Shum, Charles Webb for their advice and assistance in preparing this paper for publication.

9. References

[1] P. Emma, A. Hartstein, B. Prasky, T. Puzak, M. Qureshi, V. Srinivasan. “Context Look Ahead Storage Structures” *U.S. Patent 7,337,271*. February 2008.

[2] I. Burcea, A. Moshovos. “Phantom-BTB: A Virtualized Branch Target Buffer Design” *14th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2009.

[3] A. Hartstein, T. Puzak. "The Optimum Pipeline Depth for a Microprocessor" *Proceedings of the 29th Annual International Symposium on Computer Architecture*. May 2002. pp. 7-13.

[4] R.B. Hilgendorf, G.J. Heim, W. Rosenstiel. “Evaluation of branch-prediction method on traces from commercial applications” *IBM Journal of Research and Development*, Vol. 43 No.4, July 1999.

[5] J.K.F. Lee, A.J. Smith. “Branch Prediction Strategies and Branch Target Buffer Design” *Computer*. Vol.17 No.1. January 1984.

[6] P. Emma, K. Getzlaff, A. Hartstein, T. Pflueger, T. Puzak, E. Schwarz, V. Srinivasan. “Method and Apparatus for Prefetching Branch History Information” *U.S. Patent 7,493,480.* February 2009.

[7] E. H. Sussenguth. “Instruction Sequence Control” *U.S. Patent 3,559,183.* January 1971.

[8] J. Pomerene, T. Puzak, R. Rechtschaffen, P. Rosenfeld, F. Sparacio. “Pageable Branch History Table“ *U.S. Patent 4,679,141*. July 1987.

[9] D. Stiles, J. Favor, K. Van Dyke. “Two-Level Branch Prediction Cache” *U.S. Patent 5,163,140.* November 1992.

[10] T. Yeh, H. Sharangpani, “Method and Apparatus for Branch Prediction Using First and Second Level Branch Prediction Tables” *U.S. Patent 6,553,488.* April 2003.

[11] G. Zuraski Jr., J. Roberts. “Hybrid Branch Prediction Device with Two Levels of Branch Prediction Cache” *U.S. Patent 7,024,545.* April 2006.

[12] M. Davis, S. Jourdan, R. Hinton, B. Phelps. ”Overriding a Static Prediction with a Level-Two Predictor” *U.S. Patent 7,533,252.* May 2009.

[13] I. Dobos, W. Fried, et al., “IBM zEnterprise EC12 Technical Guide” http://www.redbooks.ibm.com/redpieces/pdfs/sg248049.pdf

[14] D. Jimenez, S. Keckler, C. Lin. “The Impact of Delay on the Design of Branch Predictors“ *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture.* 2000.

[15] F. Busaba, et al., “IBM zEnterprise 196 microprocessor and cache subsystem“ *IBM Journal of Research and Development*. Vol. 56 No. 1 / 2. Jan/Mar 2012.

[16] C.H. Perleberg, A.J. Smith. “Branch Target Buffer Design and Optimization” *IEEE Transactions on Computers*, Vol.42 No. 4. April 1993.

[17] I. Burcea, B. Falsafi, A. Moshovos, S. Somogyi. “Predictor Virtualization” *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII.* 2008. pp. 157-167.

[18] K. Aasaraai, A. Moshovos, M. Sadooghi-Alvandi. “Toward Virtualizing Branch Direction Prediction” *Design, Automation & Test in Europe Conference & Exhibition (DATE).* 2012. pp. 455-460.

[19] D. Jimenez. “Reconsidering Complex Branch Predictors” *Proceedings of the 9th International Symposium on High-Performance Computer Architecture.* 2002.

[20] IBM Corporation. “Large Systems Performance Reference” Document Number SC28-1187-15. https://www-304.ibm.com/servers/resourcelink/lib03060.nsf/pages/lsprindex?OpenDocument. July 2011.

[21] J. Coleman “Set up and run a Trade6 benchmark with DB2 UDB” http://www.ibm.com/developerworks/data/tutorials/dm0506lau/. June 2005

[22] IBM Corporation. “z/Transaction Processing Facility” http://www-01.ibm.com/software/htp/tpf/

[23] apache.org “Apache Geronimo Daytrader” https://cwiki.apache.org/GMOxDOC20/daytrader.html

[24] IBM Corporation. “Informix product family” http://www-01.ibm.com/software/data/informix/

[25] P. Michaud. “A ppm-like, tag-based predictor*” Journal of Instruction Level Parallelism*. April 2005.

[26] A. Seznec, S. Felix, V. Krishnan, Y. Sazeides. “Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor” *Proceedings of the 29th Annual International Symposium on Computer Architecture.* 2002.