The IBM z15 High Frequency Mainframe Branch Predictor

Industrial Product

Narasimha Adiga
IBM Systems Group
Bangalore, India
anarasim@in.ibm.com

James Bonanno
IBM Systems Group
Austin, Texas
bonannoj@us.ibm.com

Adam Collura
IBM Systems Group
Poughkeepsie, New York
collura@us.ibm.com

Matthias Heizmann
IBM Systems Group
Poughkeepsie, New York
maheiz@us.ibm.com

Brian R. Prasky
IBM Systems Group
Poughkeepsie, New York
brprasky@us.ibm.com

Anthony Saporito
IBM Systems Group
Poughkeepsie, New York
saporit@us.ibm.com

***Abstract* - The design of the modern, enterprise-class IBM z15 branch predictor is described. Implemented as a multi-level look-ahead structure, the branch predictor is capable of predicting branch direction and target addresses, augmented with multiple auxiliary direction, target, and power predictors. Predictions are made asynchronously, and later integrated into the processor pipeline. The design is optimized for the unique workloads executed on these enterprise-class systems, including compute intensive and both large instruction and data footprint workloads. This paper highlights the major operations and functions of the IBM z15 branch predictor, including its pipeline, prediction structures and verification methodology. Explanations as to how the design matured to its current state are also provided.**

I. Introduction

Mainframes are deployed throughout the modern world in industries such as banking, finance, insurance, retail, healthcare, air travel and many others. Such enterprise-class systems are the lifeblood of these businesses, and as such, the mainframe has a reputation as a highly reliable, available, secure, scalable, cost efficient hardware virtualization platform.

Built upon these tenants, these systems are designed to run any kind of workload at scale, and are optimized for high throughput transactions, typically to a vast database. These large system performance record (LSPR) workloads generally consist of a large instruction footprint [2].

The z15 is the latest mainframe offering from IBM. A superscalar, simultaneous multi-threaded (SMT), out-of-order processor core resides at its heart. Operating continuously at 5.2 GHz, the processor core utilizes a deep execution pipeline depicted in figure 1. The processor has a maximum throughput of up to 6 instructions per cycle. Working on the correct code path is essential, as it can take on the order of a couple dozen processor clock cycles to fill the pipeline after a branch wrong flush. A rich and robust branch predictor steers the front end of the pipeline and employs many strategies to minimize pipeline flushes and maximize performance.

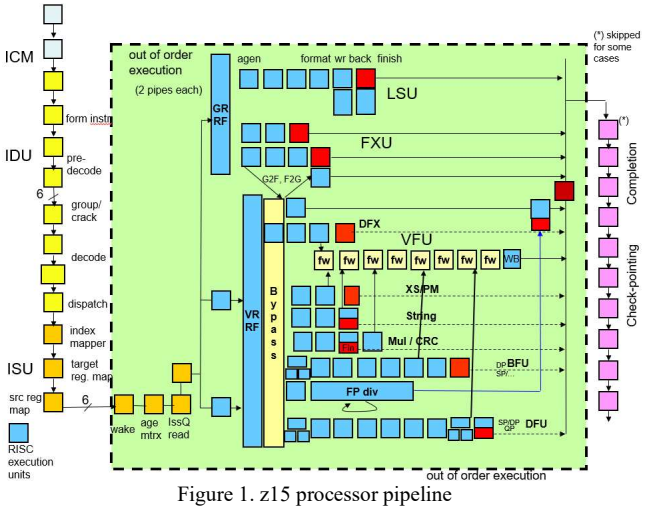

Figure 1. z15 processor pipeline

The z15 employs a four-tiered cache hierarchy. Of the 240 processor cores in a fully configured system, 190 are available to the customer, with the remainder offloading I/O operations, executing system firmware, configured as spares, or disabled to increase chip yield and decrease power consumption. Each core has private level 1 and level

This paper is part of the Industry Track of ISCA 2020's program.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00014

2 caches. Up to 12 processor cores reside on one central processor (CP) chip, and all share a level 3 on-chip cache. One 696 square millimeter CP chip contains over 15 miles of wire and over 9 billion transistors.

Each of the four CP chips per system drawer communicate with a system control (SC) chip, which in addition to enabling communication across up to 5 drawers on a system, also houses the level 4 cache. Based on the MOESI (modified, owned, exclusive, shared, invalid) protocol, the system maintains an inclusive cache design. Table 1 shows the z15 cache sizes in relationship to the past four generations of Z System mainframe processors.

The processor adheres to the z/Architecture [3]. This robust microprocessor architecture has evolved over the past six decades since the incarnation of the modern mainframe with the system s360 in 1964 [4]. This complex instruction set computing (CISC) architecture supports instructions of varying lengths: 2, 4 and 6 bytes. Because of this rich legacy, the microarchitecture must ensure performance on both new and legacy code. There are dozens of branch instructions defined, although none are true call and return instructions as supported in other architectures like IBM's Power series [5]. In general, z/Architecture branches can be divided into two major categories: indirect and relative branches.

As the name implies, relative branches have their target address at an offset from the instruction address of the branch itself. An offset field is defined as part of the branch's instruction text. The target address is computed by calculating the sum of the branch instruction address and the signed halfword offset, typically calculated by the front-end of the processor pipeline.

	zEC12 (32nm)	z13 (22nm)	z14 (14nm)	z15 (14nm)
CP chip	6 cores	8 cores	10 cores	12 cores
On Chip Cache sizes	L1: 64K I$, 96K D$ L1+: 1MB Private L2: 1MB Private L3: 48MB Shared	L1: 96K I$, 128K D$ L2I: 2MB Private L2D: 2MB Private L3: 64MB Shared	L1: 128K I$, 128K D$ L2I: 2MB Private L2D: 4MB Private L3: 128MB Shared	L1: 128K I$, 128K D$ L2I: 4MB Private L2D: 4MB Private L3: 256MB Shared
Branch Prediction	BTB1: 4K (1k x 4way) BTB2: 24K (6way) PHT: 4K CTB: 2K	BTB1: 6K (1K x 6way) BTB2: 96K (6way) PHT: 6K CTB: 2K	BTB1: 8K (2K x 4way) BTB2: 128K (4way) PHT: 8K CTB: 2K	BTB1: 16K (2K x 8way) BTB2: 128K (4way) PHT: 8K CTB: 2K
TLB sizes	64 x 2 I-TLB 256x2 4k page D-TLB 32x2 1M page D-TLB 8 fully associative D-TLB extension2 (mix page sizes) TLB2: CRSTE 128x4way TLB2: PTE 256 X 3way per CRSTE way	128 X 4 I-TLB 256 X 4 D-TLB (4k pages) 128 X 4 D-TLB (1M pages) 32 X 2 D-TLB (2G pages) TLB2: CRSTE 256 X 4way TLB2: PTE: 256 X 4way per CRSTE way	TLB1: New combined L1 virtual TLB/DIR TLB2: CRSTE 512 X 4way TLB2: PTE 256 X 6 way per CRSTE way TLB2: 64 entry 2-gig	TLB1: Combined L1 virtual TLB/DIR TLB2: CRSTE 512 X 4way TLB2: PTE 256 X 6 way per CRSTE way TLB2: 256 entry 2-gig

Table 1. Structure sizes of prior system Z processors

Indirect branches exploit the z/Architecture's method of calculating virtual addresses by the summation of three addends: two registers known as a base and an index, and a third displacement field. This summation is calculated by the fixed point units in the processor pipeline, generally about a dozen cycles into the back end of the processor pipeline.

Of the several units that comprise the processor core, branch prediction logic is the responsibility of the instruction fetch and branch prediction unit (IFB). The IFB is the core's navigator, arbitrating all pipeline restart points for both threads. Furthermore, the IFB guides the instruction cache and merge (ICM) unit, responsible for fetching instruction text from the level 1 instruction cache, attempting to ensure that only the instruction text on what it believes the correct speculative code path is delivered to the instruction decode and dispatch unit (IDU). The pipeline then dispatches to the instruction sequence unit (ISU) that maintains the out-of-order execution pipeline. Instruction issue queues are used to execute instructions in fixed-point units (FXU), vector and floating point units (VFU) and load-store units (LSU). Address translation is performed by the translator unit (XU), and checkpointing is managed by the recovery unit (RU).

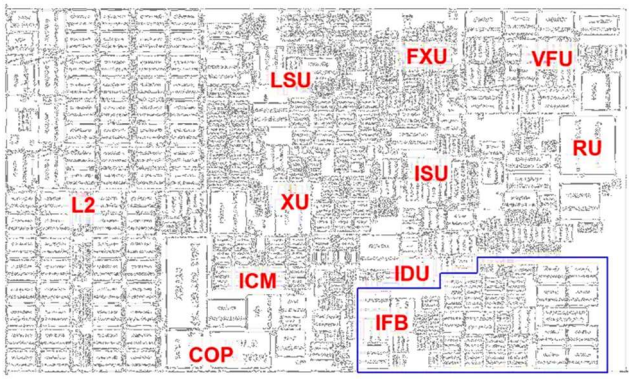

Figure 2. z15 core layout

Once an instruction is decoded to be a branch, the dynamically predicted information of that branch overrides any instruction text based static guess that the IDU would normally apply. The IFB, through the high accuracy of the branch predictor, promotes that the front end of the pipeline performs meaningful work, even though that work is speculative. Figure 2 shows the layout of the z15 processor core with the branch predictor component of the IFB highlighted.

II. Branch Prediction Design Considerations

Designs over the past decades have had to pick a balance point between single thread performance and overall throughput. Designs can increase threads or core counts and reduce cache capacity to increase the throughput and latency on a chip (and visa-versa). Mainframes have the requirement to effectively run both computationally intensive and large footprint programs that involve significant non-vector shared memory operations. Such broad requirements have yielded a design with a unique balance between core count and data, instruction, and meta caches. Addressing this balance, the z15 design includes 355 megabytes of storage on a chip for instructions and data that is shared across 12 cores.

Commercial workloads that run on mainframes typically have large instruction footprints. This is one of many aspects that designers must account for in order to ensure that fast transactional latency and throughput rates are achieved and that service level agreements can be met. Another design constraint is system throughput. Throughput is important but so is the response time of single transactions. For example, there is a finite amount of time that can elapse before a transaction such as an ATM inquiry or credit card swipe times out. Other design constraints may include time to market, silicon area, routing and wiring restrictions, leakage and switching power, performance, security, features for software debug, even customer product cycle expectations. These must also be accounted for beyond architectural correctness and instruction per cycle (IPC) performance in producing a chip. A balance must be established to provide customers a solution that grows with the advancements across the industry (e.g. manufacturing technologies and industry standards) and hence their requirements to provide solutions to their problems.

The branch prediction design considerations that follow have enabled core-balanced performance on commercial workloads. Across industry, a software transition is occurring. Monolithic programs are giving way to a large quantity of smaller, micro-services running in containers. The value provided by these design points addresses this transition in addition to traditional Z System workloads. These design trade-offs will provide insight into IBM Z.

A. Capacity

When the processor is actively running instructions, the goal is to be able to deliver as many instructions to the IDU per cycle as possible. Predicting the branch direction is the basic example of this. A non-taken branch is simple for instruction fetching since the fetching will continue sequentially. When a branch is predicted taken, the target instruction of the branch is ideally delivered for decode in the same or adjacent cycle as that of the branch. Large footprint workloads have large amounts of warm code. This makes the capacity of branch prediction structures critical to the overall performance of a design. The workload footprints run on a z15 are large enough to justify a 4 MB dedicated per core L2 I-cache which is delayed a minimal of 8 cycles over the L1 I-cache access.

The branch prediction tables (BTB, or branch target buffer is henceforth used) is equally important to increase in size. The BTB only tracks branches that have been statically guessed taken or resolved taken. Given a branch is encountered once every 4 instructions, this yields a BTB install for approximately 1 out of every 5 instructions since all branches are not installed into the BTB, such as statically guessed not taken branches that resolve not taken. System Z instructions have an average length of approximately 5 bytes. This places a branch once every 25 bytes. Rounding off to 32B, this suggests that a 4 MB instruction cache will contain up to 128K branches that will be installed into the BTB. Furthermore, if 1 of 2 BTB installed branches is predicted taken, then when fetching 128B of a cache line, there will be on average 2 taken branches that are encountered. The large L2 with a latency overhead of 8 cycles prevents going to the L3 with a latency of 45 cycles over an L1 hit. This is a savings of 37 cycles for only going to the L2. Predicting a single branch correctly saves up to 26 cycles of pipeline delay, the branch restart penalty. The ability to handle a pair of branches correctly through the use of second level branch prediction tables (BTB2) can exceed the performance advantage of the L2 I-cache. This does not decrease the value of a fast level 2 cache; it supports the design that given significant value to a second level I-cache, there is significant value to large branch meta data.

B. Latency

Latency is measured in multiple ways, including the delay after some form of a stall. Stalls are common after restarts but occur at other points also such as L1 I-cache misses. Examples of impacting stalls include the time to get the first and second predicted branch targets to the I-cache after a restart, the time to get content back from the BTB2, the time to redirect from a branch to the target instruction, and the ability for the branch prediction logic (BPL) to generate a pre-fetch stream ahead of the demand fetch to the I-cache to hide the latency induced from a L1 I-cache miss. Each of these is important to a different degree based on decisions made throughout the pipeline design. After a complete pipeline restart, the issue queue for the out-of-order engine can be empty. Recovery of filling up this reservoir along with generating a steady stream of I-fetches flowing to the I-cache can add up to 10 cycles of additional pipeline inefficiency delay to a restart event.

C. Throughput

Throughput is about both how fast branch prediction can get branch directions predicted for passing along to the IDU, and in getting the target instruction text of a branch to the IDU. For this, a fetch has to be made to the I-cache. Ideally if the L1-I is missed with a hit in the L2-I for example, the branch prediction will be running ahead of a full issue queue such that the delay of going to the L2-I and ideally L3 as needed is fully hidden.

D. Direction and Target Accuracy

Direction accuracy and target accuracy for indirect branches have rightfully had significant focus from academia over the decades [7] [8] [19]. As mentioned earlier, the impact from a wrong direction and hence also an indirect branch stall from not having a target prediction is up to 26 cycles per the pipeline length. However, the statistical penalty is about 35 cycles because of multiple queueing disruptions, including the out-of-order issue queue, which impacts pipeline efficiency.

With the aforementioned tradeoffs taken into consideration, the z15 branch predictor is composed of a main branch target buffer (BTB1) that also encompasses a branch history table (BHT). Backing up the BTB1 is a second level BTB2 that acts like a level 2 cache for the BTB1. Direction predictions are augmented by the pattern history table (PHT) and perceptron. Target address predictions are augmented by the changing target buffer (CTB) and call/return stack (CRS). A column predictor (CPRED) allows for fast re-indexing of the BTB1 to accelerate the delivery of predictions. Figure 3 shows a high-level view of the branch predictor. Of these tradeoffs, latency and direction accuracy were the key performance areas emphasized on the z15 branch predictor.

III. Capacity BTB-Based Design

Capacity, latency, throughput, and accuracy must all be considered when designing a branch predictor. The end goal is improving the performance of the design, and careful tradeoffs between the different considerations must be made. On large footprint workloads, increasing the size of the main BTB has a very regular corresponding positive impact on performance. As physics would have it, larger sizes start to negatively impact the access time of the structures (latency). Real world designs also have limited silicon area and power budgets that are negatively affected by large primary structures.

IBM Z System processors have been using a multiple level BTB design for several generations in order to increase the effective size (capacity) without a corresponding detrimental impact on the other design considerations and constraints. The original IBM Z multi-level BTB design dates back to the zEC12 mainframe [6][16]. The design has improved and evolved along the way but still shares many features of the original. The z15 BTB is made up of two levels (BTB1 & BTB2). The overall combined size of the BTBs has grown generation to generation as shown in table 1. The z15 BTB1 can hold up to 16K branches and is organized as 2K logical rows with 8 ways per row. The BTB2 can hold 128K branches and is organized as 32K logical rows with 4 ways per row.

Predictions that steer instruction fetching and speculative branch directions and targets are only made from the level 1 predictors. The BTB2 is used to backfill the main structure and is only accessed when content is thought to be missing from the BTB1. While similar to standard multi-level cache designs, the multi-level BTB differs in that it must approximate when content is missing rather than looking for a specific cache line and finding that it is definitely not present in a cache. The prior and current designs assume content is missing when three qualified successive BTB1 search attempts result in no predictions being made. The z15 design will additionally proactively fire up and search the BTB2 when an unusual number of non-predicted disruptive branches are found in the main pipeline within a given time period. Additionally, certain context changing events will trigger proactive BTB2 searches in order to prefetch and prime the level 1 predictor for the new context.

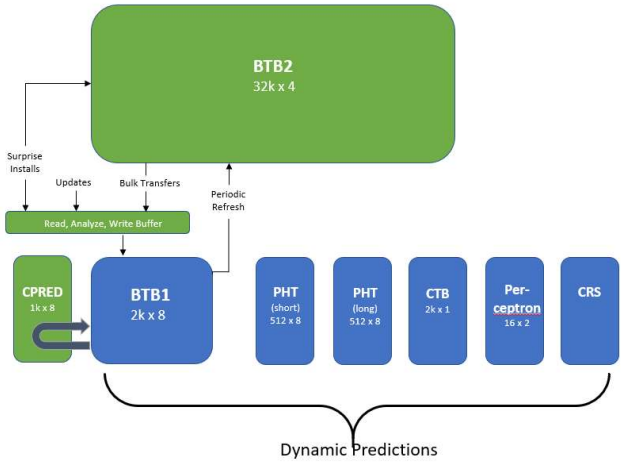

Figure 3. Overview of BPL components

When a BTB2 search occurs, up to 128 branches can be found. A staging queue sits between the BTB2 and BTB1 and is sized to handle the vast statistical majority of BTB2 branch hit transfers. Prior to the z15 design, there was a BTB preload (BTBP) structure that all BTB2 branches were written to. This structure acted as a staging ground and filter that prevented redundant or non-useful entries from overwriting more useful content in the BTB1. Predictions were made out of both the BTB1 and BTBP on prior designs and content was only moved into the BTB1 after a qualified hit in the BTBP occurred. The BTBP also acted as a victim buffer for BTB1 entries that were cast out.

This structure was removed on z15, and the extra silicon area and power budget was used to make the BTB1 larger. The BTB1 had two search ports on prior designs. On z15, the main prediction search path was changed to use a single search port covering twice the address space per search. The second port was re-purposed to provide the filtering that the BTBP implemented in the past by performing a read-analyze-write operation for any BTB1

installs from surprise branches or BTB2 hits. BTB2 hits are first written into a staging queue. Each BTB2 branch in the staging queue performs a read before write using the second search port of the BTB1 and is only written into the BTB1 if the read shows that it does not already exist in the structure.

The original zEC12 design employed a 24K BTB2 and a 4K BTB1 managed in a semi-exclusive fashion. The semi-exclusive design point attempted to maximize the overall capacity of the design by not storing redundant entries at both levels. The behavior was changed to a semi-inclusive algorithm on z15 with a periodic refresh feature that attempts to keep the latest BTB1 learning fresh in the BTB2. Part of the motivation behind this change was the absence of the BTBP victim buffer functionality. Additionally, the drastically larger BTB2 sizes allowed by mature EDRAM technology easily covered the extra entries required for the inclusive design point. In z15 the BTB2 acts as an approximate super-set of the BTB1, and entries that are aging towards being least recently used (likely to be evicted soon) are periodically written back and updated in the BTB2. Rather than burning the power and reading out all prior content of an about to be evicted branch at the time a new entry is going to be installed, the periodic refresh algorithm hides this process under normal existing pipeline search actions. When a normal BTB1 search is performed that does not yield a hit, a global count is incremented. Upon reaching a threshold, the available full content of a no-hit search is analyzed and its next to be evicted (LRU) entry is refreshed back out into the BTB2.

IV. Integration into the Processor

There are many possible ways to integrate a branch predictor into a microprocessor. Section 8 of [7] provides references to various papers describing ways to reduce branch prediction latency. The Alpha EV8 line predictor [11] fetches and predicts up to 16 conditional branches every cycle. Often branch prediction is integrated as part of the instruction fetch or decode stages of the pipeline. These are natural points in the pipeline where an instruction fetch address for a block of instructions or individual instruction boundary addresses are known. Furthermore, actual instruction text may also be available and branch versus non-branch instructions can be distinguished. However, putting the branch predictor in these stages can result in a redirection penalty for taken branches and limits the ability to I-cache prefetch. Instead, the z15 design utilizes an asynchronous lookahead branch predictor. This has been the Z System approach for over 2 decades of processors and is particularly advantageous in the z/Architecture, containing a prevalence of indirect branches, and commercial workloads with large instruction footprints.

The branch predictor is restarted along with instruction fetching, but after that it operates mostly independently. It looks for all upcoming branches based on instruction addresses of branches present in the BTB compared to the current search address, predicts their directions and targets of taken predictions, and redirects itself upon predicting a taken branch.

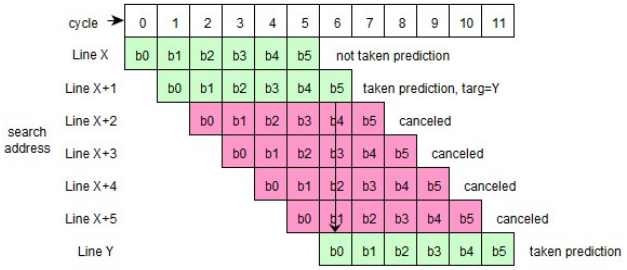

Figure 4. Branch prediction pipeline

The branch prediction pipeline consists of 6 cycles as shown in figure 4. Indexing into the BTB arrays occurs in the b0 cycle, which when superimposed over the z15 core pipeline in figure 1 coincides with the very stage after a restart, but deviates away from the core pipeline after that. An array access cycle is in b1. Metadata from the arrays is obtained in b2, and hit detection and direction application on a per branch basis performed across the b2 and b3 cycles. Ordering of the branches based on their low-order instruction address bits is also done in b3. In b4, the final prediction is prepared, including selection of the appropriate target address provider. The prediction is presented to the consumers, namely the IDU and ICM, in the b5 cycle.

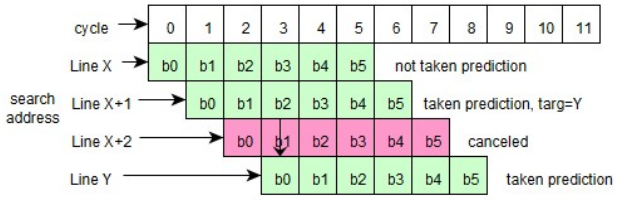

Figure 5. Branch prediction pipeline with CPRED

If there was a taken prediction predicted in the b5 cycle, the pipeline will redirect itself to the target instruction address of the predicted taken branch, performing a b0 index at the target address. This branch prediction pipeline re-indexing can occur preemptively in the b2 cycle with the aid of the CPRED, as shown in figure 5 with the CPRED described later in this section. In the absence of any predicted taken branches (no branches or not taken predictions), the branch prediction pipeline continues to the next sequential lines, with a b0 index possible every cycle.

Since the z15 BTB1 is 8-way set associative, it makes up to 8 branch predictions in a cycle. It simultaneously reports all predicted branches up to a predicted taken branch. This allows the branch predictor to get far ahead of instruction fetching. It can therefore hide the taken branch

redirection penalty entirely. Furthermore, by designing the branch footprint of the BTB to be larger than that of the level 1 instruction cache, branch prediction can serve as an effective cache prefetcher, mitigating and often eliminating the penalty of L1 instruction cache misses by handling them early enough. The branch predictor is the active navigator of instruction fetching.

Although the term asynchronous branch predictor is used because it runs independently of the main processor pipeline, there are points of synchronization. In addition to being synchronized after restarts (e.g., a pipeline flush event), care is taken to ensure that the dispatch stage of the pipeline waits for branch prediction to have been able to provide predictions for those instructions before allowing them to be proceed. While the latency and throughput of branch prediction are designed for it to usually be ahead of instruction fetching, it is not guaranteed to remain so in all scenarios. The branch predictor continuously indicates to the IDU how far it has searched: how many streams of instructions (which end with taken branches), and how many bytes within the current search stream have been searched. This allows the IDU to know when branch prediction has fallen behind. Starting with the IBM z13 machine [17], the branch predictor employs such strict synchronization. Previous mechanisms stalled dispatch when predictions might be coming but were not guaranteed to always stall and wait for them.

In addition to synchronization with dispatch to allow branch predictions to be applied, the predictions themselves are sent to the IDU logic for actually being applied to the instructions. Information about taken predictions are sent to the ICM to steer instruction fetching. Instructions are fetched sequentially until branch prediction notifies the ICM about a taken branch, at which point fetching redirects to the target address. Branch prediction information is also queued within the IFB in the global prediction queue (GPQ) to be used upon completion for performing updates. These consumers of branch predictions contain various queues that store the predictions. When they are full, they tell branch prediction to stop sending. Queues were implemented between the branch prediction pipeline and consumers to prevent the consumers from excessively throttling the search pipeline.

There are various implementation complexities with handling branch predictions being made for taken branches after instruction fetching has already sequentially fetched beyond the instruction address of the predicted branch. The IDU may parse and decode instructions beyond the branch. When this happens, all such younger sequentially fetched instructions are killed and instruction fetching and/or parsing resumes at the target instruction.

Another complexity in this type of design is when a predicted branch does not make sense in terms of the actual instructions at the predicted branch address. For example, a branch prediction in the middle of an instruction, or a branch prediction on a non-branch instruction. These scenarios occur due to partial tagging in the BTB. In such cases the IDU detects the bad branch prediction, causes the front end of the processor to restart, and triggers the bad branch prediction to be removed from the BTB.

When the BTB-based branch predictor provides a prediction for a branch instruction, it is called dynamically predicted. There are some branch instructions that are not present in the BTB at the time the BTB is being searched. When these instructions are dispatched they are called surprise branches. Such branches' directions are statically guessed based on the opcode and other fields in the instruction text. For example, unconditional branches and loop branches are statically guessed taken. Most conditional branches are statically guessed not-taken. For statically guessed taken relative branches, the front end of the processor can generate the restart address. For statically guessed taken indirect branches, the front end shuts down and waits for the target address to be computed by the execution units.

Since z13 it has been possible to change several major details of the branch predictor, such as its throughput and latency, while keeping the branch prediction interfaces to the neighboring units mostly unchanged with only small modifications.

On z13 and z14 the branch predictor was searched at a rate of 64 bytes per cycle in single thread mode by using 2 read ports each of which covered 32 bytes of address space. On z15, the prediction search process was modified to utilize only a single read port and cover 64 bytes of address space with just one search. In SMT2 mode the threads now alternatively search by utilizing this single read port every other cycle. Increasing the BTB1 associativity to 8-way helps ensure there is usually sufficient capacity to track all the branches within a 64 byte section of code in the single indexed row of the BTB1. The branch prediction search process is pipelined and its search rate of 64 bytes per cycle is double the instruction fetch rate of 32 bytes per cycle. This helps keep branch prediction ahead of instruction fetching.

On z15 with a larger BTB1 line size leveraging a single port indexed on a 64B boundary, the prevalence of searches not finding any branch predictions increased over the prior designs. To alleviate such problems and to generally improve performance and power, a way to skip over such unnecessary searches was developed – the SKOOT (Skip Over OffseT) predictor. A field was added in the BTB with each branch to store the minimal 64-byte line skip amount encountered along the target streams before the next predictable branch. It is initialized to an "unknown" state which does not perform any skipping. Over time, it is updated based on where the subsequent branches are found on the target streams, only decreasing except when being updated from the unknown state. SKOOT was also incorporated into the stream-based column predictor (CPRED) which is otherwise similar to z14 [12].

The CPRED is indexed upon entering a new stream. It predicts how many sequential searches to perform before finding the taken branch that leaves the stream, along with the BTB1 way and the redirect address. With SKOOT, that redirect address is the target address plus the SKOOT offset along that target stream. In z15, with the CPRED, the design can predict a taken branch every 2 cycles. When the CPRED does not have a prediction, the design can predict a taken branch every 5 cycles in single thread mode, and every 6 cycles in SMT2 due to port sharing. As in z14, the z15 CPRED continues to predict which branch prediction structures need to be powered up in the target stream.

Figure 6 shows the branch prediction pipeline in the absence of SKOOT, whereas figure 7 shows the pipeline with SKOOT enabled. Both figures employ the CPRED.

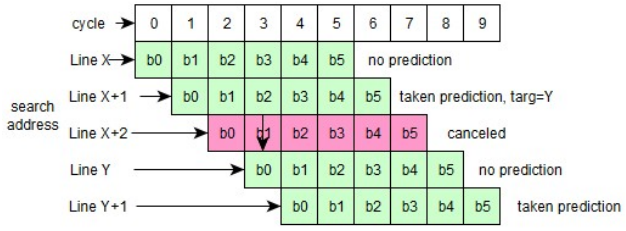

Figure 6. Branch prediction pipeline with CPRED without SKOOT

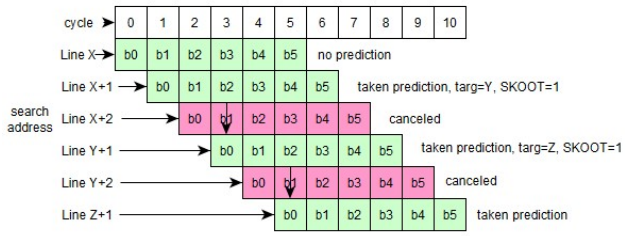

Figure 7. Branch prediction pipeline with CPRED with SKOOT

Branch predictors are updated non-speculatively after instructions complete. Updates are performed as a function of the branch prediction information from the GPQ and branch resolution information from the global completion table (GCT), a structure that tracks instructions through the processor pipeline from dispatch to completion. With the removal of the BTBP in z15, a different way was needed to ensure duplicates are not written into the BTB1 upon BTB2 hits. The solution devised also prevents duplicates from surprise installs. Upon completing surprise branches that need to be installed into the BTB1, they are placed into a write queue. Surprise guessed not-taken, resolved not-taken branches do not get installed. Surprise guessed taken or resolved taken branches do get installed. Similarly, completed branches that need to update the dynamic branch prediction to correct from mispredictions, or strengthen even when correct, also go into the completion write queue. As previously mentioned, BTB2 hits also go into a write queue for installs into the BTB1. Up to one write queue entry per cycle enters into the write queue pipeline. For BTB1 installs, this uses a second read port on the directory to see whether or not the install would create a duplicate. If not a duplicate, or it is an update, the write pipeline then writes the BTB1 and other auxiliary prediction structures. As described in the capacity section of this paper, when BTB1 victims are overwritten it is assumed that they already exist in the BTB2 and do not need to be written into the BTB2 at that time. The periodic refresh mechanism helps ensure this is true.

Because there is large gap in time between when branches are predicted and when they are updated, care must be taken to update the 2-bit counter predictor states in the BHT and PHT appropriately. Otherwise, scenarios like a weak-taken loop branch that is repeatedly predicted taken from this state and is usually resolved taken, but sometimes not-taken can cause updates to the weak not-taken state. The solution to this problem, relied on for many generations in Z System servers, has been the concept of a speculative BHT (SBHT) and speculative PHT predictor (SPHT). These direction predictors have a small number of entries that track weak occurrences of predictions that, when assumed they are correct, will update the corresponding predictor state to strong. Upon a weak prediction, a new entry is written into the SBHT or SPHT. Mis-predicted branches also update or install new entries into the SBHT and/or SPHT. Subsequently, if that BHT or PHT entry is hit on again, the SBHT or SPHT will override the prediction. The SBHT / SPHT entries are removed upon instruction completion or flush of the branches that installed them.

V. Direction Prediction

The bread and butter of the branch predictor is the BTB1, where the BHT and BTB for the direction and target address respectively reside. The BHT is a 2-bit saturating counter that indicates the direction and strength. In general, the BHT maintains the dominant direction of a branch. Any branch that has a dynamic prediction must hit in the BTB1. However, it's not unusual for branches to exhibit both taken and not taken behavior, where that behavior can be determined as a function of the executed code path up to a particular instance of a branch.

Some branches exhibit different direction behaviors that are dependent on the prior code history up to a particular instance of a branch. A quintessential example is the branch that allows a for-loop to keep repeating until a desired condition is met. Generally, the branch is predicted taken to the start of the loop. Once the condition is met, the branch ideally needs to be predicted not taken to continue sequentially beyond the end of the loop instead of repeating the loop code. A hardware structure known as the Global Path Vector (GPV) is employed in the z15 and in prior Z System generations to represent the past history up to a particular branch. The GPV is formed as a function of the

location (instruction address) of the past N taken branches. Prior to z14, the GPV tracked the last 9 taken branches. Since z14 and into z15, a 17 taken branch history is maintained. Since the branch predictor is re-indexed whenever a taken branch is encountered during prediction, not taken predictions do not participate in the formation of the GPV.

As a taken branch is encountered, select bits of the branch's instruction address are hashed down to a smaller 2-bit vector called a branch GPV. This branch GPV is then shifted into the main GPV that represents the history; the oldest branch's branch GPV value is shifted out of the vector when the youngest taken branch's branch GPV is shifted in. A 17 taken branch history represented this way results in a 34-bit GPV vector. In order to correctly predict these conditional branches such as the for-loop above, the branch predictor augments the BHT with a Pattern History Table (PHT). The tagged PHT has been used on z branch predictions since the z196 [15] as a single table, but starting with z15 it exploits a variation of the TAGE algorithm based off of [8]. Two TAGE PHT tables are employed in z15– a short and a long table – each 512 rows deep per BTB1 way for a total branch capacity of 8K. Both tables coincidentally have the same number of rows and are indexed similarly as a function of the GPV and branch instruction address, but the short TAGE PHT table's index function includes the most recent 9 branches in the GPV history, whereas the long TAGE PHT table's index function includes the most recent 17 branches in the GPV history.

Installation into the TAGE PHT tables occurs whenever a predicted branch resolves with a branch wrong direction. The BTB1 is updated to indicate the branch has exhibited taken and not taken behavior (bidirectional), and installation into the TAGE PHT tables ensues. Each TAGE PHT entry contains a usefulness count value. An existing TAGE PHT entry can only be overwritten if the current usefulness value is 0. When a new TAGE PHT install is attempted, the table with the usefulness value of 0 is selected; if both tables have a usefulness of 0, the short table is favored over the long table at a ratio of 2:1. If the short TAGE PHT table resolved with a branch wrong direction, an install is attempted into the long table.

At prediction time, the TAGE PHT will be used to override the BHT direction prediction if the BTB1 indicates that the branch is allowed to use the TAGE PHT and if there is a TAGE PHT hit. Once the prediction is made from the TAGE PHT, the branch prediction tracking structure (Global Prediction Queue) stores the prediction state of the branch until it completes. The GPQ also stores the alternate prediction – what the branch prediction would have selected for the direction prediction in the absence of the primary provider – in this case, the TAGE PHT. Once the resolution of the branch is known at completion time, the usefulness value of the TAGE PHT entry used is assessed and updated as follows.

If the TAGE PHT prediction was correct but the alternate predictor (which could be either the short TAGE PHT or the BHT) was wrong, then the usefulness is incremented. Conversely, if the TAGE PHT prediction was incorrect but the alternate predictor would have been correct, then the usefulness is decremented. If the TAGE PHT prediction was either correct or incorrect, and the branch resolution would have been the same if the alternate predictor was used, then the usefulness is neutral and unchanged. This attempts to keep TAGE PHT entries that provide the correct prediction over an alternate predictor's incorrect direction prediction to persist longer, reducing the likelihood of replacement in the near future, or make it more susceptible for replacement if the TAGE PHT prediction is worse than the alternate.

In general, if a TAGE PHT table hits, it can provide the prediction. However, a weak TAGE PHT prediction can sometimes be detrimental, particularly after a context switch [8]. As such, weak filtering is employed. Before allowing a weak TAGE PHT to provide the prediction direction, a weak prediction counter is checked. If the count is above a predetermined threshold, the weak TAGE PHT is allowed to provide the prediction. There can also be cases where the long TAGE PHT table is in a weak state, but the short table is strong. In this instance, the short table is selected as the provider. This does not apply to strong TAGE PHT predictions; they are not subject to such filtering.

While the TAGE PHT tables generally perform a good job predicting when a conditional branch should be predicted taken versus not taken based on the prior code path history, it struggles with others. Starting on z14, a neural-network based perceptron branch direction prediction was introduced to help augment the pattern-based auxiliary direction prediction [9][18]. The z15 perceptron carries this design forward.

As described in [9] and in patents [13][14], the perceptron improves direction accuracy for branches that are not otherwise predictable with sufficient accuracy by BHT or PHT structures. It employs a table of weights that correspond to path history bits from the GPV. Unlike direction predictors, such as the PHT, that equally weight the importance of all instruction addresses in the history, the perceptron attempts to find the history bits that most heavily correlate to the direction outcome of a particular branch. Since the perceptron's focus is on hard to predict branches, only 32 perceptron entries are employed, implemented as a 16 row by 2 way set associative structure, and shared between the two threads.

The perceptron direction prediction is formulated by calculating the sum of several weights. The sign of the final sum determines the direction of the prediction, and the magnitude of a particular weight is representative of how highly correlated it is; the higher the magnitude, the greater its influence on the final direction outcome. Each weight corresponds to a bit in the GPV. The value of the GPV bit

corresponding to a particular weight at the time of the prediction determines the sign that should be used for that weight before summation occurs. The perceptron is indexed as a function of the BPL search address. At completion time, the weights are either incremented or decremented as a function of the resolution direction of the branch. If the branch resolved taken, all weights that correspond to a GPV bit of 1 are incremented; others are decremented. If a branch resolved not taken, all weights that correspond to a GPV bit of 1 are decremented; others are incremented. A process called virtualization is used to reduce the amount of storage required; 2:1 virtualization permits 34 GPVs to map to 17 weights. When a perceptron entry is created, it must first learn by increasing the magnitude of its individual weights. Over time, some weights may end up with a magnitude close to zero. For those poorly correlating weights, virtualization occurs where a different branch in the execution history is assigned to that weight. Since the weight's previous GPV bit was not highly correlated, the perceptron tries a different predetermined bit in the GPV to see if it will be highly correlated.

A small number of perceptron entries is sufficient to provide significant prediction accuracy improvements because it is often the case that a small subset of branch instruction addresses is responsible for a disproportionately larger proportion of the total mispredictions in a workload. It is critical to keep the right branches in the perceptron table; namely those that are predicted well by the perceptron and poorly by the alternative TAGE PHT + BHT predictors. A replacement algorithm based on usefulness and protection limit maintained with each of the entries accomplishes this. A new perceptron entry must learn before it is considered useful and permitted to provide the direction prediction. In order to learn, the new entry must be given a chance to persist and avoid replacement. Every new perceptron installation starts with a non-zero protection limit. A non-zero protection limit prevents the perceptron entry from being overwritten prematurely. Every time the entry is attempted to be replaced, the protection limit is decremented. It can only be replaced when the limit is 0. Furthermore, a usefulness value is accrued as the perceptron strengthens. Across all perceptron entries in a row, the least useful entry, the one with the smallest usefulness value, is selected as the entry to be replaced, provided it has a protection limit of zero. Usefulness is incremented in cases where the perceptron is correct and the alternate predictor is incorrect. When usefulness is below a threshold, it is incremented even when the perceptron is wrong when the alternate prediction is also wrong.

As the BPL search process hits perceptron entries, their direction prediction is initially not used but is tracked through the pipeline in the GPQ. At completion time, the direction prediction of the structure that provided the branch prediction (PHT or BHT) is compared to the perceptron's alternate prediction. If the provider was wrong but the perceptron was correct, the perceptron's usefulness is incremented; in the opposite case, the usefulness is decremented. Once a perceptron entry's usefulness is above a predetermined global threshold, the perceptron becomes the provider and the PHT or BHT becomes the alternate. The same process can later be used to demote the perceptron when the PHT or BHT start to predict better.

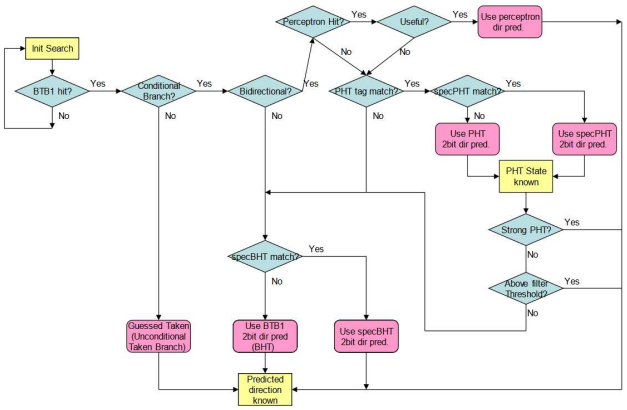

Figure 8. branch direction determination

The flowchart in figure 8 depicts the method in which direction branch predictor – BHT, TAGE PHT or perceptron – is selected as the provider of the direction prediction. In general, for any branch that is not marked in the BTB1 as unconditional, the branch is first checked to see if it can use the auxiliary direction predictors. If the bidirectional state is set, the perceptron is given the first chance to provide the prediction. If it hits in the perceptron and is useful, the perceptron is the provider. If it either doesn't hit in the perceptron or hits but is not useful, the PHT tables are then consulted. The speculative short and long TAGE PHT tables are consulted first, and if hit, either the short or long speculative TAGE PHT direction is the provider. If the branch is not found in the speculative TAGE PHT tables, the main short and long TAGE PHT tables are used to obtain the TAGE PHT direction prediction. If the prediction is strong, it is the provider. However, if the TAGE PHT direction prediction is weak, it is subject to weak filtering; either the other TAGE PHT table (speculative or not) would provide the prediction based on the weak prediction rules, or the direction prediction falls into the hands of either the main BHT in the BTB1 or the speculative BHT. As per all predicted branches, a BTB1 hit must occur in order to provide any prediction.

VI. TARGET PREDICTION

In addition to having a branch's direction be dependent on the history of the executed code path, a

branch can also have more than one branch target address. Such a branch is known as a changing target or multi-target branch. The quintessential changing target branch is where a common function shared between two different parts of instruction code needs to branch back the point where the function was accessed. IBM Z branch predictors support these kinds of multi-target branches with the changing target buffer and a call/return stack heuristic predictor.

Each of the logically 2K entries of the CTB contains one entry, which like the BTB1, contains a target address. There are virtual instruction address tag bits contained with each entry as well; a CTB entry can only be used if there is a tag match for the current address space undergoing search. The CTB has been present since the z10 branch predictor [20]. In the z15, there are four, 512 entry SRAM arrays that comprise the CTB tables. For any given branch prediction search with a particular GPV pattern, there could be one CTB branch target address provided. The CTB is indexed solely as a function of the prior code path history as represented in the GPV. Prior to z15, the 9 taken branch history of the GPV formed the CTB index. On z15, the 17 taken branch history of the GPV now forms the CTB index.

Despite not having explicit call and return instructions in the z/Architecture, the branch predictor does maintain a one-entry deep call/return stack for branches that exhibit call and return behaviors [10]. The logic uses branch target distance as a heuristic to try to identify branches that behave like calls and returns. The mechanism operates in two similar components – detection and prediction. The z14 introduced a basic call/return stack predictor [9], which was further enhanced on z15.

A mechanism at the back-end of the processor pipeline is used to try to detect pairs of branches behaving like calls and returns. For every completed resolved taken branch, the distance between the branch instruction address and its target address is calculated. If it exceeds a predetermined threshold number of byte blocks, the next sequential instruction address (NSIA) after the completed taken branch is stored onto a one-entry deep stack and marked valid. All subsequent completed resolved taken branches have their target address compared to the NSIA value on the stack, but also to different byte offsets from the NSIA: 2, 4, 6 and 8 bytes. If a completed taken branch's target address matches any of those valid, NSIA plus offset values, that branch has its BTB1 branch metadata updated to indicate that it's a possible return instruction with a particular offset from the NSIA: 0, 2, 4, 6, or 8 bytes. The stack can continually be updated even while valid if another branch matches the distance heuristic, as long as it doesn't otherwise match the NSIA plus offset already on the stack. When a possible return is detected and matches any of the NSIA plus offsets, the stack is marked invalid.

At prediction time, the same distance heuristic is employed to setup a prediction-time, one-entry deep stack. Any predicted taken branch those target address exceeds the same predetermined distance threshold has it's NSIA placed onto the stack and marked valid. If a subsequent branch that is marked as a return is then predicted from the BTB1, and the one-entry deep prediction stack is valid, the NSIA obtained from the stack plus offset value obtained from the BTB1 is used as the branch's target address; the stack is then invalidated.

For the CTB or CRS to provide the target address of a dynamically taken branch, the branch must first have been dynamically predicted and resolved with a wrong target. The BTB1 always has a target address, and with a limited number of CTB entries and the risk of incorrectly marking a branch as a return, the desire is to use as few auxiliary predictors as needed. Once a dynamically predicted branch resolves with a wrong target, a field in the BTB1 is updated to indicate this branch is no longer a single target branch, but now a multi-target branch. The CTB and CRS can only provide a target address prediction if the BTB1 indicates the branch is multi-target. Selecting between CRS or CTB for the target address is done using the algorithm depicted in flowchart in figure 9.

If the branch is marked as a multi-directional branch in the BTB1, the call/return stack, provided it can be used and the prediction stack is valid, will provide the target address. If not, the CTB will be used provided there was a CTB hit. If neither the call/return stack nor the CTB provides the target address prediction, the BTB1 is used.

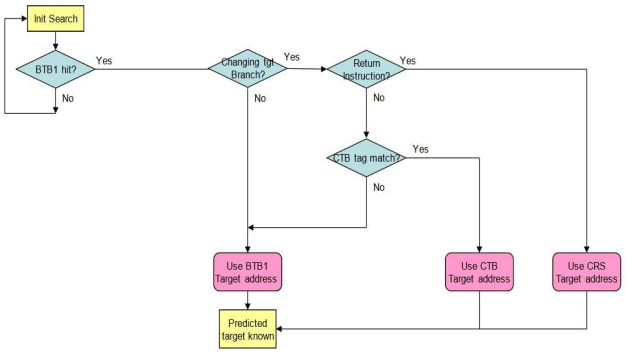

Figure 9. Branch target determination

Whenever a BTB1 predicted branch target resolves with a wrong target, the BTB1 is updated with the correct target address, and a CTB entry is installed. Whenever a CTB predicted branch target resolves with a wrong target, the CTB alone is updated with the correct target address. If the CRS is used for the target address prediction but resolves with a branch wrong target, the mechanism failed to properly identify a call/return pair. The branch's BTB1 metadata is updated to indicate that this branch was thought to be a return, but it resolved with a wrong target. Therefore, the branch will not use the CRS in the future; the CRS is blacklisted for this branch. Instead, future predictions of the multi-target branch would rely on the CTB or BTB1 for the target address. Such blacklisted

branches are given a second chance through amnesty – every Nth completing wrong target branch that was blacklisted is removed from the blacklist, provided it still resulted in a successful call/return pair matching. Subsequent predictions of this possible return branch are then allowed to use the CRS again.

As mentioned in section IV, the CPRED is used to power down auxiliary prediction structures if they are not needed for a new stream. If the bidirectional state or multi-target state is not set, the PHT, perceptron and CTB are subject to power down via the CPRED.

VII. Verification Methodology

The z15 verification effort was split into a hierarchy of levels: designer level, unit level, chip and system level. Verification efforts were synchronized with the progress of the design teams. Before the first hardware components were implemented, a parameterizable, sizeable performance modeling environment was created in C++ to evaluate the performance of different design options. As an input to the performance modeling environment, instruction traces of workloads that run on a mainframe system were read.

With the first implementation of hardware components, the functional verification cycle started. Simulation-based and formal verification techniques were applied. Most of the functional verification at unit level was based on cycle simulation of constrained random stimulus environments at a signal level. For better simulation performance and scalability, a cycle-based simulator was selected over an event-based simulator. The design team applied the necessary hardware language restrictions, like avoiding delay specifications or complex sequential constructs to allow for using a pure cycle-based simulation engine. Event simulators evaluate the model of a design under test (DUT) as a series of events over time using queues. Processing these events using the queues can result in signal value changes of the model that create new events which trigger further processing until the model stabilizes [1].

Cycle-based simulation engines use much simpler algorithms, hence the speed up, but require the hardware language restrictions mentioned above [1]. Constrained random verification environments support a symbolic language that allows a user to specify constraints in a parameter file. These constraints are interpreted by driver code of the verification environment. Constraints restrict the random behavior of drivers and allow the user to determine the probability of certain events, signal values or sequence of signal values to stimulate the DUT. Higher level functional verification on the chip and system level used instruction stream inputs that were generated by testcase generation tools that allowed for constrained random selection of instruction groups. The hierarchical approach of simulation code development allowed for the enablement of monitors, developed at unit level, also in higher level verification environments.

To verify the z15 branch prediction unit, a mix between black box and white box verification strategies was chosen. Black box verification approach involves monitoring and checking of the DUT only at the interface level. White box approach involves monitoring and checking of interface and internal signals of the DUT [1]. While certain design aspects can be functionally checked at architectural level using a black box approach, a white box approach was used to check performance related design behavior. For a cycle-based constrained random simulation environment, models of important hardware structures were created in C++ that were driven by internal hardware signals and were in lockstep with the hardware. These hardware signal driven models in C++ were more of an abstraction of the internal hardware workings than an independent reference model with values set by verification code only. Hardware implementation errors would corrupt values in these models. At certain events, monitors added expect values to the hardware signal driven models in C++. At certain checkpoint events, monitors crosschecked these expect values with the actual state of the hardware driven model. This is illustrated in figure 10. Expect values at a certain checkpoint were never forwarded as input to the next checkpoint stage. Crosschecking was done using a modular approach that allowed for disabling certain checkers via parameter files while there were pending fixes for hardware defects.

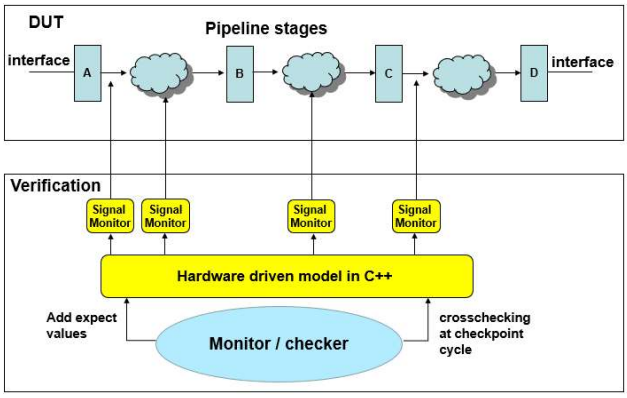

Figure 10. White box verification

The driver based constrained random unit simulation environment also employed preloading of the branch predictor arrays like BTB1 and BTB2 to initialize states into those arrays which would otherwise be difficult to get to or would take a large number of simulation cycles to reach. This helped to find corner-case defects early in the verification bring-up. This preloading code was capable of loading these arrays either from a static test case with a predetermined instruction stream, or from a dynamic test

that generates at cycle zero a random set of instructions to be executed during the test.

The branch prediction verification checking and hardware driven models were broadly split into a) search (or read) side monitors which handled indexing and prediction, and b) write side monitors that handled the installs and updates into the different prediction arrays as shown in figure 11. In this figure, the Design Under Test is the Branch Prediction logic shown as rectangular boxes consisting of the BTB arrays and its control logic and queues. The verification components are shown as oval boxes consisting of Interface Monitors that abstract signals in the design into Transactions and the Unit Monitors that contain the hardware driven reference models. The search side and write side monitors were decoupled from each other to keep the modularity mentioned above. The reference prediction arrays (like BTB1) which were used by the search side monitors were always updated by hardware write values and not by the expected writes that the write side monitors generated.

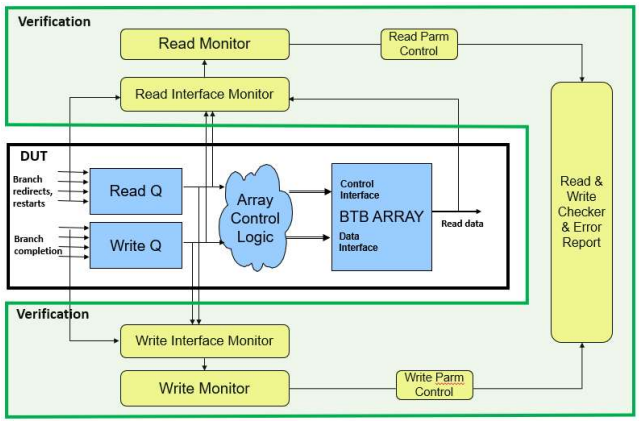

Figure 11. Verification environment with decoupled read and write checking

White box verification strategies require a bigger implementation effort than typical black box environments and bear the risk of a big impact on the verification environment in case of a major design change. Their advantage is early detection of performance related hardware problems close to the source of failure. Many performance problems don't cause functional failures that can be detected using a black box architectural level verification environment.

VIII. Conclusion

The IBM z15 processor employs an example of a modern, enterprise-class branch predictor. Essential to keeping the processor pipeline full, the branch predictor uses several strategies to optimize correct branch direction and branch target predictions.

As part of the development process, the design described in this paper underwent rigorous scrutiny to ensure that all design goals, such as architectural function, area, power, frequency, and performance, were met. Paramount to the branch predictor, performance validation was essential. This performance validation helped the development team select the most appropriate design to meet all requirements by simulating the various algorithms in a cycle-accurate environment where various parameters could be configured.

Final performance metrics are determined by running representative workloads on the z15 and prior vintage Z system machines. On common LSPR workloads, the average number of mispredicted branches per thousand instructions decreased 9.6% between the z14 and z13, and another 25% between the z15 and z14.

In the end, the z15 branch prediction design met all requirements, operating continually at 5.2 GHz and delivering significant performance gains to the mainframe platform for this and for generations to come.

Acknowledgment

We would like to thank the review committee and our colleagues Kevin Shum, Christian Jacobi, Daniel Lipetz, Steve Hnatko, Varnika Atmakuri, Suman Amugothu, James Cuffney, Ajit Honnungar, Challa Lakshmi, David Greenberg, Mary Jo Saccamango, Sterling Coleman, Brandon Bruen, Ching Zhou, Jane Bartik, and Jang-Soo Lee for their assistance on this paper and for their efforts on the z15 branch predictor.

References

[1] Bruce Wile, John Goss, Wolfgang Roesner, Comprehensive Functional Verification: The Complete Industry Cycle. Morgan Kaufmann, San Francisco, CA. 2005. pp. 86-89, 182, 231, 646.

[2] "Large Systems Performance Reference for IBM Z." https://www-01.ibm.com/servers/resourcelink/lib03060.nsf/pages/lsprindex?OpenDocument

[3] IBM Corporation, "z/Architecture principles of Operation," 12th ed., SA22-7832-12, August 2019. [online]. Available: http://www-01.ibm.com/support/docview.wss?uid=isg2b9de5f05a9d57819852571c500428f9a

[4] "System 360 From Computers to Computer Systems." [online]. https://www.ibm.com/ibm/history/ibm100/us/en/icons/system360/

[5] IBM Corporation, "Power ISA." March 2017. [online]. https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

[6] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, Anthony Saporito. "Two Level Bulk Preload Branch Prediction." IEEE 19th International Symposium on High Performance Computer Architecture, 2013.

[7] Mittal, A Survey of Techniques for Dynamic Branch Prediction. 2018. Arxiv.org/abs/1804.00261

[8] A. Seznec. “The L-TAGE Branch Predictor.” Journal of Instruction Level Parallelism, May 2007. https://www.jilp.org/vol9/v9paper6.pdf

[9] C. Jacobi, A. Saporito, M. Recktenwald, A. Tsai, U. Mayer, M. Helms, A.B. Collura, P.-K. Mak, R.J. Sonnelitter III, M.A. Blake, T.C. Bronson, A.J. O’Neill Jr., V.K. Papazova. “Design of the IBM z14 Microprocessor.” IBM Journal of Research of Development, Vol. 62 March/May 2018.

[10] James Bonanno, Michael Cadigan, Jr., Adam Collura, Daniel Lipetz, Brian Prasky. “Distance-Based Branch Prediction and Detection.” Patent pending, application number 15/165,395. Published Nov 2017.

[11] A. Seznec, S. Felix, V. Krishnan, Y. Sazeides. “Design tradeoffs for the alpha EV8 conditional branch predictor.” Proceedings 29th Annual International Symposium on Computer Architecture. 2002.

[12] James Bonanno, Michael Cadigan Jr., Adam Collura, Christian Jacobi, Daniel Lipetz, Anthony Saporito. “Stream based branch prediction index accelerator with power prediction.” Patent. US10430195.

[13] James Bonanno, Michael Cadigan Jr., Matthias Heizmann, Brian Prasky. “Auxiliary branch prediction with usefulness tracking.” Patent. US 9507598.

[14] James Bonanno, Michael Cadigan Jr., Adam Collura, Matthias Heizmann, Daniel Lipetz, Brian Prasky. “Perceptron branch prediction with virtualized weights.” Patent. US 9442726.

[15] F. Busaba, M.A. Blake, B. Curran, M. Fee, C. Jacobi, P.-K. Mak, B.R. Prasky, C.R. Walters. “IBM zEnterprise z196 microprocessor and cache subsystem.” IBM Journal of Research and Development, Vol.56, 2012.

[16] K. Shum, F. Busaba, C. Jacobi, “IBM zEC12: The third-generation high-frequency mainframe microprocessor,” IEEE Micro, vol. 33, Mar./Apr. 2013.

[17] B.W. Curran, C. Jacobi, J.J. Bonanno, D.A. Schroter, K.J. Alexander, A. Puranik, M.M. Helms, “The IBM z13 multithreaded microprocessor.” IBM Journal of Research and Development, vol. 59, Jul./Sep. 2015.

[18] D.A. Jimenez, C. Lin. “Dynamic branch prediction with perceptrons.” Proce. HPCA 7th Int. Symp. High Perform. Comput. Architect., 2001, pp. 197-206.

[19] P.-Y. Chang, E. Hao, and Y. N. Patt,“Target Prediction for Indirect Jumps,” in Proceedings of the 24th International Symposium on Computer Architecture, pp. 274-283, June 1997.

[20] C.-L. Shum, F. Busaba, S. Dao-Trong, G. Gerwig, C. Jacobi, T. Koehler, E. Pfeffer, B.R. Prasky, J.G. Rell, A. Tsai. “Design and microarchitecture of the IBM System z10 microprocessor.” IBM Journal of Research and Development, vol. 53, 2009.