WEP Warwick Evans Publishing

International Journal of Computer Science and Information Technology

ISSN: 3005-9682 (Print), ISSN: 3005-7140 (Online) | Volume 6, Number 2, Year 2025
DOI: https://doi.org/10.62051/ijcsit.v6n2.05
Journal homepage: https://wepub.org/index.php/IJCSIT/index

Optimized Design of Branch Predictors Based on Hummingbird E203

Zhiqiang Liu, Shuai Liu, Junyi He *, Dongliang Yuan, Jiawei Liu

College of Computer Science & Technology, Henan Polytechnic University, Jiaozuo 454000, China
*Corresponding Author: Junyi He

ABSTRACT

As the data processed by modern processors becomes increasingly complex, enhancing the computational capabilities of processors has become an unavoidable challenge. In this context, the optimization of branch predictors becomes important. As an open-source and low-power RISC-V processor core, The Hummingbird E203 has demonstrated great potential in the fields of education, academic research, and IoT. However, limited by the lack of prediction accuracy of the static branch predictor, the performance of the Hummingbird E203 also suffers a bit. In order to improve the branch prediction accuracy, this paper optimizes the original branch predictor under the premise of controlling the hardware overhead, and implements four dynamic sub-predictors to improve the performance of Hummingbird E203. The experimental results show that the highest prediction accuracy among the four predictors is the hybrid predictor consisting of the Gshare branch predictor and the local history-based branch predictor. In the test results of Dhrystone and CoreMark, the average prediction accuracy of the hybrid branch predictor has increased by 15.5%, and the score has increased by 1.76%.

KEYWORDS

Hummingbird E203; Dynamic branch predictor; RISC-V; Hybrid predictor

1. INTRODUCTION

In the architectural design of modern high-performance processors, optimization of the branch prediction module has become a key technical direction for enhancing instruction parallelism [1]. When the branch predictor mispredicts, the pipeline must be flushed to discard erroneous instructions and refetch the correct path for execution. This process wastes processor resources and significantly degrades overall performance [2]. Consequently, improving branch prediction accuracy represents a critical focal point for advancing processor performance.

The Hummingbird E203 is one of the processors in the Hummingbird E200 series, adopting a two-stage variable-length pipeline architecture. In terms of the instruction set, the Hummingbird E203 not only supports the RV32I basic instruction subset but also is compatible with multiple extended instruction sets [3]. The Hummingbird E203 processor targets the IoT (Internet of Things) domain, demonstrating superior performance and power efficiency compared to commercial processors in the same class such as the ARM Cortex-M series [4]. However, constrained by its lightweight static branch predictor, the E203 exhibits room for improvement in branch prediction accuracy.

To address the unsatisfactory branch prediction accuracy of the Hummingbird E203, this paper leverages the hardware-software platform of the Hummingbird E203 to improve the original lightweight static branch predictor and implement four dynamic branch predictors, which are: (1)

BTB-based branch predictor; (2) Gshare branch predictor; (3) Local-history-based branch predictor; (4) Hybrid branch predictor. Through benchmark test programs, the actual prediction accuracy, scores, and resource overhead of these branch predictors are measured. By comprehensively evaluating these indicators, the branch predictor with the optimal energy efficiency ratio is selected.

2. RISC-V AND HUMMINGBIRD E203 ARCHITECTURE

2.1. RISC-V Instruction Set Architecture

As a fully open-source and royalty-free RISC ISA, RISC-V delivers unprecedented development and innovation potential to the semiconductor industry through its reduced instruction set and architectural simplicity. Furthermore, it offers exceptional extensibility and powerful customization capabilities [5-6]. Since its inception in 2010, RISC-V has gradually gained favor with more and more enterprises. The Xuantie 910 processor developed by Pingtouge Semiconductor adopts a 12-stage out-of-order execution pipeline with 3 issuances and 8 executions, suitable for multiple fields such as 5G communication and intelligent manufacturing [7]; At the RISC-V Summit, the Institute of Computing Technology of the Chinese Academy of Sciences released the high-performance processor "Xiangshan", which adopts an 11-stage out-of-order execution pipeline and has a maximum frequency of 1.3 GHz [8]. The second-generation architecture "Nanhu" is comparable in performance to the ARM A76 and is mainly targeted at fields such as communication equipment, industrial control, and automotive electronics [9]. There are many other open-source processor cores, such as Rocket Core [10], BOOM Core [11], and Hummingbird E203.

The RISC-V instruction set includes six conditional branch instructions: beq, bne, blt, bge, bltu, and bgeu [12]. Their instruction formats are shown in Figure 1.

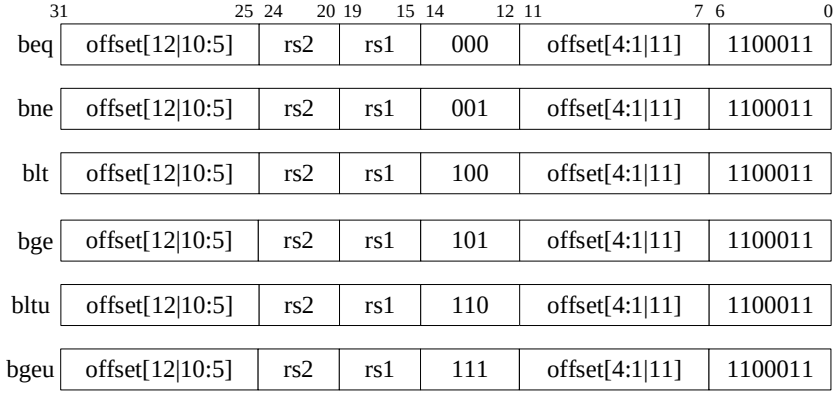

Figure 1. RISC-V Conditional Branch Instruction Format

Whether a conditional branch instruction performs a branch depends on the comparison result of the values in registers rs1 and rs2. The branch target address is determined by the current instruction PC and the offset, and the calculation method is:

$$Target = PC + sext(offset) \tag{1}$$

Among them, sext(offset) is the sign-extended immediate value [12].

2.2. IFU Architecture of Hummingbird E203

The Hummingbird E203 adopts a two-stage variable-length pipeline architecture. The first stage is the IFU (Instruction Fetch Unit) instruction fetch stage, which fetches instructions from the ITCM (Instruction Tightly-Coupled Memory) or BIU (Bus Interface Unit) based on the current PC and stores them in the IR (Instruction Register) connected to the EXU (Execution Unit) execution stage

interface. Meanwhile, the instructions are transmitted to the Mini-dec module for pre-decoding, followed by static branch prediction to generate the address of the next instruction. The ITCM or BIU is then accessed based on this address to fetch the instruction [13-15]. The architectural diagram of the IFU stage in Hummingbird E203 is shown in Figure 2.

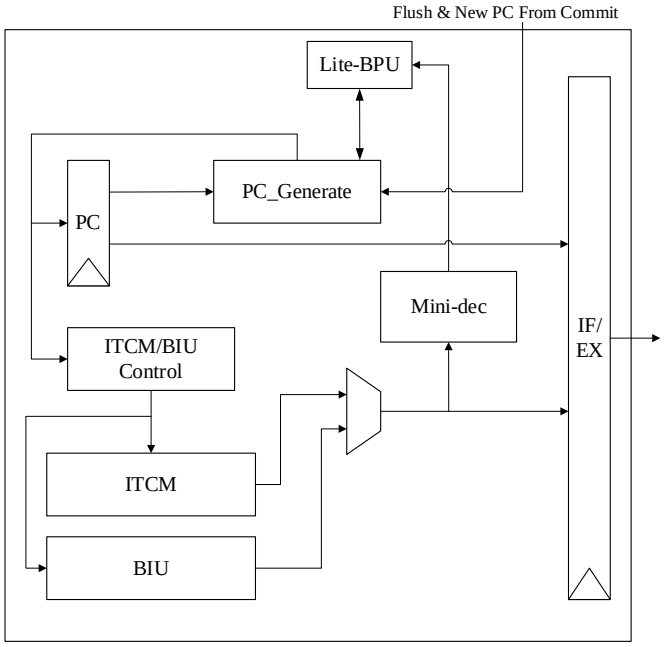

Figure 2. The Structure of Hummingbird E203 Instruction Fetch Unit (IFU)

3. DESIGN AND VERIFICATION OF BRANCH PREDICTORS

This section details the implementation specifics of several dynamic branch predictors.

3.1. BTB-based Branch Predictor

The first dynamic predictor implemented in this paper is the BTB-based branch predictor, which maintains three entries in the form of CAM tables. Among them, btb_pc is used to record the address of the branch instruction for matching with the instruction fetch PC; btb_target is used to record the branch target address of the branch instruction; btb_counter records the branch direction, and finally a register btb_valid is used to record whether each entry is valid.

The btb_counter entry consists of a two-bit saturated counter. In the two-bit saturated counter, there are four different branch prediction states: 00 represents Strongly Not Taken; 01 represents Weakly Not Taken; 10 represents Weakly Taken; 11 represents Strongly Taken. If a branch instruction is taken and the saturation counter has not reached the saturation state, the counter value is incremented by 1; if the branch is not taken and the saturation counter value has not reached the saturation state, the counter value is decremented by 1. The state transition diagram of the two-bit saturating counter is shown in Figure 3, where T and NT denote the taken and not-taken states, respectively.

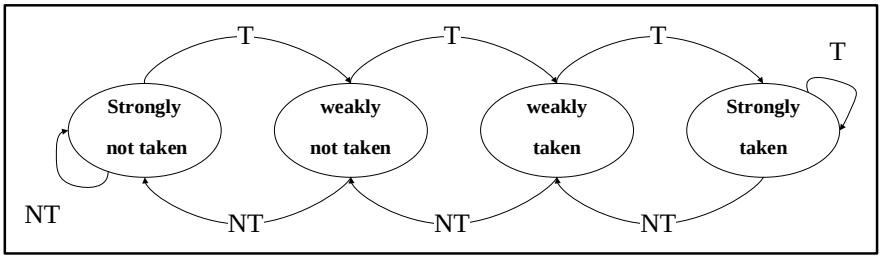

Figure 3. The state machine of a 2-bit saturating counter

During prediction, the predictor compares and indexes the btb_pc entries based on the current branch instruction PC. If the indexing is successful, the corresponding btb_target and the high bit of btb_counter are taken as the prediction results and sent to the IFU instruction fetch module, which generates the nextpc according to the prediction results.

During updates, if there are still invalid entries in the BTB table, one of the entries is selected, the branch instruction PC and the actual branch target are written into it, and the btb_counter is initialized to 10. If all entries in the BTB are valid, the entry with btb_counter = 00 is preferentially selected for replacement and update. This is because during prediction, a branch instruction not found in the BTB is also judged as not taken, which has the same effect as the 00 state.

This approach has a clear issue: if the same PC branches again in the future, a new entry needs to be created, and the btb_counter will jump directly from 00 to 10. This is likely to cause prediction errors. However, the probability of prediction errors occurring next time due to this method is definitely lower than replacing other entries. Finally, if all entries are valid and none of the btb_counters are 00, an entry is randomly selected for replacement and update.

Considering the area of Hummingbird E203, the BTB predictor implemented in this paper uses three 32-item table entries, and PC [31: 2] indexes the btb_pc table entries when predicting. The structure of the BTB-based dynamic branch predictor is shown in Figure 4.

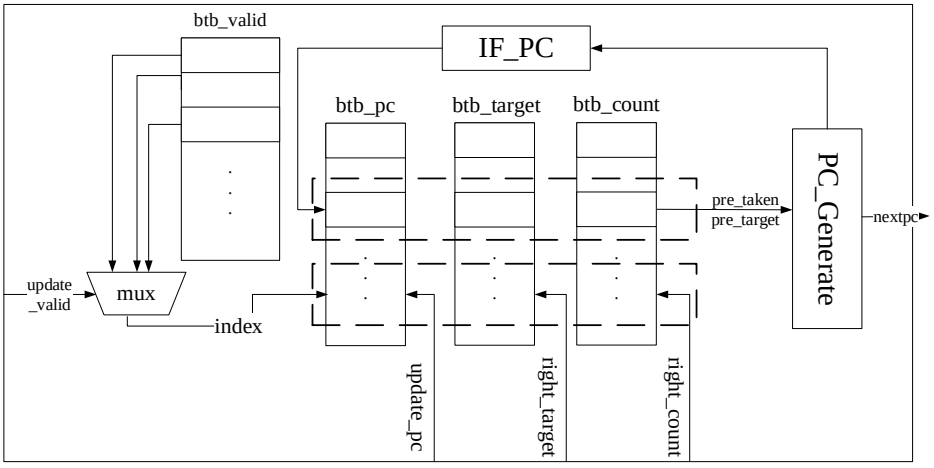

Figure 4. The dynamic branch predictor based on BTB

3.2. Gshare Branch Predictor

The second branch predictor implemented in this paper is the Gshare branch predictor. It contains a Global History Register (GHR) and a Pattern History Table (PHT). GHR records the execution results of the last N branches (1 means jump, 0 means no jump); PHT has a structure similar to the btb_counter table entry of the BTB branch predictor and consists of two saturation counters. Since the Gshare branch predictor can only predict the branch direction and does not provide branch target prediction, this paper implements a Branch-Target Buffer (BTB) to record the addresses of branch instructions and their branch target addresses.

Due to the lightweight feature of Hummingbird E203, the number of PHTs used is limited, so the problem of branch aliasing is inevitable. The Gshare branch predictor implemented in this paper first dissimilates the high and low bits of the PC, and then dissimilates the PC after the dissimilarity operation with the GHR to generate the index value. This can reduce the probability of branch aliasing to a greater extent.

Considering the lightweight design concept of Hummingbird E203, the BTB in this paper has 64 entries, which are indexed using 6 bits from the PC hash operation. The PHT consists of 256 two-bit

saturation counters. For indexing, the high 8 bits and low 8 bits of the IF_PC are first XORed to generate a new PC, and then the new PC is XORed with the GHR to produce the index value. Since the RISC-V instruction set is aligned to 4 bytes, the lowest two bits of the instruction address are 0, which has no practical significance when indexing and increases the probability of branch aliasing, so the lowest two bits are discarded when using the PC-valued dissimilarity operation. The structure of Gshare branch predictor implemented in this paper is shown in Figure 5.

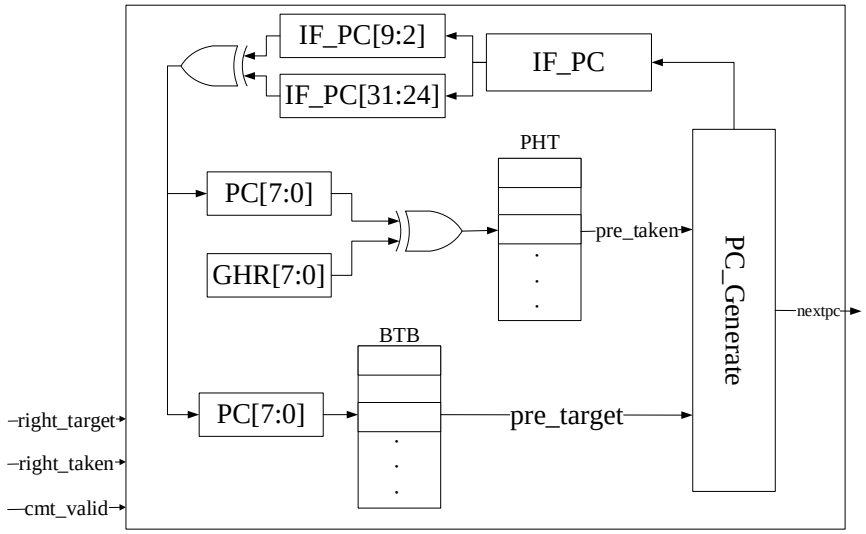

Figure 5. The Gshare branch predictor

During prediction, the upper 8 bits of the PC are XORed with PC [9: 2] to generate a new PC value. This derived PC also serves as the index for the BTB. Then, it is XORed with the GHR to obtain the index value of the PHT. The IFU instruction fetch module generates the nextpc based on the prediction result.

During updates, the actual branch result is shifted left into the GHR (1 is shifted in for taken, 0 for not taken). The two-bit saturation counter in the PHT is located using the index generated by the hash operation, and updated with the actual branch result. If the branch target address stored in the BTB differs from the actual branch target address in the EXU stage, the old value is overwritten with the actual branch target address.

3.3. The Local-history-based Branch Predictor

The third branch predictor implemented in this paper is a local-history-based branch predictor. It consists of a Branch History Table (BHT) and a Pattern History Table (PHT). Traditional local-history-based branch predictors also do not provide branch target prediction. Based on this, this paper implements a Branch-Target Buffer (BTB) to predict the branch target addresses of branch instructions. In the local-history-based branch predictor, the PHT and BTB are constructed identically to those in the Gshare branch predictor. The BHT is composed of multiple Branch History Registers (BHRs), where each BHR records the execution results of a branch instruction over the past N times.

Since the local history-based branch predictor records the historical jump information of multiple branch instructions, the overhead of hardware resources is high. In order to balance performance and power consumption at the same time, the BHT in this paper consists of 64 BHRs, and the number of BTB table entries also has 64 entries, all of which are indexed using 6 bits of the PC value. The number of bits in each BHR is set to 8. The PHT consists of 256 two-bit saturated counters that use the value of the BHR as the index. Same as Ghsare branch predictor, the lowest two bits of PC are discarded for indexing and the BHT is indexed using PC [7: 2] as index value. The structure of the local history based branch predictor implemented in this paper is shown in Figure 6.

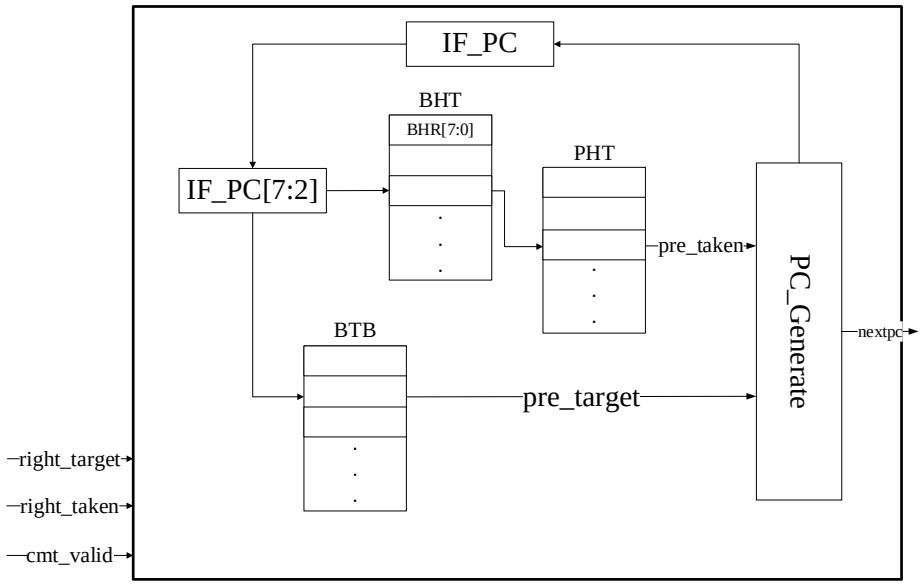

Figure 6. The local-history-based branch predictor

During prediction, partial values of the PC are used to index the BHT to obtain the local historical branch value of the branch. This value is then used to index the PHT to retrieve the value of the two-bit saturation counter and determine the predicted direction. And the lower N bits of the PC are used to index the BTB to obtain the predicted branch target. The IFU instruction fetch module generates the nextpc based on the prediction result.

During updates, the old BHR value is used to index the PHT, and the values of the two-bit saturation counters in the PHT are updated based on the actual branch result. The BHR is then indexed by the PC value, and the actual branch result of the branch instruction is shifted left into the BHR (1 is shifted in for a taken branch, 0 for a not-taken branch). The update process for the BTB is the same as that of the Gshare branch predictor.

3.4. The Hybrid Branch Predictor

The fourth branch predictor implemented in this paper is a hybrid branch predictor. It consists of a Gshare predictor and a local history-based branch predictor, both of which select the result with higher prediction accuracy in the recent time period as the result of the hybrid prediction thereof through a 2-way selector. The selection signal of the selector is generated by a CPHT (Choice Pattern History Table), which has a structure similar to that of the PHT. Each entry in the CPHT is composed of two-bit saturation counters. The selector will use the high bit of the saturation counter as a control signal. When the high bit is 0, the selector will choose the prediction result of the Gshare predictor as the final result; when the high bit is 1, it will choose the prediction result of the BHR predictor as the final result. When the currently selected predictor makes a correct prediction, the CPHT adjusts in the direction of the current tendency. If both the currently selected predictor and the other predictor make incorrect predictions, the state of the CPHT remains unchanged. It only adjusts in the opposite direction of the current tendency if the current predictor makes an incorrect prediction. If the high-order bit of the saturation counter changes, another predictor will be selected instead. Similarly, to reduce the probability of alias conflicts, the hash value of the Gshare predictor is used to index the CPHT. The structural diagram of the hybrid predictor designed in this paper is shown in Figure 7.

During prediction, the prediction results from the Gshare branch predictor and the local-history-based branch predictor are input into the selector, which selects one of them as the final prediction result based on the high-order bit of the CPHT.

During updates, the two predictors are updated according to their respective methods. The actual branch result is compared with the predictions of the two predictors, and the comparison results are used to modify the values of the saturation counters.

The hybrid predictor implemented in this paper uses both the Gshare predictor and the BHR predictor, and can select the predictor with higher accuracy based on specific branching instructions, which compensates for the drawbacks of using the two predictors individually. Accordingly, the hybrid prediction hardware overhead increases.

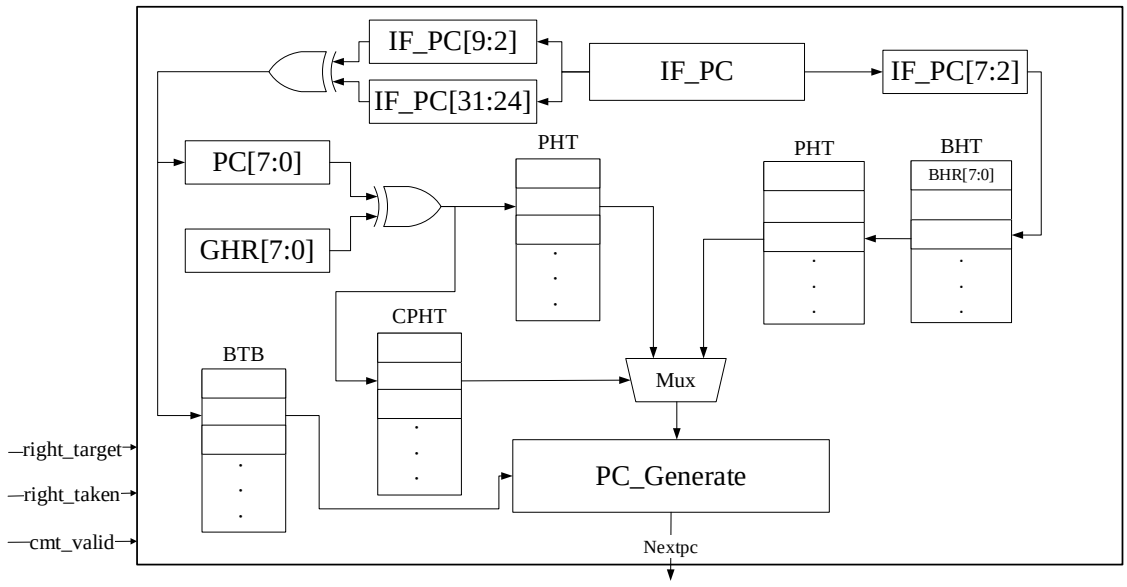

Figure 7. The Hybrid branch predictor

4. ANALYSIS OF EXPERIMENTAL RESULTS

This paper implements four branch predictors using RTL code and ports them to the LITE-BPU module of Hummingbird E203. Test programs such as CoreMark and Dhrystone are executed using the ported processor to show the performance of the branch predictors by recording the accuracy and scores of conditional branch instruction predictions.

This paper uses Vivado software for simulation and functional verification. In the experiments, the original lightweight branch predictor of Hummingbird E203, the BTB-based branch predictor, the Gshare branch predictor, the local-history-based branch predictor, and the hybrid predictor are respectively used to run test programs. After the simulation and functional verification, Vivado software is employed to synthesize the above four branch predictors on the FPGA platform, evaluating their power consumption and hardware resource overhead.

4.1. Performance Evaluation of Branch Predictors

The test results of the two test sets are shown in Table 1 and Table 2.

According to the experimental results shown in Table 1, several dynamic predictors in the Dhrystone test program show a significant improvement in accuracy compared to the original lightweight static predictor. The hybrid branch predictor achieved the highest accuracy and score. Compared with the original branch predictor, its accuracy increased by 11.8%, and the Dhrystone score improved by 2.40%. When indexing, the BTB-based predictor traverses all entries, resulting in low lookup efficiency and a decrease in prediction accuracy. The Gshare branch predictor has lower prediction accuracy than the BHR predictor due to the longer initial learning time of the GHR (Global History

Register). In the Dhrystone program, there are many short loops with few iterations, and a 4-bit BHR is sufficient to record the local branch instruction jumping patterns.

Table 1. Accuracy and Marks of Dhrystone

Predictors	Accuracy (%)	Mark
Original Predictor	84.5	1.289436
BTB Predictor	93.6	1.308761
Gshare Predictor	94.9	1.311042
BHR Predictor	95.5	1.317123
Hybrid Predictor	96.3	1.320349

Table 2. Accuracy and Marks of CoreMark

Predictors	Accuracy (%)	Mark
Original Predictor	67.2	2.129021
BTB Predictor	79.6	2.136103
Gshare Predictor	82.4	2.142470
BHR Predictor	83.8	2.149014
Hybrid Predictor	86.4	2.153249

According to the experimental results shown in Table 2, the hybrid predictor continues to have the highest accuracy and score in the CoreMark test program, with a 19.2% increase in accuracy and a 1.14% increase in CoreMark score compared to the original branch predictor. Compared with the Dhrystone test suite, CoreMark contains a higher proportion of globally correlated branches. As a result, the Gshare predictor using a global branch history register achieves relatively higher accuracy. The BHR predictor has weaker ability to capture global correlations, leading to a narrower gap in prediction accuracy compared with Gshare.

4.2. Evaluation of Branch Predictor Resource Overhead

In this paper, five branch predictors including Hummingbird E203 lightweight predictor are synthesized separately on FPGA platform using Vivado software to get the data related to resource overhead. The utilization of various resources is shown in Figure 8.

According to the analysis of the synthesis results shown in Figure 8, although the hybrid predictor has the highest prediction accuracy, its resource utilization rates in all aspects are also the highest. Compared with the Gshare predictor, the BHR predictor requires an additional BHT (Branch History Table) for maintenance, so it consumes more resources. For complex branch predictors compared to simple ones, the main resource occupation is reflected in LUT and FF. The BTB predictor has an average increase of 0.88% in each resource usage compared to the original static predictor; the Gshare predictor has an average increase of 2.09% in each resource usage; the BHR sub-predictor has an average increase of 3.19% in each resource usage; and the hybrid predictor has an average increase of 6.05% in each resource usage. Combining the branch predictor performance and resource overhead, the Gshare branch predictor outperforms the local history-based branch predictor and the hybrid branch predictor in terms of resource overhead without lagging behind too much in accuracy; the local history-based branch predictor performs well enough to satisfy the needs of Hummingbird E203 and its resource overhead meets the expectations; the hybrid branch predictor has a higher resource overhead, but combined with its higher The hybrid branch predictor has a higher resource overhead, but combined with its higher prediction accuracy, the overall performance is within the acceptable range.

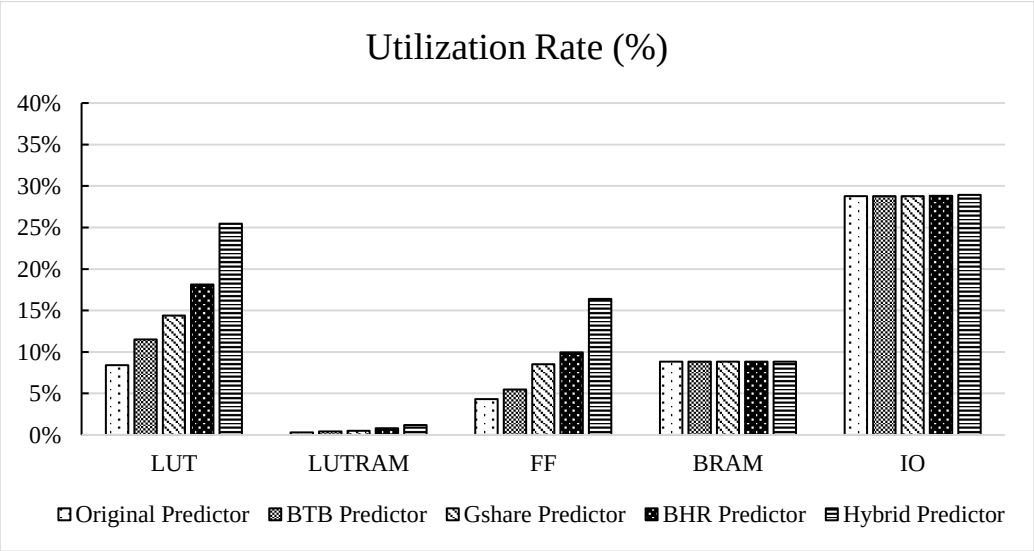

Figure 8. Resource Usage of Branch Predictors

5. SUMMARY

Based on the Hummingbird E203 processor core, this paper designs and optimizes several dynamic branch predictors, successfully improving the accuracy of branch predictors and addressing the problem of low performance of the original static predictor. To further reduce the probability of branch aliasing, this paper improves the traditional Gshare branch predictor by using more complex hash operations. Traditional branch predictors lack prediction of jump targets. This paper improves them by implementing a BTB (Branch Target Buffer) for each type of branch predictor, thus making up for the shortcomings of traditional branch predictors. The improved processor ran the Dhrystone and CoreMark benchmarking programs with up to a 15.5% increase in average branching accuracy, with each branch predictor showing a more substantial improvement.

REFERENCES

[1] Jiemin Li, Shancong Zhang. Research and Performance optimization of five stage pipelined RISC-V processor [J]. Microelectronics & Computer, 2022, 39(03):78-85.

[2] Egan C, Steven G, Quick P, et al. Two-level branch prediction using neural networks [J]. Journal of Systems Architecture, 2003, 49(12-15):557-570.

[3] Zhenbo Hu. Designing a CPU Step by Step. Beijing: Posts & TelecomPres. 201806.426.

[4] Tianchuan Deng, Zhenbo Hu. An ultra-low-power processor pipeline-structure [J]. Application of Electronic Technique, 2019, 45(06):50-53.

[5] Samuel G. Will RISC-V revolutionize computing? [J]. Communications of the ACM, 2020, 63(5):30-32.

[6] Chang Liu, Yanjun Wu, Jingzheng Wu, et al. Survey on RISC-V System Architecture Research [J]. Journal of Software, 2021, 32(12):3992-4024.

[7] Dawei Li. Backend optimization of Go language compiler forXuantie C910 [D]. Hubei:Huazhong University of Science and Technology, 2023.

[8] Institute of Computing Technology, Chinese Academy of Sciences. Xiangshan: Open-source High-performance RISC-V Processor [EB/OL]. [2025-3-25]. https://github.com/OpenXiangShan/XiangShan

[9] Kaifan Wang, Yinan Xu, Zihao Yu, et al. XiangShan Open-Source High Performance RISC-V Processor Design and Implementation. [J]. Journal of Computer Research and Development, 2023, 60(03):476-493.

[10] Bailey S, Rigge P, Han J, et al. A Mixed-Signal RISC-V Signal Analysis SoC Generator With a 16-nm FinFET Instance. [J]. J. Solid-State Circuits, 2019, 54(10):2786-2801.

[11] Celio C, Chiu P, Asanović K, et al. BROOM: An Open-Source Out-of-Order Processor with Resilient Low-Voltage Operation in 28-nm CMOS [J]. IEEE Micro, 2019, 39(2):52-60.

[12] David Patterson, Andrew Waterman. The RISC-V Reader: An Open Architecture Atlas [M]. Beijing: Publishing House of Electronics Industry: 202401. 239.

[13] Shupeng Pan, Youyao Liu.Reciew of RISC-V Microprocessor and Commercial IP [J]. Integrated Circuits and Embedded Systems, 2020, 20(06):5-8+12.

[14] Nuclei System Technology. Hummingbirdv2 E203 Core and SoC [EB/OL]. [2023-06-13]. https://doc.nucleisys.com/hbirdv2/index.html

[15] A. Waterman, Y. Lee, D.A. Patterson, et al. The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA [R]. Berkeley: EECS Department, University of California at Berkeley, 2014.