BADGR: A Practical GHR Implementation for TAGE Branch Predictors

David J. Schlais
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI USA
schlais2@wisc.edu

Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI USA
mikko@engr.wisc.edu

***Abstract*—In this work, we explore global history register (GHR) implementations for Tagged Geometric length (TAGE) style branch predictors with speculative updates. We break down the requirements to both update and recover TAGE predictors' history registers during normal operation and after mispeculation, discussing where various designs exhibit large checkpoint and/or operation overheads. To reduce these inefficiencies, we introduce BADGR, a novel GHR design for TAGE predictors that lowers power consumption and chip area over naive checkpointing techniques by 90% and 85%, respectively.**

I. Introduction

As modern processor pipeline depth and issue width increase, the performance penalties caused by a branch misprediction also increase, motivating the need for highly accurate branch predictors. Virtually every modern processor contains a sophisticated branch predictor to improve processor performance. Branch predictors try to find patterns within a program's execution history in order to accurately predict the outcome of a given branch. This history used for prediction can be local for a given branch, global for the entire program, or often times a hybrid combination of both. The hybrid Tagged Geometric length (TAGE) branch predictors are some of the best predictors proposed in academia [1]. Although several papers describe the performance of TAGE through trace- and execution-driven simulation [2] [3], there is limited prior work on detailed hardware implementations.

Practical designs of TAGE assume speculative updates of a very deep (roughly 2000 branch outcome) GHR for up-to-date predictions. Theoretically, a TAGE-style branch predictor could be built without a recovery mechanism by allowing wrong-path branches to stay in the GHR. However, maintaining accurate GHR bits is critical to preserve TAGE's high prediction accuracy. In this work, we show that zero and partial recovery leads to a large penalty in TAGE misprediction rates. Clearly, an efficient GHR checkpoint/recovery mechanism is necessary to maintain TAGE's accuracy and should also produce fast recoveries to undo GHR corruption from wrong path predictions to improve processor performance.

Conventional global history-based branch predictors face these same requirements, but usually have global history registers on the order of tens of branches, requiring minimal hardware to checkpoint and leaving little opportunity for area or power savings. The large GHR of TAGE-style predictors creates a new challenge for optimizing the storage, retrieval, and recovery of GHR bits. Conceptually, the design of a recoverable GHR may seem straightforward within TAGE by using a circular buffer [4] or by recovering only folded histories, but we explore and identify a number of hidden overheads that accrue when building these designs for TAGE. We additionally explore the overheads in other potential checkpointing designs, analyzing a complex trade-off between low-cost speculative GHR updates, GHR reads, and ease of GHR recovery. The discovery of these hidden overheads motivates a careful re-examination of this problem that has not been proposed or evaluated in prior work.

In this paper, we analyze four potential GHR designs and the trade-offs in the context of TAGE predictors with thousands of history bits. The next four paragraphs give a brief overview of these four designs: traditional checkpointing, backwards shifting, circular buffer, and BADGR.

Traditional Checkpointing. This baseline design is familiar from a long history of predictors that rely on branch outcome history. It implements the GHR as a shift register, where each in-flight branch shifts a new speculative history bit into the GHR. The state of the GHR before the speculative update is saved by writing a full checkpointed copy of the GHR into a checkpoint file, which is typically built from dense SRAM. Detailed analysis in this paper shows that this method is not scalable for TAGE in modern, power-constrained processors. Therefore, an exploration of alternative checkpointing solutions is necessary in order to efficiently implement TAGE.

	Checkpoint	Backwards	Circular Buffer	BADGR
Access type	Fixed	Fixed	Variable	Hybrid
In-flight branch scalability	High	Medium	Low	Low
GHR folding	Low	Low	High	Medium
GHR storage	Medium	Medium	Medium	Low
GHR update	High	High	Low	Low

Fig. 1. Summary of area and power overheads of various GHR implementations for TAGE (Lower is better).

This work was supported in part by NSF Award CCF-1318298.

978-1-5090-5142-7/16/$31.00 ©2016 IEEE

Backwards Shifting. Backwards shifting similarly uses a shift register to update the GHR but reverts back to a previous state by shifting backwards. Control logic is required to shift back multiple positions within a single cycle for designs that support multiple in-flight branches. Both traditional checkpointing and backwards shifting require every bit within the large GHR to shift when a new branch enters the pipeline, which adds significant dynamic power.

Circular Buffer. This design implements the GHR as a collection of flip-flops and a head pointer. The head pointer designates where the most recent history bit resides, and moves as branches are encountered and resolved. The entire GHR can be implicitly reconstructed using this pointer. Circular buffers restore the GHR back to any previous state by moving the head pointer back to its checkpointed previous value. However, due to the large number of GHR entries, we show the circular buffer requires expensive GHR reads needed during GHR folding, and incurs significant overheads that reduce its seeming attractiveness over the previously mentioned designs.

BADGR. To address the inefficiencies of the previous three recovery mechanisms, we propose BADGR: Bank Assignment to Deconflict Geometric Recurrences. BADGR is a novel hybrid checkpointing method that cleverly reorganizes the GHR bits of TAGE into dense SRAM banks through modulo-based bank assignment. This organization takes advantage of TAGE's updating techniques to guarantee the geometric length history bits are conflict-free, regardless of head pointer location. BADGR eliminates the dynamic power of shift register designs while reducing the need for the complex selection logic of the circular buffer design for GHR reads.

This paper makes the following specific contributions:

- We motivate the need for efficient checkpointing and recovery for modern branch predictors with very long histories by showing that lack of full checkpointing causes a significant increase in misprediction rates.
- We identify and discuss key trade-offs in the implementation of various checkpointing schemes for TAGE: namely, checkpoint capacity (proportional to area); cost of updating the history; the cost of accessing the history during normal operation; and finally, the cost of accessing checkpointed state.
- We show detailed analysis for three different checkpointing schemes previously unexplored for TAGE. We explore both the benefits and inefficiencies compared to naive traditional checkpointing (as implemented in current branch predictor designs).
- We evaluate how the number of TAGE tagged tables impacts the cost of various checkpointing techniques.
- Finally, we introduce BADGR, a novel checkpointing implementation that minimizes storage cost even further by utilizing dense SRAM cells instead of flip-flops, and cleverly arranges history bits into banks at a modulo distance that guarantees no bank conflicts during access, while also providing an inexpensive method for recovering to an earlier branch state via a variant of the backwards shifting algorithm.

II. Motivation

Accurate branch prediction provides significant performance gains, especially in longer pipelines [5]. The penalty of branch misprediction continues to grow as issue width and pipeline-depth increases [6]. TAGE achieves high prediction rates by using multiple tables to find patterns over various history lengths (geometrically related), and select a prediction based on the longest history that finds a (PC, history) match. The tags and indices to these tables are determined through hashing the PC with the folded global histories, which are updated through extracting bits from the speculated GHR. We will show that this bit extraction can consume a significant portion of TAGE's overall power and area for certain GHR designs.

To increase performance, many modern processors support out-of-order branch resolution, creating additional complexity for recovering the GHR in logical program order. To guarantee full TAGE recovery for typical modern processors that support out-of-order branch resolution, a nonspeculative "committed" copy of the GHR state is not sufficient to eliminate GHR wrong-path branch corruption. This is because a mispredicted younger branch may resolve prior to the older branch updating the committed GHR. Copying the committed GHR to the front end GHR would not reflect any older branches yet unresolved in the pipeline. For these reasons, a dedicated checkpoint is required for each branch. Full checkpointing refers to checkpointing both the GHR and folded GHRs.

Figure 2 shows that for an out-of-order processor with 192 ROB entries, mispredictions per 1000 instructions (MPKI) of the the L-TAGE (TAGE with a built-in loop predictor) with no checkpointing rise on average by over 37% compared to full checkpointing. For the SPEC2006 benchmarks we simulated, this MPKI penalty is similar to decreasing from 15 tagged tables to 8. Instead of supporting checkpointing, a simple solution to avoid corruption would be to clear the GHR on a misprediction (denoted Flash Zero in Figure 2). However, since TAGE's main prediction benefit comes from storing long histories, it does not perform well with this strategy, resulting in a 54% MPKI penalty.

Since only the folded GHRs are used for accessing TAGE's tagged tables, it may seem that only checkpointing these folded histories would be enough for recovery. However, this method produces a 139% MPKI penalty. Initially, it may seem surprising that this partial restoration performs worse than no restoration, but restoring the folded GHRs without restoring the GHR creates an out-of-sync relationship between the GHR and folded GHRs. That is, since all future history bits currently in the GHR are XOR'd out of the folded histories at the wrong time (GHR position), the notion of geometric lengths is lost. It is statistically very unlikely to recover from the out-of-sync nature between the GHR and folded GHRs. On the other hand, for no checkpointing, even if the GHR and folded GHR contain incorrect values, their GHR positions are still in-sync. The error of a corrupted GHR bit in the folded GHR is eventually removed at the end of the geometric length.

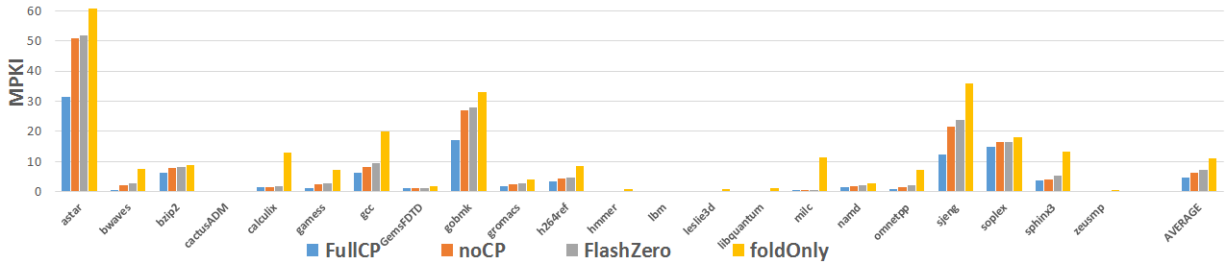

Fig. 2. TAGE MPKI for various levels of checkpointing (Lower MPKI is better).

These severe drops in prediction accuracy motivate the need for recovery mechanisms upon mispeculation.

Branch predictors are also one of the several main contributors to overall processor power [7]. Even simple predictor components such as a branch target buffer (BTB) tend to consume over 10% of a processor's total energy [8]. Additionally, the tagged tables alone from the TAGE predictor, as well as many other proposed branch predictors, can use between 16kB-64kB of storage [9], roughly the size of an L1 cache. Since these numbers are significant relative to the total power and area of a processor, it is important to optimize these metrics as well. No prior publications have determined the power and area required to checkpoint TAGE's long history.

III. TAGE GHR Requirements

When designing TAGE's GHR, it is important to consider the hardware required both during normal operation, as well as upon misprediction recovery. In order to support both of these in a modern system, TAGE's GHR requirements include: (1) Scalability to support an increasing number of in-flight branches, (2) Support for out-of-order branch resolution, (3) Efficient storage of GHR and folded GHR bits, (4) Ability to update the GHR within a single cycle of an incoming branch or misprediction redirect, and (5) Ability to update the folded GHR histories for each geometric history length within a single cycle of an incoming branch or misprediction redirect.

In-Flight Branch Scalability: In-flight branches refer to branches that have entered into the pipeline but have not yet been resolved as taken or not-taken. As with almost all global history-based branch predictors, speculative changes are made to TAGE's GHR upon entering the pipeline. Intel's current Haswell processor keeps 192 instructions in the out-of-order window; if over 10% are branch instructions, then the processor should support roughly 20 in-flight branches (this does not even include branches not yet in the out-of-order window). A practical implementation of TAGE should incur low power and area overheads as the number of in-flight branches supported increases.

GHR folding: TAGE does not use the full GHR for tags in the predictor tables. Instead, to save storage space, each GHR geometric length designated to a tagged table is folded to a smaller number of bits to be hashed with select bits of the incoming PC. The PC hashes with folded histories to create each TAGE table index and tag. This folding technique is logically implemented as a parity function (logic XOR) over each column, where the number of columns is equal to the folded length (see Figure 3). Upon entry of a history bit, H, instead of recalculating the logic XORs over each column, the new folded GHR, c', can be calculated through the following steps [10]:

1) Circular shift: *For* $n > 0$: $c'_n = c_{n-1}$; $c'_0 = c_{max}$
2) xor entering history bit: $c'_0 = (c'_0 \text{ xor } H)$
3) xor last bit of the geometric series: $c'_{(U\%F)} = (c'_{(U\%F)} \text{ xor } b_{max})$

where the $'$ symbol represents the new/updated value, U = unfolded length, F = folded length, and max represents the MSB ($b_{max} = 17$ and $c_{max} = 4$ in this case).

As shown in Figure 3, for each folded GHR, only two of the folded history bits (c'_3 and c'_0 in this example) need new GHR information to be recalculated (since their matching-colored circles are not identical). The other folded history bits can simply be circular shifted. On any given update, each folded GHR can be created using the old folded value, incoming history bit, and a single bit from the GHR (denoted b_{max}), as opposed to reading all GHR bits and recalculating using large XOR trees. For this reason, all recovery mechanisms considered in this paper implement GHR updates using this optimized GHR folding method to reduce power and area.

As specified in the third Championship Branch Prediction (CBP3) implementation for ISL-TAGE, the bits needed to calculate the folded histories are at history depths 3, 8, 12, 17, 33, 35, 67, 97, 138, 195, 330, 517, 1193, 1741, and 1930. A practical implementation of TAGE should be able to efficiently extract these GHR bits within a single cycle with low power and area overheads. In order to support single-cycle folded GHR recovery on branch mispredictions, we checkpoint each

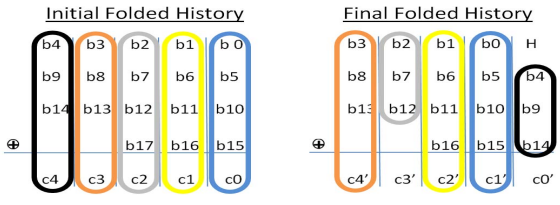

Fig. 3. Example of updating a 5-bit folded GHR for a tagged table with a history length of 18.

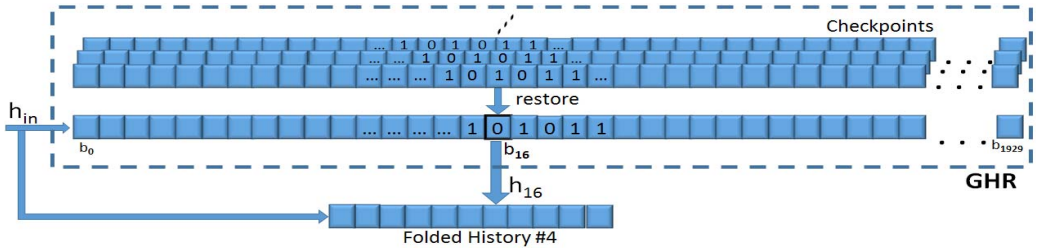

Fig. 4. Fixed-location access: The GHR element, h_n, is always stored at flip-flop (or latch) number b_n. This figure shows traditional checkpointing updating a fourth tagged table folded history.

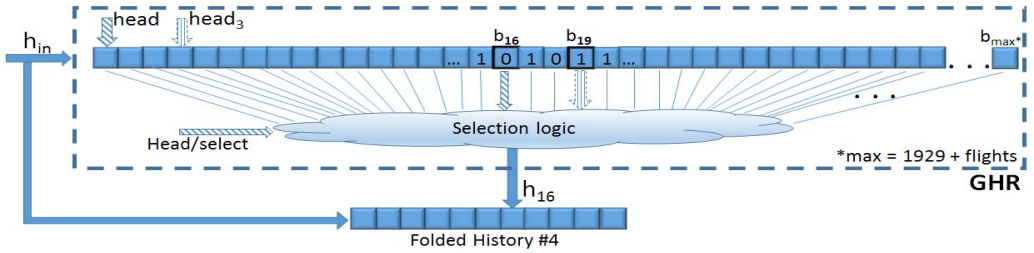

Fig. 5. Variable-location access: The GHR element, h_n, could reside in any of the GHR flip-flops or latches, depending on the head pointer position. This figure shows a circular buffer updating a fourth tagged table folded history. Note the large fan-in to the selection logic.

of the 11-bit average folded GHRs.

GHR Storage: The GHR could be stored either as a collection of flip-flops or in an SRAM structure. Although flip-flops are not as area or energy efficient as SRAM cells, they do not exhibit the read limitations of SRAM blocks. These limitations make an SRAM-based designs impractical for previously proposed checkpointing methods for TAGE. Section V describes how our novel implementation, BADGR, addresses these challenges.

GHR Update: Lastly, the GHR must efficiently reflect changes to the GHR state upon branch predictions at the front-end of the processor's pipeline. This can either be done through the use of shifting bits (fixed-location designs) or updating head-pointers (variable-location designs).

IV. Conventional Approaches

In this section, we explore the trade-offs of traditional checkpointing, backwards shifting, and circular buffer implementations of TAGE's GHR. We classify traditional checkpointing and backwards shifting as *fixed-location* designs, since each GHR history depth bit always resides in the same (fixed) physical location (See Figure 4). Alternatively, the circular buffer is an example of a *variable-location* design, since a given GHR history depth bit resides in multiple (variable) physical locations depending on head pointer location.

A. Traditional Checkpointing

Pros: *GHR Folding.* Since traditional checkpointing is a fixed-location design, GHR folding is highly efficient. The 15 bits read from the GHR always reside in the same location and require no selection logic. Figure 4 shows an example of how fixed-location access methods update the folded GHR for a fourth tagged table (requiring GHR bit 17 to update). These designs guarantee the bit of history depth 17 (h_{16}) to be located at the same flip-flop (b_{16}) at all times.

Cons: *In-flight scalability, GHR Update, and GHR Storage.* Due to TAGE's large GHR, each checkpoint requires thousands of stored bits. The as the number of in-flight branches supported increases, the overhead increases linearly. Due to the fixed-location design, each of the history bits are shifted for every branch entering the pipeline, which results in high dynamic power consumption in GHR updates. Although the checkpoints can be stored in SRAM cells, the GHR itself must be stored in more area and power expensive flip-flops.

B. Backwards Shifting

Pros: *GHR Folding.* Backward shifting is also a fixed-location design, and also guarantees no need for selection logic between the GHR and folding modules.

Neutral: *In-flight scalability.* Backwards shifting eliminates the need for entire GHR checkpoints by shifting n bits in a single-cycle, where n is the depth of the mispredicted in-flight branch. Extra flip-flops are extended to the end of the GHR to support additional in-flight branches. However, as the number of supported in-flight branches increases, the number of inputs required in the selection logic (for multiple position shifts) at each bit position grows proportionally.

Cons: *GHR Update and GHR Storage.* Similar to traditional checkpointing, as each branch enters the pipeline, every GHR bit shifts, incurring high dynamic power consumption on GHR updates. Backwards shifting is also a flip-flop based design and does not utilize dense SRAM cells for GHR storage.

C. Circular Buffer

Pros: *In-flight scalability and GHR Update.* Unlike fixed-location designs, circular buffers have inexpensive GHR up-

dates that require a head pointer update and only one flip-flop to be written each time a branch enters the pipeline. Increasing the number of in-flight branches supported simply requires additional flip-flops at the end of the buffer.

Cons: *GHR Folding and GHR Storage.* In variable-location designs, the location of the 15 history bits needed reside in different flip-flops as the head pointer moves, resulting in expensive selection logic for GHR folding. Figure 5 shows an example of updating the fourth tagged table's folded GHR. If the head points to the first flip-flop, the bit required for updating the folded GHR is b_{16}. In this example, a misprediction from in-flight branch of depth 3 moves the head pointer up 3 positions, making history bit 17 reside in b_{19}. Since the head pointer can be located at any history location, expensive selection logic (an extremely large multiplexer on the order of 2000:1) needs to select the correct bit to update the folded history, which causes high area and power overheads for GHR folding. The cost of this selection logic is further increased since all 15 folded histories are updated in parallel for single-cycle updates. For this reason, an otherwise area-efficient SRAM array cannot be utilized to store the circular buffer, since the SRAM would need 15 independent read ports in addition to one write port needed for GHR and folded GHR single-cycle updates. *As described in the following section, one of the key contributions of this paper is to show how to practically implement such a multi-ported structure.*

V. BADGR Implementation

We showed in the previous section that fixed-location or variable-location access is a critical factor in the trade-offs between scalability, GHR folding, GHR storage, and GHR update. Refer back to Figure 1 for a summary of each method's overhead breakdown. Our novel design, Bank Assignment to Deconflict Geometric Recurrences (BADGR), is a hybrid of fixed and variable-location methods, carefully designed to combine the advantages of both variable-location and fixed-location methods. Namely, BADGR contains a head pointer to incur the low GHR update and scalability costs from the variable-location access model while staying as fixed-location as possible to reduce the number of potential locations of the GHR bits required for GHR folding.

Instead of expensive flip-flops, BADGR uses area-efficient 6-transistor SRAM cells, which allows for smaller GHR storage and GHR update overheads. Similar to the circular buffer method, BADGR contains a GHR head pointer; updating the GHR requires only a head pointer update and single bit GHR write upon entry of a new branch.

As explored in Section III, for each folded GHR update, only one GHR bit needs to be read per tagged table (denoted t). The challenge is to efficiently ensure that all t bits can be read from the SRAM within a single cycle. For 15 tagged tables, the 15 history depth bits needed for GHR folding are 3, 8, 12, 17, 33, 35, 67, 97, 138, 195, 330, 517, 1193, 1741, and 1930. The first way this can be done is by having all required bits reside in the same SRAM row.

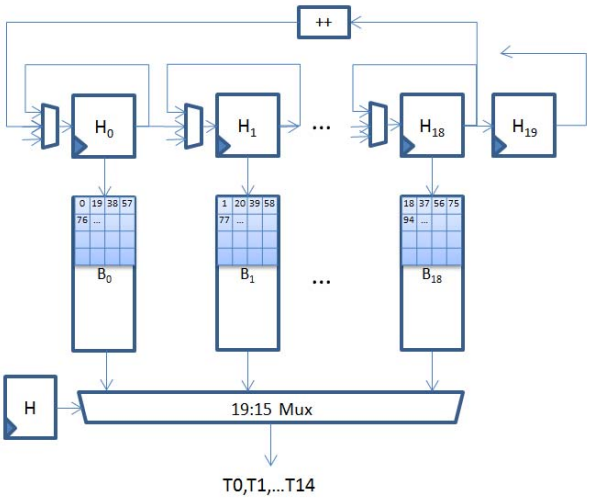

Fig. 6. Implementation of Bank Assignment to Deconflict Geometric Recurrences (BADGR) checkpointing.

One extreme would be to have an SRAM row allocated for each possible head pointer value, with the t bits needed in order for the given head pointer. In this case, head pointer 0 would point to an SRAM row containing bits 3,8,12,..., and head pointer 1 would point to 4,9,13,... and so on. This is highly inefficient, as there would be t copies of every bit. For example, bit 1930 is read when the head pointer resides at location 0,1927,1922,1918,...and so on. This incurs storage overheads and also requires t writes per branch to update all copies of the updated history bit in the SRAM. Another extreme would be to have a single SRAM row, and all history bits are read on a prediction to guarantee the t bits reside in the same row. However, this method still requires the $(1930+n)$:1 selection logic from the circular buffer technique to extract GHR bits for GHR folding.

Between these extremes is a design that attempts to organize each SRAM row with every possible collection of history bits that will be read together for various head pointers. However, since each of the t bits will have its own dependencies on t-1 other head pointer locations, each of those requiring t potentially different bits grouped together, the explosion of bits needed to be contained within the same row converges back to the extreme with a single SRAM row.

The alternative to requiring all of the prediction bits to reside within the same SRAM row is through banking the SRAM. This allows each bank to select its bit from different (or same) rows as other banks. However, the complexity arises in guaranteeing conflict-free bank assignment; each of the history bits needed for any GHR folding update must reside in separate banks to avoid expensive multi-porting.

BADGR's solution to this problem is through cleverly organizing each SRAM bank with the GHR bits with the same (bit % banks) value, where % is the modulo operator (see Figure 6 for an example of bit organization for 19 banks). For all geometric history lengths G_i, as long as each $(G_i\%banks)$

value produces a unique result, we can guarantee that no two GHR geometric length bits will reside within the same bank. Moving the head pointer still maintains conflict-free assignment. This can be proven intuitively through modulo arithmetic: if all $(G_i\%banks)$ values are unique, all $((G_i + headpointer)\%banks)$ will also be unique. Mathematically, conflict-free bank assignment is guaranteed by satisfying the following equation:

$i \in Z: \ i \ \in [0,t],$ (abs [(G_i-$G_{\neq i}$)])% b $\neq 0$

where t=tagged tables, b=banks, and G_i = geometric-length values ($G_0 = 0$, the incoming history bit).

Through an exhaustive search for valid b, we determined that we could achieve this unique-column constraint with 37 banks. However, we attempted to reduce the selection logic even further by examining smaller b values with a only a few G_i conflicts in the equation above. By slightly changing two of the TAGE history lengths (195 becomes 196, 1741 becomes 1742), this constraint is further relaxed to only 19 banks. Due to the minor changes in history lengths, changing these history lengths had negligible effects on TAGE's prediction accuracy (hundredths of a percent).

Since BADGR reads 19 bits from SRAM banks for GHR folding (assuming 15 tagged tables), a 19:15 multiplexer is required to select the 15 useful bits, which is dramatically simpler than the $(1930 + n)$:15 multiplexer from the circular buffer. The current GHR head pointer, H, is used to control location of GHR update, as well as the 19:15 multiplexer, so that the bit from the appropriate column is sent to its corresponding GHR folding mechanism.

In the BADGR design, the bit from each bank is selected using a distributed head pointer, as illustrated in Figure 6. Spacial locality in future accesses allows the distributed head pointers to shift between banks. On wrap-around from the last column (in our case H_{18}), the row number is both shifted to the first column and incremented by 1. For a circular buffer of size 1957 bits, each of the shift registers is only $\log_2(\frac{1957}{19})$, or 7 bits in size. The total circular buffer size should be evenly divisible by the number of banks to guarantee conflict-free bank assignment on wrap-around (we use 1957).

In BADGR, recovering the SRAM pointers from mispredictions is similar to the backward shift method. Pointers need not be explicitly stored, but rather are shifted backwards multiple locations based on the in-flight depth of the branch mispredicted. The RTL implementation can be made either through extending extra pointers for in-flight branches (shown in Figure 6) or through backwards shifting and decrement logic in the case of wrap-around. Due to these bank pointer shift registers and reduced selection logic, we denote the area and power costs for GHR folded updates as medium.

In summary, BADGR combines fixed-location's low GHR folding overhead with the circular buffer's low overheads for GHR updates and in-flight branch scalability, all within in area- and energy-efficient SRAM banks.

VI. Evaluation Methodology

For the performance-based checkpointing simulations, we used the L-TAGE branch predictor on the gem5 simulator [11] with the O3 CPU model run at 1GHz. We ran the first 30 billion instructions for each benchmark within the SPEC CPU2006 Suite with 2048 entries per tagged table. For our prediction performance metric, we used mispredictions per 1000 instructions (MPKI) since MPKI is the standard metric for Championship Branch Predictor competitions, as well as many other branch prediction academic papers.

Since gem5 simulations do not give chip area or power consumption, we designed the checkpointing mechanisms in SystemVerilog. From there, we used Synopsys Design Compiler to calculate area and power consumption. For our first tests we are interested in the effects of the GHR modifications required for supporting in-flight branches. This includes the GHR register, folded GHR copies, and GHR folding selection logic. Prediction selection refers to the selection logic to determine which of the 15 tagged table predictions are selected. Power estimations include the total of static and dynamic power at 1GHz. Our designs were synthesized to a 40nm TSMC standard cell library (tcbn40lpbwp). Implementing BADGR's SRAM structure proved to be difficult using typical memory modeling techniques, since tools such as CACTI do not model SRAM banks that are as small as the 103 bits we need. For this reason we used Synopsys Memory compiler to estimate the area and power for BADGR's SRAM structure, as well as TAGE's tagged tables when testing the overall impact on TAGE. The tagged tables have a single read/write port, as [9] proposes a method to enable such a design per tagged table. The area and power from the memory compiler came from a 65nm transistor library, but was scaled down to 40nm using CACTI [12] scaling factors.

In order to match the TAGE version implemented in [13], unless otherwise stated, our simulations default to a history length of 1930, 15 tagged tables, folded GHR sizes of 10 and 11 bits, and 20 in-flight branches. However, since the number of tagged tables and history length are related, when we vary the number of tagged tables, we also reduce the GHR history length appropriately. We used the CBP2014 framework to measure MPKI for varying numbers of tagged tables [10].

VII. Results

Our first test evaluation looks at the chip-area required to implement each of the four checkpointed GHR designs as shown in Figure 7. As expected, the *fixed-location* designs do not scale as well as the *variable-location* designs. After the costly initial selection logic overhead has been paid, the circular buffer scales better to support more in-flight branches. BADGR's hybrid design has minimal selection logic overhead, while also maintaining in-flight scalability. The SRAM structure of BADGR's GHR also reduces the overall area over the other methods. BADGR demonstrated the most area-efficient design, reducing overall area compared to traditional checkpointing by 85%, and all other designs by $\geq 53\%$ for 20 in-flight branches.

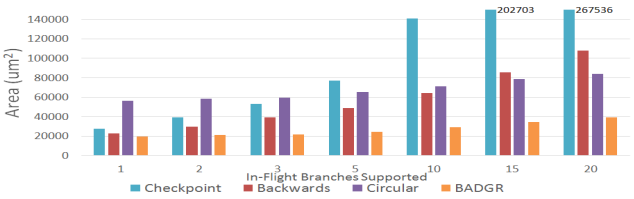

Fig. 7. Chip area of various checkpointing methods.

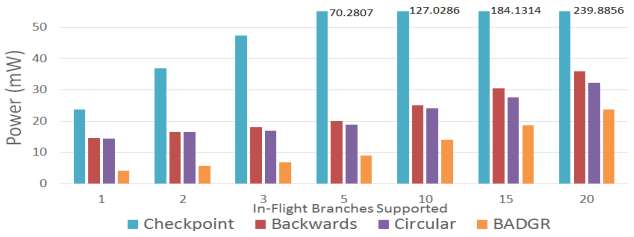

Fig. 8. Power of various checkpointing methods.

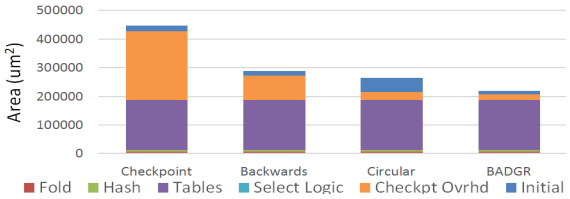

Fig. 9. Area of various TAGE components.

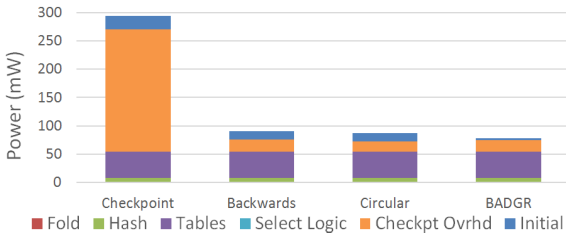

Fig. 10. Power of various TAGE components.

Our next test evaluates the power consumption of the four checkpointing methods as shown in Figure 8. The in-flight branch scalability gap is not as apparent in power consumption as it is for the overall area. This is most likely due to the fact all designs checkpoint the folded GHRs for each tagged table. Since these bits (although much fewer than full GHR checkpoints) must be written for every incoming branch, each design shows the dynamic power increase. However, BADGR and the circular buffer technique still show a slower power increase than backwards shifting and traditional checkpointing as in-flight branches increase. Overall, BADGR outperforms the other checkpointing techniques in power consumption. For 20 in-flight branches, BADGR's modulo-banked design shows a 90% reduction in power consumption over traditional checkpointing, and $\geq$ 27% reduction for all other designs.

To show that optimization of GHR checkpointing plays a critical role in reducing the amount of total power and area of TAGE, we broke down the costs of several functional components within TAGE. *Initial* refers to the cost to checkpoint 1 in-flight branch, and is mostly dominated by the initial GHR storage, GHR update, and folded GHR update costs. *Select logic* routes the prediction of the largest tagged table hit to be the final prediction. *Checkpointing overhead* is considered to be any additional power/area to support 20 in-flight branches.

As shown in Figures 9 and 10, a GHR with traditional checkpointing contributes overhead to over 74% and 58% of the overall TAGE system power consumed and area, respectively, whereas the circular buffer GHR and recovery overhead contributes 44% and 35%. These numbers show that checkpointing plays a large role in the total area and power of TAGE predictors. BADGR shows a 76% and 12% reduction in overall TAGE power consumption over TAGE predictors using traditional checkpointing or a circular buffer (savings are even greater if using less than the modeled 2048 entries per tagged table), and consumes 30% of the entire TAGE system power. Similarly, BADGR shows a 54% and 19% reduction in overall TAGE area over TAGE predictors using traditional checkpointing or a circular buffer. BADGR requires only 14% of the entire TAGE system area. BADGR's substantial reduction in hardware overheads required for mispeculation recovery frees a larger portion of the overall area and power. These savings can either be used to meet area and power budgets or store more entries into tagged tables for even greater prediction accuracy.

Lastly, we analyze how each of the checkpointing methods are impacted by varying the number of tagged tables in Figures 11 and 12. Increasing the number of tagged tables greatly improves TAGE performance for select traces [14], but at the cost of extra storage space, chip area, and power. We confirm this finding by determining that 16 of the 40 CBP4 benchmarks are sensitive to long history (denoted CBP4-SLH). We define a benchmark to be SLH if it shows $\geq$ 40% MPKI improvement from increasing the number of tagged tables from 10 to 15. With only 5 tagged tables (33 history bits), BADGR is not particularly beneficial over the other techniques, as the large benefit of BADGR is gained by eliminating wide muxes, which is not a problem with only 33 history bits. However, Figures 11 and 12 show that BADGR scales best when increasing the number of tagged tables, and is the most efficient solution to checkpointing a TAGE predictor with more than 10 tagged tables.

VIII. Related Work

Prior work has shown that speculative updates to branch history structures are necessary for high branch prediction accuracy, whether for conditional branches or subroutine returns. Early work by Jourdan et al. identified the problem and proposed solutions that are conceptually similar to those proposed here [15]. Subsequent work by Skadron et al. focused on return address stack corruption and methods for combating its destructive effects [16]. A more recent paper revisited this issue and showed that corruption was easy to detect, and

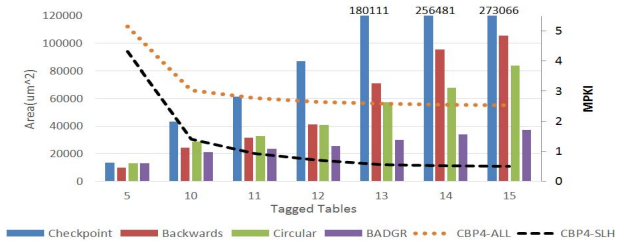

Fig. 11. Area to checkpoint various tagged tables.

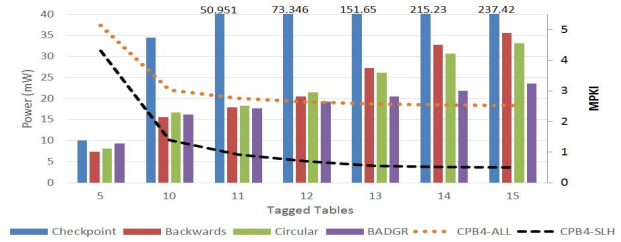

Fig. 12. Power to checkpoint various tagged tables.

proposed using the BTB as a fallback for providing correct target addresses when corruption is detected [17].

The scale of the problem with modern predictors like LTAGE, which require checkpointing of thousands of history bits, motivates a careful re-examination of this problem. In [9], Seznec proposes methods to make speculative updates of the tagged tables. For updating the GHR, he briefly proposes that a circular buffer would be an appropriate way to support GHR speculation, but many of the issues raised in this paper were left unexplored. The prior work also does not describe any RTL-level evaluation of the hardware mechanisms to support the aspects within TAGE such as folded histories, nor estimate the area or power costs it would require to implement.

One alternative for reducing the cost of checkpointing branch predictor state is to selectively create checkpoints only for low-confidence branches. This avoids storage and energy overhead for creating checkpoints that are rarely utilized, since they correspond to branches that are (nearly) always predicted correctly. Jacobsen et al.'s seminal work on branch confidence estimation suggests several hardware mechanisms for solving this problem [18], while later work from Grunwald et al. on speculation gating based on branch confidence could also be applied to this problem [19]. We leave evaluation of selective checkpointing based on branch confidence to future work, but note that since BADGR scales easily to a large number of in-flight branches, the marginal savings from reducing the number of checkpoints will be relatively insignificant, and unlikely to overcome the cost of confidence estimation.

IX. Conclusion

In this work we evaluated the performance effects of various GHR checkpointing methods for TAGE branch predictors that buffer thousands of branch history bits. By incorporating a hybrid of the circular buffer with benefits of fixed-location access flip-flops, we proposed a novel modulo-banked checkpointing mechanism named BADGR that has less power and area inefficiencies than any other proposed checkpointing technique. We found that even with 20 in-flight branches, BADGR can support checkpointing using 90% less power and taking 85% less area than naive checkpointing.

Our analysis of the relative power and area overheads of predictors with very long branch histories raises an important concern regarding the primary cost metric previously used in the prior branch prediction championships. These competitions have focused on the aggregate storage costs of the pattern history tables and other structures that store "useful state" for the predictors. However, our results show that the ancillary structures required for realistic implementation are significant, and can even dominate predictor area and power consumption. Future championship organizers may want to consider incorporating cost metrics that account for these structures in order to fairly assess the cost of competing predictor designs.

References

[1] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, 2007.
[2] D. A. Jiménez, "An optimized scaled neural branch predictor," in *ICCD*, pp. 113–118, IEEE, 2011.
[3] A. Seznec, "Storage free confidence estimation for the tage branch predictor," in *HPCA*, pp. 443–454, IEEE, 2011.
[4] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.
[5] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *ISCA*, 2002.
[6] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *ISCA*, 2004.
[7] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *MICRO*, p. 93, IEEE Computer Society, 2003.
[8] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, "Power issues related to branch prediction," in *HPCA*, pp. 233–244, IEEE, 2002.
[9] A. Seznec, "A new case for the tage branch predictor," in *MICRO*, pp. 117–127, ACM, 2011.
[10] T. J. of Instruction-Level Parallelism, "Championship branch prediction (cbp-4)," 2014. [Online; accessed 15-June-2014].
[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.
[12] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "Cacti 5.0," Tech. Rep. HPL-2007-167, HP Labs, 2007.
[13] T. J. of Instruction-Level Parallelism, "2nd jilp workshop on computer architecture competitions (jwac-2): Championship branch prediction," 2011. [Online; accessed 11-Nov-2015].
[14] D. Gope and M. H. Lipasti, "Bias-free branch predictor," in *MICRO*, pp. 521–532, IEEE, 2014.
[15] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt, "Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution," *Int. J. Parallel Program.*, vol. 25, Oct. 1997.
[16] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," in *MICRO*, pp. 259–271, IEEE Computer Society Press, 1998.
[17] H. Vandierendonck and A. Seznec, "Speculative return address stack management revisited," *ACM Trans. Archit. Code Optim.*, vol. 5, pp. 15:1–15:20, Dec. 2008.
[18] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *MICRO*, MICRO 29, (Washington, DC, USA), pp. 142–152, IEEE Computer Society, 1996.
[19] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence estimation for speculation control," in *ISCA*, ISCA '98, (Washington, DC, USA), pp. 122–131, IEEE Computer Society, 1998.