COBRA: A Framework for Evaluating Compositions of Hardware Branch Predictors

Jerry Zhao, Abraham Gonzalez, Alon Amid, Sagar Karandikar, Krste Asanović
University of California, Berkeley
{jzh, abe.gonzalez, alonamid, skarandikar, krste}@berkeley.edu

***Abstract*—We present COBRA, a framework which enables a realistic hardware-guided methodology for evaluating compositions of hardware branch predictors. COBRA provides a common interface for developing RTL implementations of predictor sub-components, as well as a predictor composer that automatically generates hardware predictor pipelines from sub-components based on a high-level topological model of a desired algorithm. We demonstrate how COBRA aids in the design and evaluation of diverse predictor architectures and how our hardware-centric approach captures concerns in predictor characterization that are not exposed in software-based algorithm development. Using COBRA, we generate three superscalar pipelined branch predictors with diverse architectures, synthesize them to run at 1 GHz on a commercial FinFET process, integrate them with the open-source BOOM out-of-order core, and evaluate their end-to-end performance on workloads over trillions of cycles. The COBRA generator system has been open-sourced as part of the SonicBOOM out-of-order core.**

I. Introduction

Branch prediction accuracy is critical to modern high-performance processors, since any misprediction may cause hundreds of in-flight instructions to be flushed from the pipeline. However, the complexity of modern branch prediction algorithms and implementations presents a challenge to computer architects interested in characterizing the state-of-the-art in this area. State-of-the-art branch predictors are typically comprised of many interacting sub-predictors [22], [41]. For example, a high performance design might contain a next-line predictor, branch-target-buffer, globally indexed counter tables, locally indexed counter tables, and loop predictors [18], [19], [32]. While such algorithms can be easily implemented as software functional models, prior work as shown that software simulators demonstrate substantial modelling error [6], [20]. Software models also cannot describe the physical implementation costs of a particular design point.

On the other hand, building a hardware implementation of such a complex pipeline within a superscalar speculating core is a daunting task. While existing open-source cores provide black-box interfaces for implementing a custom predictor [13], [49], these interfaces provide no tools to help an architect manage superscalar execution, pipelining, and speculation. The designer must consider the prediction pipeline as a monolithic blackbox, instead of a more natural composition of sub-components.

COBRA aims to bridge this gap between the productivity of a software model and the realism of a hardware implementation by providing a framework for generating hardware branch predictors from a library of sub-components according to a high-level topological description of the algorithm. COBRA provides a common RTL interface for implementing a sub-component of a predictor pipeline, and includes a starter library of RTL implementations using this interface. COBRA also provides a branch predictor "composer", which interprets a topological model of a predictor design to generate a complete predictor pipeline from a library of sub-components. This composer also synthesizes various predictor management structures, such as history providers, repair mechanisms, and history files, to manage the state of predictor sub-components. The resulting pipeline is automatically integrated into the

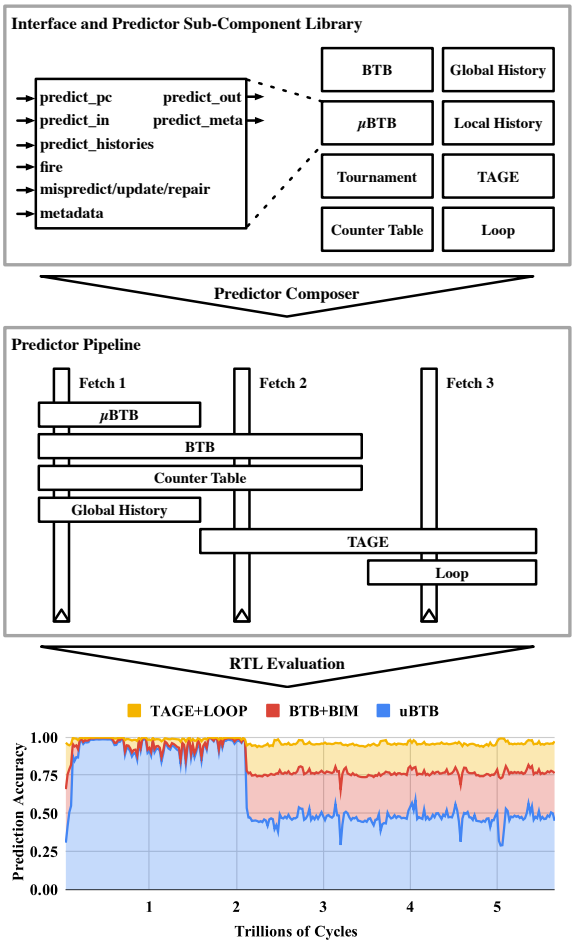

Fig. 1. The COBRA flow. COBRA generates a complete predictor pipeline from a library of sub-components conforming to the COBRA interface, and enables design feedback from multi-trillion cycle simulations and physical design.

frontend of the open-source Berkeley Out-of-Order Machine (BOOM) [13] to enable evaluating the predictor in the context of a complete core. Figure 1 depicts the complete COBRA flow.

To demonstrate how COBRA enables holistic hardware-guided evaluations of branch predictors, we generate three varied predictor designs using the COBRA composer, including a high-performance TAGE design [40]. We evaluate each predictor on the entire SPECint17 suite [9] with reference inputs by running FPGA simulations of a complete out-of-order core with each predictor attached. We demonstrate how evaluating a hardware predictor in the context of a complete core reveals subtle nuances in predictor design that would not be captured in a trace-based software simulation model.

The COBRA branch predictor generator is open-sourced and available as part of the latest version of the BOOM out-of-order core, SonicBOOM [50].

II. Background and Related Work

A. Branch Prediction

Branch prediction is a well-studied area in general-purpose processor design. Over decades of research in this field, several key ideas have proven to be critical concerns when designing a complete predictor within a high-performance core.

Correlating future branch behavior with *branch history* is a thoroughly validated approach for performing branch prediction [35]. The refinement of history-based algorithms has been a large contributor to overall improvements in branch predictor accuracy, as pattern-history-tables [48], GShare [29], GSelect [29], YAGS [16], GEHL [38], PPM [30], TAGE [40], and perceptron [24] all propose various techniques for leveraging branch history.

Combining multiple prediction structures through an *arbitration scheme* into a hybrid predictor has also proven to be successful, as a collection of predictors with affinities for different branch behaviors can be more accurate and efficient than a single generic predictor [22], [26], [29].

Predictor latency and pipelining are important factors to consider when physically implementing a prediction algorithm, as more complex prediction algorithms may not fit in a single cycle at high frequency [21], [23]. Realistic implementations of complex algorithms typically require multiple pipeline stages to calculate indices, read predictor memories, arbitrate a final prediction, and generate a target address [39]. Predictor delay may increase the frequency of frontend bubbles, decreasing overall instruction throughput. Thus, to reduce the frequency of frontend bubbles inserted by a slow, long-latency predictor, modern predictor implementations will typically include faster low-latency predictors to provide a earlier prediction before the backing predictor can respond [4], [42].

Another important concern in predictor design is *superscalar prediction* [12], [17]. Although a simple design might serialize the instruction stream behind branches, we observe that the trend towards wide instruction fetch units points towards the importance of superscalar prediction as well. We measured that serializing the fetch unit behind branch predictions in a 4-wide fetch BOOM core decreased IPC by 15% in the Dhrystone synthetic benchmark [47].

Two other concerns in predictor design include *predictor repair* and *predictor storage*. Predictor repair mechanisms are necessary for maintaining the state of local histories and loop counters, which are often corrupted due to misspeculated updates [44], [46]. Predictor storage efficiency is important because high performance branch predictors might require thousands of entries with counters, tags, and target addresses. Many techniques have been devised for efficiently storing predictor state in single or dual-ported SRAMs [37], [40].

Given the complexity of modern branch predictor pipelines, a framework for accurate evaluations of predictor algorithms should expose these concerns to architects using clearly defined abstractions.

B. Software Branch Predictor Simulators

Software simulators, such as ChampSim [2], CBPSim [1], BPSim [51], SimpleScalar [10], or gem5 [8], are often used in the development of novel branch predictor algorithms. Although these simulators provide architects the freedom to implement flexible software abstractions for predictor composition and design, they also present significant disadvantages [34], [45].

Execution-driven simulators like gem5 and SimpleScalar simulate precise microarchitectural behaviors, but do so at the expense of simulation speed. Trace-based simulators like ChampSim and CBPSim are faster, but cannot model microarchitectural behaviors like speculation and superscalar execution. Both execution and trace-driven simulators have been shown to demonstrate substantial modelling error for branch prediction accuracy [3], even after hand-tuning [15], [20]. When single-digit percentage reductions in mispredictions are commercially valuable, the difficulty of hand-tuning a software simulator to accurately model hardware behaviors limits the usefulness of such simulators towards branch predictor design exploration.

Software simulators also provide limited insight into the cost or complexity of the physical realization of a predictor algorithm. While BPSim can analytically estimate area utilization and energy consumption of the TAGE predictor algorithm, it relies on hand-crafted models that do not generalize to other designs.

By generating realistic hardware predictor RTL instead of relying on software models, COBRA enables fast and comprehensive evaluations of predictor designs through FPGA simulation, as well as accurate physical design feedback from standard EDA tools.

III. The COBRA Predictor Interface

The COBRA interface is designed to support diverse physically realizable implementations of hardware branch predictor sub-components. Section IV discusses how sub-components conforming to the COBRA interface can be composed into a complete predictor pipeline.

A. Pipelined Predictors

In the COBRA interface, prediction begins when the predictor sub-component receives the fetch PC, at cycle 0. However, branch predictor sub-components have diverse latencies. For example, a fast micro BTB might provide a target address on the next cycle after fetch redirection, while a backing TAGE-like predictor might require 3+ cycles to index the counter tables and arbitrate a final prediction.

In the COBRA interface, an implementation of a predictor sub-component may respond at any cycle $p \geq 1$ after predictor query. The implementation of a predictor must only guarantee that for a prediction made at cycle p, either the same prediction or a more powerful prediction is provided for all cycles $d > p$. The requirement is not restrictive, as any prediction made at cycle p is assumed to be valid in future cycles. The internal pipelining of a predictor sub-component is not limited beyond this, providing freedom to the predictor microarchitect.

B. Branch Histories

Since incorporating a history vector is a central technique of many branch prediction algorithms, our interface provides both local and global histories as input vectors `ghist`, `lhist` to a predictor sub-component. Global histories generally need to be read out of some history register, and local histories need to be read from a local history table. The history vectors are provided only at the end of the first cycle to these predictors. Figure 2 shows when a pipelined predictor may access the PC and history inputs, and the stages at which it might respond.

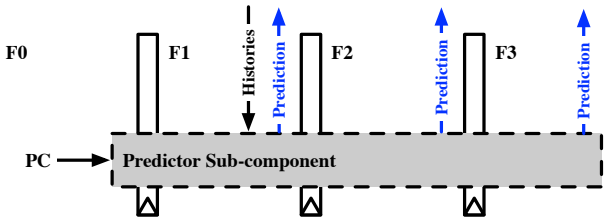

Fig. 2. Pipeline diagram depicting when predictors are queried (Fetch-0), when histories are provided (Fetch-1), and when predictions can be made (Fetch-1,2,3,...).

C. Superscalar Prediction

In the COBRA interface, a predictor sub-component outputs a vector of predictions `predict_out` for a PC that is expected to fetch multiple instructions. This enables the design of superscalar sub-components which can learn behaviors for multiple branches within a fetch-packet. This is especially useful for avoiding aliasing between branches in branch-dense code.

For example, consider two adjacent conditional branches that are frequently in the same fetch packet. A bimodal counter table which reads only one entry per cycle would not be able to learn the target addresses for both branches correctly, as both branches would alias onto the same entry. A superscalar counter table would be able to provide correct target addresses for both branches.

However, more complex predictor sub-components, like perceptron predictors or loop predictors, might only be able to provide a single prediction per cycle. Implementations of these sub-components can provide a single prediction for the entire vector, or learn the index into the fetch-packet at which to provide the prediction.

D. Metadata

The COBRA interface provides an abstract `metadata` field to each predictor sub-component. The implementation of any sub-component specifies the bitlength of the metadata it wants to store, and provides this metadata on cycle p alongside its prediction. The interface guarantees that this metadata will be provided to the predictor sub-component at a future point during update or repair. This field is useful both for reducing the number of read ports in predictor memories, and for supporting repair mechanisms for predictors storing local branch state.

We observe that many predictor sub-components ought to be implemented as area-efficient single or dual ported memories, since predictor accuracy improves substantially with storage budget [31]. To fully utilize fetch throughput at steady-state, a pipelined predictor should be able to both provide a prediction and learn an update every cycle. Thus, at full throughput, a predictor would naïvely require 1 write and 2 reads (1 predict and 1 update) per cycle.

To avoid the second read at update-time, a predictor implementation in the COBRA framework could store the read data from predict-time in the `metadata` field, and recover it automatically from the interface at update-time. COBRA does not restrict what information may be stored in this field, so implementations of sub-components may generally use this field to reduce repeated computation or memory accesses at update-time. For example, a set-associative structure could store the hit-way, while an arbitration scheme can store the ID of the provider sub-component. Given that different types of sub-components will have unique uses for this field, we allow each sub-component to independently specify the bit-length required.

E. Predict, Update, and Repair

We observe that predictor sub-components have diverse requirements on predict, update and repair. The COBRA interface describes a common set of prediction "events" which encompass the set of signals most predictor sub-components might require. The signals are:

- `predict`: Informs the sub-component to begin generating a prediction for a provided PC
- `fire`: Informs the sub-component to speculatively update local state for a prior `predict` PC
- `mispredict`: "Fast" immediate update from a mispredicted branch
- `repair`: Informs the sub-component to repair misspeculated local state for a given `predict` PC
- `update`: "Slow" commit-time update from a committing branch

Although implementations of predictor sub-components may choose to use and ignore arbitrary subsets of these five signals, we target two primary use cases: commit-time-update and speculative-update.

As an example, a predictor which learns correlations with global history might be tolerant to delayed commit-time updates. Such a predictor would not be corrupted due to misspeculation, but would also suffer from the additional delay before the backend commits a branch. An implementation of this design would use the `update` signal provided by the interface to update predictor state when branches commit.

On the other hand, a predictor which learns local behavior might require immediate updates after prediction, for example, to increment a loop counter. Such a predictor would also require immediate repair upon misspeculation, as misspeculation could corrupt the predictor's state with falsely speculated updates. An implementation of this style of predictor would use the `fire`, `mispredict`, and `repair` signals to manage its local state.

The `mispredict`, `repair`, and `update` signals all provide the same fetch PC and histories provided at `predict` time, so that the predictor sub-component can regenerate indices calculated at predict time. In addition, these three events also provide the resolved or misspeculated direction for the relevant branches for a fetch PC. The `metadata` bitvector a predictor generates at predict time is also provided back to it at `mispredict`, `repair`, and `update` time.

F. Predictor Composition

While untagged predictor sub-components might provide a base prediction for all PCs, tagged predictor components have the option to provide no prediction. Additionally, some predictor sub-components may only provide a partial prediction, for instance a BTB which provides the target address for a branch, but not the predicted direction.

To support both use cases, the interface provides an additional input field `predict_in(d)` representing predictions made from other branch predictor sub-components at any cycle d. The implementation of a predictor sub-component may use as input any `predict_in(d)` with $d \leq n$, where n is the desired latency for the output prediction.

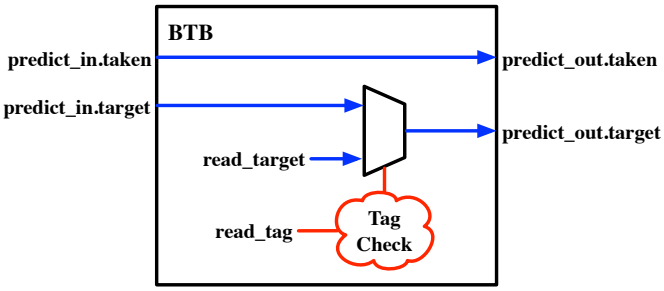

Fig. 3. A BTB which learns only target addresses augments the incoming `predict_in` prediction.

In the event where a predictor component chooses to provide no prediction on cycle n, the implementation should simply "pass-through" the incoming prediction from cycle $d = n$. In the event where a predictor component chooses to provide a partial prediction at cycle n, the implementation should simply override fields of the incoming `predict_in(d)` with $d = n$.

Figure 3 displays how this interface enables decoupling of a BTB from other predictor sub-components. This BTB augments a predicted direction provided by some other predictor sub-component with a predicted target. In the event of a tag miss, the BTB passes through the predicted target from `predict_in`. This enables composition of this BTB with other predictor sub-components.

We additionally observe that arbitration schemes like a tournament predictor choose one of many incoming predictions. Thus, a predictor sub-component may be implemented to require multiple `predict_in` inputs. Such a predictor can arbitrate between multiple `predict_in` inputs to determine the output prediction.

G. Sub-Component Library

We demonstrate the flexibility of the COBRA predictor interface by implementing a library with a subset of commonly used predictor sub-components. These predictor sub-components are defined as synthesizable RTL modules in the Chisel hardware construction language [5]. We implement only a representative subset of sub-components to demonstrate the flexibility of the COBRA interface, as other predictor types, like perceptron [24] or statistical-corrector [40], may be implemented similarly.

1) Counter-tables: The sub-component library includes bimodal counter tables with a parameterized indexing option, so they can be indexed by a global history, local history, PC, or any hashed combination of the above. The `metadata` field of the predictor interface stores counter values to avoid re-reading the tables at update time.

2) BTBs: The sub-component library includes two types of BTBs: a large 2-cycle BTB, and a small 1-cycle microBTB (uBTB). The set-associativity of the BTBs is enabled by the `metadata` field of the interface, as the implementations uses this field to recover the hit way at update time.

3) Tournament selector: The tournament selector contains a 2-bit counter table, indexed by global history to select the winning sub-predictor. The selector uses the `metadata` field to track the predictions made by the sub-predictors to determine an update for the counter table.

4) TAGE: The TAGE sub-component manages updates and predictions from a set of global-history tagged tables according to the algorithm described in [40]. The `metadata` field is used to track the index of the provider and allocator tables.

5) Loop predictor: The loop predictor attempts to correct periodic mispredictions made by a base predictor. Its design is a simpler version of the loop predictor implemented in [41]. Unlike previously discussed sub-components, the loop predictor is updated at query time, and repaired immediately on mispredicts. This sub-component uses the `metadata` field to track the contents of its counter entries such that it can restore those entries during the repair phase.

IV. The COBRA Predictor Composer

COBRA also provides a composer for hardware branch predictors. The COBRA composer is driven by a topological representation of the desired predictor design. Given a topological representation for a design, the composer generates a synthesizable hardware pipeline from available predictor sub-components. The composer also integrates the pipeline with generated management structures for maintaining the state of the entire pipeline.

A. Predictor Topologies

We observe that a complete predictor pipeline can be represented as an ordering of predictor sub-components, where the ordering specifies which predictor sub-component provides the final prediction. For a collection of predictor components with varying latencies, an ordering of the predictor sub-components specifies the hierarchy of predictions used to generate a final prediction from the entire pipeline at each pipeline stage. Specifically, for any latency d, the subset of the predictor topology containing sub-components with latency $n \leq d$ specifies the final prediction made d cycles after query. Given two sub-components p_a, p_b, an ordering "$p_b > p_a$" indicates that sub-component p_b provides the final prediction in any cycle where the final prediction is ambiguous.

For example, consider a predictor pipeline with three sub-components: a single-cycle uBTB (uBTB_1), a two-cycle tagged pattern-history-table (PHT_2), and a two-cycle loop predictor (LOOP_2). We explore how the COBRA topological representation enables expressing two different compositions of these basic components.

$$\text{LOOP}_2 > \text{PHT}_2 > \text{uBTB}_1$$
$$\text{uBTB}_1 > \text{PHT}_2 > \text{LOOP}_2$$

Consider the prediction that the pipelines specified by these topologies will provide 1 cycle after fetch. In both topologies, the subset of the topology with $n \leq 1$ contains only the uBTB, so both pipelines will provide the same predictions 1 cycle after fetch. However, the prediction emitted at cycle 2 differs substantially between either topology.

In the first topology, the PHT is specified to override the prediction formed by the uBTB. Thus, if the PHT predicts not taken, it will override the prediction provided by the uBTB from cycle 1. The behavior is similar for the loop predictor. In the case where the loop predictor catches a loop, it will override the predictions formed by both the PHT and uBTB. In the case where neither the loop predictor nor the PHT form a prediction, the prediction made by the uBTB in cycle 1 is automatically carried over to cycle 2.

In the second topology, the uBTB is specified as the most powerful prediction. Thus, a hit in the uBTB in cycle 1 will cause the uBTB prediction to be the final prediction in both cycles 1 and 2, regardless of whether or not the PHT or loop predictors hit. In the case where the uBTB provides no prediction, the generated pipeline will use the PHT prediction when both the PHT and loop predictors hit.

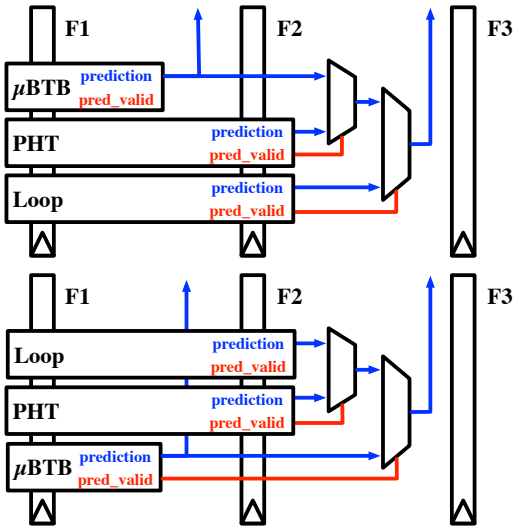

Fig. 4. Pipeline diagrams of two possible predictor topologies. The prediction provided at Fetch-1 is naturally pipelined into future pipeline stages. The uBTB provides the Fetch-1 prediction in both topologies.

Figure 4 depicts the pipelines generated by either topology. This topological representation makes no assumptions about what the "best" topology for a given set of sub-components is. Instead, freedom is presented to the designer to specify an arbitrary topology based on the sub-components and the desired pipeline.

1) Predictor Arbitration: For predictor arbitration schemes which learn to choose a final prediction from two or more sub-components, we observe that a simple ordering cannot be used to describe the desired pipeline. Consider a tournament predictor scheme, which contains a 2-cycle untagged local-history-indexed table (LHT_2), 2-cycle untagged global-history-indexed table (GHT_2), and a 3-cycle arbitration scheme for selecting a final prediction, (TOURNEY_3). Such a topology can be expressed in the form:

$$\text{TOURNEY}_3 > [\text{GHT}_2, \text{LHT}_2]$$

This form of expressing topologies is very flexible at permitting generation of diverse branch predictor pipelines. Consider the case where the designer wants to incorporate a loop predictor. Three reasonable topologies can be expressed to describe integration approaches for a 2 or 3-cycle loop predictor.

$$\text{TOURNEY}_3 > [(\text{LOOP}_2 > \text{GHT}_2), \text{LHT}_2]$$
$$\text{TOURNEY}_3 > [\text{GHT}_2, (\text{LOOP}_2 > \text{LHT}_2)]$$
$$\text{LOOP}_3 > \text{TOURNEY}_3 > [\text{GHT}_2, \text{LHT}_2]$$

In the first two modified topologies, the loop predictor augments either the global or local-indexed tables, while in the final topology, the loop predictor corrects the final tournament prediction. This flexibility demonstrates the power of the topological representation, as it can capture a wide variety of predictor designs.

```
// Construct the predictor sub-components
val loop    = Module(new LoopPred(nEntries=16))
val gbim    = Module(new HBIM(useGlobal=true))
val lbim    = Module(new HBIM(useLocal=true))
val tourney = Module(new Tourney)

// Express the edges of the topology
tourney.io.predict_in(0) := gbim.io.predict_out
tourney.io.predict_in(1) := lbim.io.predict_out
loop.io.predict_in(0) := tourney.io.predict_out

// Specify the source for the final prediction
final_prediction := loop.io.predict_out
```

Fig. 5. Example code demonstrating construction of construction of a predictor pipeline from a desired topology and available sub-components.

B. Predictor Pipeline Generator

To specify a desired topology to the COBRA composer, the designer specifies the sub-component nodes in the design, the topology connecting the nodes, and the node providing the final prediction. For example, consider the topology presented previously:

$$\text{LOOP}_3 > \text{TOURNEY}_3 > [\text{GHT}_2, \text{LHT}_2]$$

Figure 5 demonstrates how a user would drive the composer to produce the pipeline for this topology. Notice how the code can be easily modified to elaborate any of the three pipelines described in Section IV-A1.

The composer additionally generates control-flow-redirection logic to provide natural overriding of earlier predictions from low-latency sub-components with later predictions made by long-latency sub-components [21]. This is similar to the technique used in the Alpha21264 [26].

1) Predictor Management Structures: A substantial part of the complexity of a predictor pipeline lies in the predictor management structures, which are responsible for maintaining the state of the predictor through speculative execution and updates. Thus the COBRA predictor composer also generates these structures, and automatically connects them to the generated pipeline.

The generated history file is a circular buffer which tracks the state of predictions in the pipeline. When the processor pipeline backend resolves a branch, the target address and direction are updated into the history file. Entries are dequeued from the history file in program order as the core commits branches. When entries are dequeued, the resolved direction and PC are sent to all the predictor sub-components to perform predictor updates.

The `metadata` field discussed in Section III-D is stored in the history file. The `metadata`, along with the global and local histories, are provided to the predictors at update, mispredict, and repair time, such that the predictor sub-components can recover entry indices and read data.

2) Predict, Update, and Repair: A state machine sits alongside the history file to generate update or repair signals for the branch predictor sub-components. In steady state, this state machine generates commit-time update signals as it dequeues the history file. After a mispredict, the state machine performs a "forwards-walk" through the history file to generate repair signals, which restore the state of local-history and loop predictors. This is similar to the forwards-walk scheme proposed in [46].

3) History providers: The COBRA predictor composer additionally generates history providers. The current implementation supports a global history and a PC-indexed local history.

The global history provider speculatively updates itself based on the predicted direction of in-flight branches. Our initial simple implementation corrects mispredictions by storing snapshots of the global history register in the history files. The local history provider is also speculatively updated by predicted directions of in-flight branches. On mispredict, the local history provider is repaired by the repair mechanism described above.

COBRA's initial implementations of these history providers are basic, but sufficient for demonstrating the flexibility of the framework. A more efficient global-history register could be implemented using pointers into a circular buffer. Other variants of history information, like path histories [33], can also be implemented as new history providers.

C. Composer Implementation

A COBRA-generated predictor can be integrated into a host processor as a drop-in replacement for the host processor's existing branch prediction and fetch redirection logic. Since COBRA generates a complete predictor pipeline, including the predictor management structures, integration should require minimal changes to existing RTL in the host core. We choose the Berkeley Out-of-Order Machine (BOOM) as the host processor to demonstrate integration of a COBRA-generated branch predictor. BOOM is a configurable open-source superscalar out-of-order RISC-V core [13]. Figure 6 depicts COBRA integration with BOOM's instruction fetch unit.

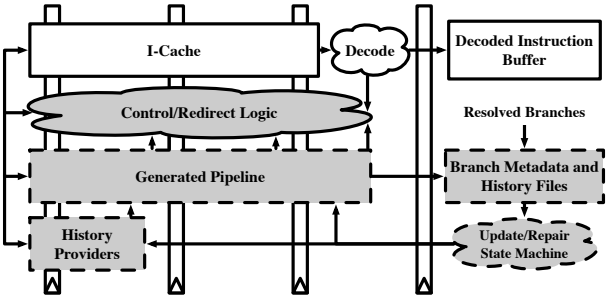

Fig. 6. BOOM instruction fetch unit modified to accept a predictor pipeline generated by COBRA. Gray components indicate new COBRA-generated components.

The only prediction sub-component from the original BOOM core which was preserved was the return-address-stack (RAS). As the design of return-address-stacks is relatively less complex than the design of other predictor structures in branch prediction, we do not incorporate it in the initial implementation of COBRA.

V. Evaluation

We demonstrate COBRA's utility for generating and evaluating high-performance branch predictor architectures by exploring three diverse predictor designs. We compare the end-to-end performance of these designs to the end-to-end performance of existing open-source and commercial hardware predictor implementations.

A. Design Exploration

We demonstrate COBRA's versatility by generating three diverse predictor designs out of the COBRA sub-component library, summarized in Table I, Figure 7, and the topological representation below.

$$\begin{aligned} \text{LOOP}_3 > \text{TAGE}_3 > \text{BTB}_2 > \text{BIM}_2 > \text{uBTB}_1 \\ \text{GTAG}_3 > \text{BTB}_2 > \text{BIM}_2 \\ \text{TOURNEY}_3 > [\text{GBIM}_2 > \text{BTB}_2, \text{LBIM}_2] \end{aligned}$$

The first "TAGE-L" topology describes a TAGE (TAGE_3) predictor with a loop corrector (LOOP_3). The backing predictor is a PC-indexed bimodal counter table (BIM_2). This design is vaguely similar to the state-of-the-art branch predictor algorithm TAGE-SC-L [41], only with no statistical corrector, and a simpler loop predictor. The second "B2" topology describes a similar predictor to the one implemented in the original BOOM core. This predictor pairs a single partially tagged table of history-indexed counters (GTAG_3) with a PC-indexed backing counter table (BIM_2). The third "Tourney" describes a tournament predictor, with a globally-indexed tournament selector (TOURNEY_3) choosing between counters provided by untagged global and local history-indexed tables ($\text{GBIM}_2, \text{LBIM}_2$). This predictor is similar to the ones in the Alpha21264 [26] and riscyOOO [49] cores.

COBRA enables the reuse of the counter tables, BTB, and predictor management structures across all three topologies, greatly reducing the design effort of each design. Furthermore, individual predictor sub-components were designed and validated independently, before evaluation of the complete predictor pipelines.

For evaluation, all three predictor pipelines were attached to a four-wide BOOM core, as configured according to Table II. The resulting core was then synthesized at 1GHz through Cadence Genus [11] on a commercial FinFET process. Synchronous memories in the core, including most branch predictor memories, were mapped to available SRAMs in that technology. Critical paths in the resulting design are not affected by the generated branch-predictor (they are in the issue-units of BOOM).

Figure 8 reports the total area breakdown of the branch predictor for each of the three designs, including the cost of predictor management structures. Expensive tagged predictor sub-components, such as the TAGE tables and the BTB, are relatively costly compared to untagged structures. Furthermore, predictor management structures incur non-trivial cost, as the local history provider generates a large PC-indexed table

TABLE I
Parameters of evaluated COBRA-designed predictors.

Name	Description	Storage
Tourney	32-bit global, 256x32-bit local histories 2K-entry BTB w. 16K-entry 2-bit BHT 1K tournament counters	6.8 KB
B2	16-bit global history 2K partially tagged + 16K untagged counters 2K-entry BTB	6.5 KB
TAGE-L	64-bit global history 7 TAGE tables 2K-entry BTB w. 32-entry uBTB 256-entry loop predictor	28 KB

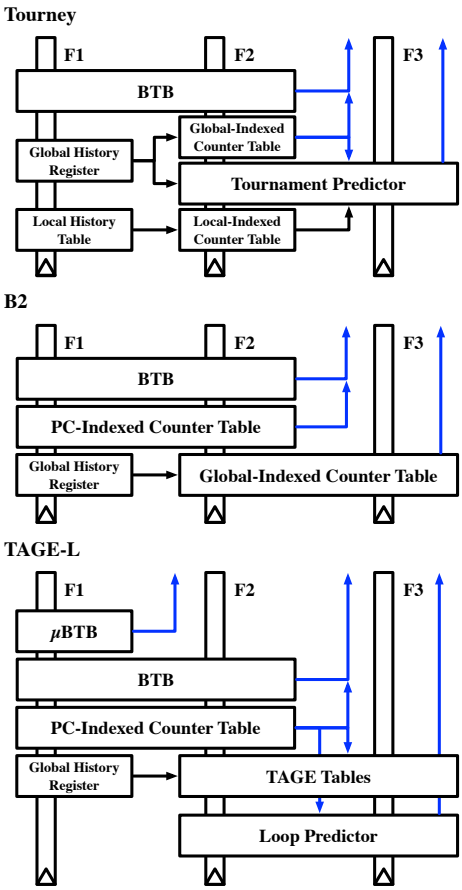

Fig. 7. Pipeline diagrams of the COBRA-generated predictors.

TABLE II
BOOM parameters in evaluation of COBRA predictors.

Frontend	16-byte wide fetch 4-wide decode/rename/commit 32-bit and 16-bit RVC instructions
Execute	128-entry ROB 8 pipelines (4 ALU, 2 MEM, 2 FP) 3x 32-entry IQs (INT, MEM, FP)
Load-Store Unit	32-entry LDQ, 32-entry STQ 2 LD or 1 ST per cycle
TLBs	32-entry L1 DTLB 32-entry L1 ITLB 1024-entry L2 TLB
L1 Caches	8-way 32 KB ICache and DCache next-line prefetcher
L2 Cache	8-way 512 KB
L3 Cache	4 MB FASED [7] LLC model
Memory	16 GB FASED [7] DDR3 timing model

TABLE III
EVALUATED SYSTEMS FOR SPECINT17 PERFORMANCE COMPARISON.

Core	Intel Skylake	AWS Graviton	BOOM		
Branch predictor	Undisclosed		Tournament	B2	TAGE-L
L1 Cache Sizes (I/D)	64/64 KB	48/32 KB	32/32 KB		
L2/L3 Cache Size	1 MB/24 MB	2 MB/0 MB	512 KB/4 MB		
Compiler/OS	gcc/Ubuntu 18.04 Server		gcc/Buildroot Linux		
Platform	AWS EC2 bare-metal		FireSim FPGA-accelerated cycle-exact simulation		

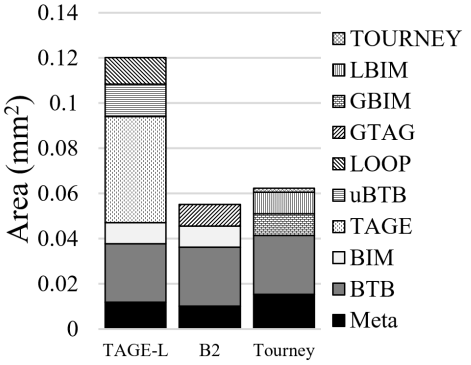

Fig. 8. Area utilization of our three predictor pipelines, broken down across predictor sub-components. Meta denotes predictor management structures like history files and history providers.

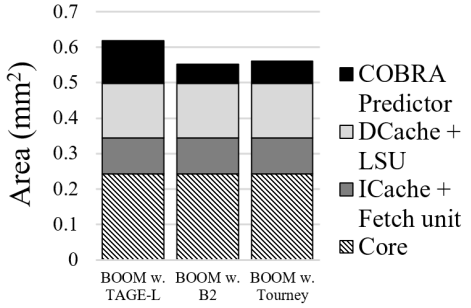

Fig. 9. Area utilization of 4-wide fetch BOOM cores with each of the 3 evaluated predictors.

of histories.

Figure 9 reports total area breakdown of the core, to highlight the relative cost of the predictors compared to other core components. The total area of even a large predictor design is only a small portion of the area of a large superscalar out-of-order core.

B. Performance Evaluation

We additionally evaluate the branch prediction accuracy and IPC of BOOM with the COBRA-generated branch predictors, and compare to commercial server-class cores. The modified BOOM cores were synthesized on AWS F1 FPGAs using FireSim [25]. The FireSim simulations ran at 30 MHz on the FPGAs, and modeled the system running at 3.2 GHz. We run the SPECint17 speed benchmarks with reference inputs as single-threaded workloads. The SPECint suite was compiled using `gcc`, with `-O3` optimizations. Branch prediction accuracy and IPC were measured with the out-of-band profiling tools in FireSim.

To demonstrate the value of the holistic evaluation COBRA enables, we additionally compare our COBRA-augmented BOOM core to Intel Skylake and AWS Graviton cores, running as `c5n.metal` and `a1.metal` instances on AWS EC2. For these cores, SPECint was compiled using `gcc` with `-O3`. Branch prediction accuracy and IPC were measured with the Linux `perf` profiling tool. Table III displays the comparison of all profiled systems.

Figure 10 displays the SPECint17 evaluation results. The B2 and Tournament predictors are less accurate, but also require far less area than the TAGE-L predictor. The Tournament predictor notably suffers from aliasing issues on several workloads, caused by the lack of a tagged predictor sub-component in this design.

VI. DISCUSSION AND FUTURE WORK

Evaluating our predictor in the context of a complete core revealed issues in predictor design that would not have been exposed in a software model of the algorithm. We discuss representative examples of the phenomena we observed, demonstrating the value of end-to-end branch-predictor evaluation.

A. Physical Design

Pipelining a predictor is sometimes necessary for avoiding critical paths or congestion. However, increasing prediction latency will also increase the number of bubbles inserted into the pipeline on a redirection. Unlike a software model, our hardware-centric predictor design framework enables evaluating the performance implications of predictor pipelining and delay.

Our original implementation of the TAGE-L predictor arbitrated a final decision in 2 cycles, instead of 3. While this design could redirect the PC faster, it introduced a critical path, as table read, tag check, and prediction arbitration were all performed in one cycle. To resolve this issue, we inserted an additional pipeline stage into the TAGE sub-component, such that the prediction latency became 3 cycles.

Since the COBRA interface for predictor sub-components supports varied latencies, the TAGE sub-component could be modified in isolation from other sub-components. Thus, modifying TAGE required no modifications to the composer, or to the topology for a complete TAGE-based predictor pipeline.

Delaying the TAGE response had no impact on overall prediction accuracy, and a minimal ($\approx 1\%$) degradation of IPC. This is partially because not all branches in a SPEC workload are hard-to-predict, and benefit from a TAGE-based prediction [28]. Furthermore, the BOOM core can fetch more

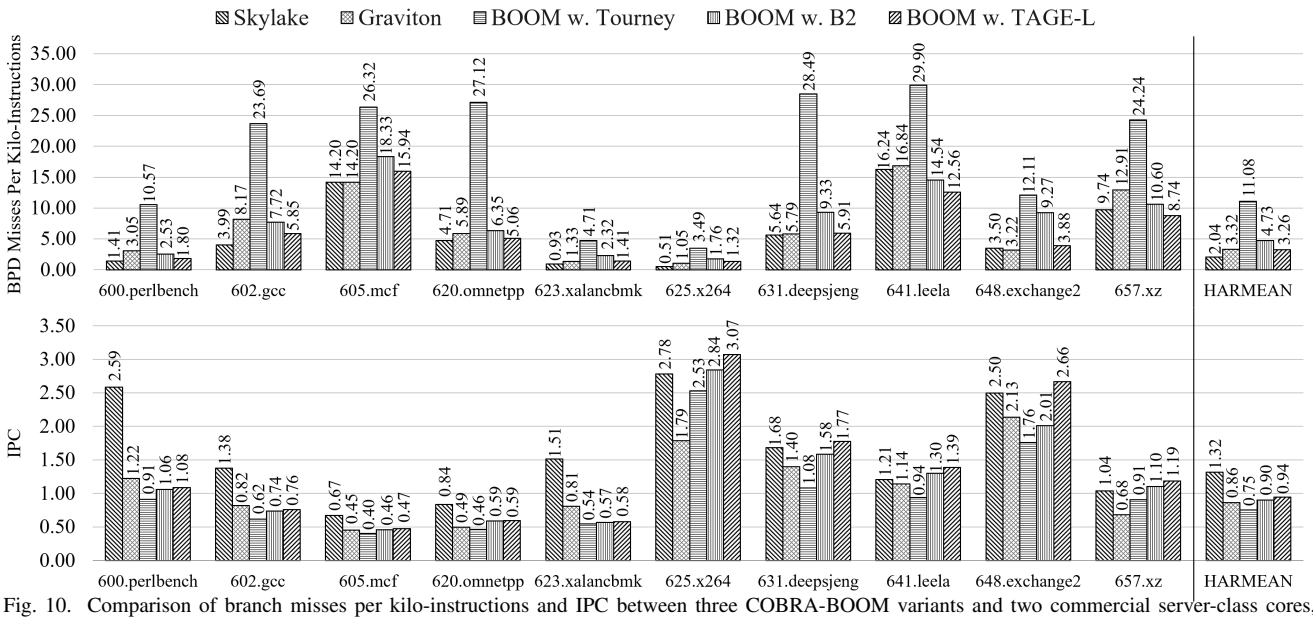

Fig. 10. Comparison of branch misses per kilo-instructions and IPC between three COBRA-BOOM variants and two commercial server-class cores, on 10 SPECint benchmarks. HARMEAN is harmonic mean across all benchmarks. Comparison against Skylake and Graviton is approximate due to different ISAs.

instructions than it can decode in a cycle, so backpressure from decode will frequently hide temporary stalls in instruction fetch.

In the future, we plan on more aggressively tuning the predictor sub-components according to additional feedback from physical design. Predictor energy consumption is expected to be an important concern, as the energy cost of continuously reading predictor SRAMs is significant [36].

B. Speculative Execution

Since the frontend of a core is inherently speculative, predictor structures must be tolerant to misspeculation. While trace-based simulators do not model speculative execution, a predictor generated by the COBRA framework can be fully evaluated within the context of a speculating core. We examine a specific instance of how speculation significantly affects branch prediction accuracy.

One structure for which misspeculation is especially dangerous is the global history register, as misspeculated updates to the global history register potentially corrupt many future predictions. In our original design for the global history provider, misspeculated global history updates were repaired, but predictions formed from a misspeculated history were not replayed. We explored an alternate design, in which repairing global history forced a replay of instruction fetch with the corrected history. Although replaying fetch inserts bubbles into the fetch pipeline, correcting invalid predictions made with a misspeculated global history has positive effects on overall prediction accuracy.

We found that repairing the global history improved mean IPC by 15%, and reduced the branch mispredict rate by 25% across all SPECint benchmarks. However, on select short loop-based benchmarks, the delay from history repair decreased overall IPC. On Dhrystone, for example, additional frontend bubbles inserted by the history repair process decreased total IPC by 3%. A future, more optimal design could dynamically learn when to aggressively speculate past malformed predictions, and when to insert bubbles to recover a correct history.

C. Core Optimizations

Microarchitectural optimizations in the backend of a core can have significant implications on branch prediction accuracy. Since our predictor composer generates complete pipelines suitable for integration into high-performance cores, we can holistically evaluate the interactions between a predictor and the backend.

To demonstrate, we modified the backend of BOOM to decode short-forwards-branches (or “hammock” branches) into set-flag and conditional-execute micro-ops [27]. Specifically, execution of the short-forwards-branch would set a predicate bit instead of redirecting control flow, and the instructions in the shadow of the branch would read the predicate bit when issued to determine whether to execute, or perform no operation. This is similar to the mechanism implemented in the IBM Power8 microarchitecture [43].

We found that enabling the short-forwards-branch optimization improved the accuracy of all three branch predictor designs due to two effects. First, branches that could be dynamically decoded as short-forwards-branches would no longer cause mispredictions, as the correct behavior for these branches would always be ”not-taken”. Second, branch predictor sub-components no longer needed to devote entries to learning the direction of these short-forwards-branches, freeing up resources for learning other hard-to-predict branches.

Overall, this optimization substantially improved prediction accuracy on the CoreMark EEMBC benchmark [14]. Without the optimization, a 4-wide fetch BOOM core with the COBRA TAGE-L predictor achieves only 4.9 CoreMarks/MHz, and 97% branch prediction accuracy. However, enabling the optimization improves performance to 6.1 CoreMarks/MHz,

with 99.1% branch prediction accuracy. Combined with future backend optimizations like custom instructions or macro-op fusion, COBRA provides an important path for researchers interested in end-to-end core evaluation and optimization.

VII. Conclusion

We present COBRA, a framework for modeling and evaluating compositions of hardware branch-predictor pipelines. The COBRA interface supports reusable implementations of branch predictor sub-components, and the COBRA composer enables hardware-based evaluation of arbitrary branch-predictor topologies. To our knowledge, the COBRA-augmented BOOM core is the fastest open-source RISC-V core by IPC [50]. COBRA has been open-sourced as part of the latest version of BOOM to provide a productive, realistic platform for advancing the state-of-the-art in branch predictor design and high-performance cores.

Acknowledgements

The information, data, or work presented herein was funded in part by the NSF CCRI ENS Chipyard Award #2016662, as well as by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was partially funded by ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] "Branch prediction championship 2016 kit," https://www.jilp.org/cbp2016/framework.html.

[2] "Champsim," https://github.com/ChampSim/ChampSim.

[3] A. Akram and L. Sawalha, "A comparison of x86 computer architecture simulators," 2016.

[4] A. ARM, "Cortex®-a72 mpcore processor technical reference manual (revision r0p2)."

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[6] R. Bhargava, L. K. John, and F. Matus, "Accurately modeling speculative instruction fetching in trace-driven simulation," in *1999 IEEE International Performance, Computing and Communications Conference (Cat. No. 99CH36305)*. IEEE, 1999, pp. 65–71.

[7] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanović, "FASED: FPGA-accelerated simulation and evaluation of DRAM," in *FPGA'19*.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[9] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[10] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH computer architecture news*, vol. 25, no. 3, pp. 13–25, 1997.

[11] "Rtl design, genus style," https://www.cadence.com/en_US/home/multimedia.html/content/dam/cadence-www/global/en_US/videos/tools/digital_design_signoff/rtl_design_genus_style, Cadence, 2015.

[12] P. Caprioli and S. Chaudhry, "Multiple branch predictions," Dec. 1 2005, uS Patent App. 11/068,626.

[13] C. Celio, "A highly productive implementation of an out-of-order processor generator," Ph.D. dissertation, PhD thesis, EECS Department, University of California, Berkeley, 2018.

[14] E. M. B. Consortium *et al.*, "Coremark benchmark," 2013.

[15] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proceedings of the 28th annual international symposium on Computer architecture*, 2001, pp. 266–277.

[16] A. N. Eden and T. Mudge, "The yags branch prediction scheme," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 69–77.

[17] P. G. Emma, J. W. Knight, J. H. Pomerene, and T. R. Puzak, "Simultaneous prediction of multiple branches for superscalar processing," Jul. 18 1995, uS Patent 5,434,985.

[18] A. Fog, "The microarchitecture of intel, amd and via cpus," *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2011.

[19] A. Frumusanu, "Arm's cortex-a76 cpu unveiled: Taking aim at the top for 7nm," May 2018. [Online]. Available: https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse

[20] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 13–22.

[21] D. A. Jiménez, "Reconsidering complex branch predictors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 2003, pp. 43–52.

[22] D. A. Jiménez, "Multiperspective perceptron predictor," *Championship Branch Prediction (CBP-5)*, 2016.

[23] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 67–76.

[24] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[25] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.

[26] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE micro*, vol. 19, no. 2, pp. 24–36, 1999.

[27] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE, 1998, pp. 278–285.

[28] C.-K. Lin and S. J. Tarsa, "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *arXiv preprint arXiv:1906.08170*, 2019.

[29] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[30] P. Michaud, "A ppm-like, tag-based branch predictor," *Journal of Instruction Level Parallelism*, vol. 7, no. 1, pp. 1–10, 2005.

[31] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 292–303.

[32] M. Milenkovic, A. Milenkovic, and J. Kulick, "Demystifying intel branch predictors," in *Workshop on Duplicating, Deconstructing and Debunking*, 2002.

[33] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE, 1995, pp. 15–23.

[34] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, no. 6, pp. 4–12, 2015.

[35] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, 1992, pp. 76–84.

[36] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, "Power issues related to branch prediction," in *Proceedings Eighth International*

Symposium on High Performance Computer Architecture. IEEE, 2002, pp. 233–244.

[37] D. J. Schlais and M. H. Lipasti, "Badgr: A practical ghr implementation for tage branch predictors," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 536–543.

[38] A. Seznec, "The o-gehl branch predictor," *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[39] 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, pp. 1–6, 2007.

[40] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 117–127.

[41] A. Seznec, "Tage-sc-l branch predictors again," 2016.

[42] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 295–306, 2002.

[43] B. Sinharoy, J. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira *et al.*, "Ibm power8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2–1, 2015.

[44] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 259–271.

[45] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in computer architecture evaluation," *Computer*, vol. 36, no. 8, pp. 30–36, 2003.

[46] N. Soundararajan, S. Gupta, R. Natarajan, J. Stark, R. Pal, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Towards the adoption of local branch predictors in modern out-of-order superscalar processors," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 519–530.

[47] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[48] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th annual international symposium on Microarchitecture*, 1991, pp. 51–61.

[49] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind, "Composable building blocks to open up processor design," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 68–81.

[50] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[51] C. Zhou, L. Huang, and Q. Dou, "Bpsim: An integrated missrate, area, and power simulator for branch predictor," in *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 2017, pp. 1–4.