Exploring Cache Policies on FPGA-Accelerated Simulations: Tradeoffs Between Usability and Simulation Speed

Soraya Mobaraki, Thierry Gil, Lionel Torres, David Novo

To cite this version:

Soraya Mobaraki, Thierry Gil, Lionel Torres, David Novo. Exploring Cache Policies on FPGA-Accelerated Simulations: Tradeoffs Between Usability and Simulation Speed. RSP 2025 - 36th International Workshop on Rapid System Prototyping, Sep 2025, Taipei, Taiwan. In press, ⟨10.1145/3768617.3770732⟩. ⟨lirmm-05406182v2⟩

HAL Id: lirmm-05406182

https://hal-lirmm.ccsd.cnrs.fr/lirmm-05406182v2

Submitted on 23 Dec 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Exploring Cache Policies on FPGA-Accelerated Simulations: Tradeoffs Between Usability and Simulation Speed

Soraya Mobaraki Thierry Gil Lionel Torres David Novo

LIRMM, Univ. Montpellier, CNRS

Abstract

FPGA-accelerated simulation offers a promising path toward achieving fast and cycle-accurate architectural evaluations. However, while platforms like FireSim significantly improve simulation speed compared to traditional software-based simulators, they still pose usability challenges for rapid microarchitectural prototyping due to the complexity of register-transfer level (RTL) development. In this work, we present a modular and latency-insensitive interface that simplifies the integration of cache replacement policies into FireSim's L2 cache. Our approach supports multiple implementation strategies, including host-based software, RTL, and high-level synthesis (HLS), thereby enabling multiple tradeoffs between simulation speed, hardware resource usage, and development effort. We use the advanced Hawkeye replacement policy as a case study to demonstrate the versatility of our approach and evaluate tradeoffs among the different implementation strategies. Our results show that, with the proposed latency-insensitive L2 interface, the HLS strategy strikes a favorable balance: It incurs only a 1.57× simulation slowdown compared to baseline FireSim, while avoiding the need for RTL expertise and significantly reducing design effort. Latency-insensitive interfaces therefore make it practical and efficient for non-hardware experts to evaluate advanced cache replacement policies on FPGA-accelerated simulators.

CCS Concepts

• **Computer systems organization → Heterogeneous (hybrid) systems**; **Embedded systems**; • **Hardware → Reconfigurable logic and FPGAs**.

Keywords

Computer architecture, FPGA-accelerated simulation, cache replacement, high-level synthesis

ACM Reference Format:
Soraya Mobaraki, Thierry Gil, Lionel Torres, and David Novo. 2025. Exploring Cache Policies on FPGA-Accelerated Simulations: Tradeoffs Between Usability and Simulation Speed . In *36th International Workshop on Rapid System Prototyping (RSP '25), September 28-October 3 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3768617.3770732

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s).

RSP '25, Taipei, Taiwan

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-2224-0/2025/09
https://doi.org/10.1145/3768617.3770732

1 Introduction

Simulation is an essential tool for computer architecture research, allowing evaluation of design alternatives before committing to costly hardware fabrication. Traditional software-based simulators like gem5 [6] and ChampSim [7] offer flexibility and ease of use, but often rely on sampling techniques to manage the prohibitive runtime of full-system evaluations. FPGA-accelerated simulators, such as FireSim [13], address this challenge by providing fast and cycle-accurate simulation of realistic hardware systems, making them particularly attractive for studying system-level behavior and running full workloads.

Despite these advantages, FPGA-accelerated simulators present usability barriers. Integrating custom microarchitectural components, such as new cache replacement policies, typically requires deep RTL expertise and time-consuming development effort. In contrast, software simulators provide high-level abstractions that simplify rapid architectural experimentation. This accessibility gap limits the broader adoption of FPGA-accelerated simulation.

To address this challenge, we propose a modular and latency-insensitive interface that enables plug-and-play integration of selected key functions. We demonstrate this approach by integrating custom cache replacement policies into FireSim's L2 cache. Our design decouples the cache control logic from the replacement policy and supports a range of implementation strategies, including RTL, HLS, and host-based software execution. This interface provides a uniform integration point that abstracts away timing details while preserving cycle-accurate simulation.

To validate our approach, we implement an advanced cache replacement policy, Hawkeye [11], using each supported integration strategy. We evaluate simulation speed, hardware resource usage, and development effort. The results show clear tradeoffs between integration strategies. The HLS integration strategy stands out by offering most of the simulation speedup while preserving a software-like development flow.

In summary, this work makes the following contributions:

- We introduce modular and latency-insensitive interfaces for key architectural functions as a practical way to improve usability in FPGA-accelerated simulations.
- We design and implement a modular and latency-insensitive interface to integrate cache replacement policies into FireSim.
- We propose and implement three distinct integration strategies: RTL, HLS, and CPU-based software, enabling flexible cache replacement evaluation. We open-source our interface and policy implementations [1] to inspire future work.
- We conduct a detailed experimental study using the Hawkeye policy, highlighting tradeoffs between simulation speed, hardware resource utilization, and development effort.

2 Background and motivation

FPGAs are widely used in architecture research for pre-silicon evaluation [8, 13, 18], supporting two main approaches: prototyping and simulation. Both rely on FPGA execution but differ in purpose, abstraction level, and flexibility.

FPGA prototyping involves synthesizing the full RTL design and deploying it directly on an FPGA. It has long been used to prototype ASICs, allowing hardware validation and software development to begin months before silicon is available [8, 15, 17]. However, FPGA prototyping is not well-suited for predicting system-level performance due to timing mismatches between slow FPGA RTL (on the order of 100MHz) and real-speed components like DRAM.

Instead, FPGA-accelerated simulation adds infrastructure to enable runtime control, instrumentation, and components at higher-than-RTL abstraction levels [2, 13]. This enables faster iteration and accurate performance estimation, making it better suited for microarchitectural research. Nevertheless, modifying internal logic—such as cache replacement policies—still demands substantial hardware expertise, a limitation this work aims to overcome.

A core concept in FPGA-accelerated simulation is the host-target separation [13, 18]. The *target* refers to the hardware design under simulation—processor cores, caches, memory—written in a combination of RTL and higher abstraction simulation models. The *host* refers to the physical FPGA and CPU running the simulation, often augmented with tools for instrumentation, control, and memory emulation. This separation enables cycle-accurate yet extensible simulations, combining precision with flexibility.

Motivation example. In set-associative caches, the replacement policy selects a cache line (i.e., victim way) to evict when a cache set is full. For example, the Least-Recently-Used (LRU) policy evicts the line accessed furthest in the past. Many policies have been proposed to improve cache efficiency. ChampSim, a trace-based software simulator commonly used to evaluate new policies, exposes two hooks for custom policies:

- `GetVictimInSet(set, current_set, PC, paddr, type)`, where `set` denotes the cache set being accessed, `current_set` is a pointer to the cache lines in that set (enabling access to their metadata), `PC` is the program counter of the instruction causing the cache access, `paddr` is the physical address accessed, and `type` indicates the memory access type (e.g., load, store, prefetch).
- `UpdateReplacementState(set, way, paddr, PC, type, hit, victim_addr)`, where `way` refers to the selected cache line, `hit` indicates whether the access resulted in a hit, and `victim_addr` corresponds to the address of an evicted line (if any).

This clean abstraction significantly simplifies the development and evaluation of new cache replacement policies. In this work, we investigate the feasibility and implications of supporting a similarly flexible replacement policy interface within an FPGA-accelerated architecture simulation framework. To this end, Figure 1 highlights the two main challenges we address: (1) Identifying and isolating the relevant functionality in the original RTL architecture to enable targeted exploration of cache replacement policies (Section 3); and (2) designing a generic interface that allows flexible integration of new cache replacement implementations while preserving accurate timing behavior (Section 4).

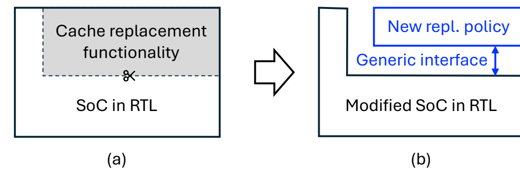


Figure 1: The original RTL (a) and modified RTL (b), which improves usability for non-hardware experts.

3 Modularizing cache replacement logic in RTL

In this section, we examine the inclusive L2 cache generated by Chipyard [2], an open-source hardware design framework for RISC-V SoCs. After outlining its functionality, we propose a modular partitioning to isolate components relevant to the replacement policy.

3.1 L2 overview

Figure 2 shows the inclusive L2 cache in our target SoC, connected to L1 caches via the TileLink SystemBus. The L2 performs several functions: (1) arbitrating L1 requests, (2) serving hits from its memory banks, (3) forwarding misses and handling refills, (4) selecting eviction victims, (5) invalidating L1 lines to maintain inclusiveness, (6) tracking sharers/coherence via a directory, and (7) handling coherence traffic in multi-core systems. To support these functions, it includes five key structures: a requests buffer, a scheduler for arbitration, MSHRs to track outstanding misses, a directory for metadata and coherence, and a banked memory for data storage.

A typical L2 access proceeds as follows. The scheduler (1) performs a tag lookup in the directory. On a hit, it (2) retrieves the data from the banks, (3) updates sharers, and (4) returns the data to L1. On a miss, the L2 (2) checks or allocates an MSHR, (3) selects a victim and gets its metadata, (4) sends invalidations, (5) fetches the line from memory, (6) updates memory and directory, and (7) returns data to L1. Thus, the directory plays a central role: it detects hits/misses, enforces inclusion, tracks sharers, and manages metadata for coherence and replacement. As such, in the next subsection we target the directory for modularizing eviction logic in L2.

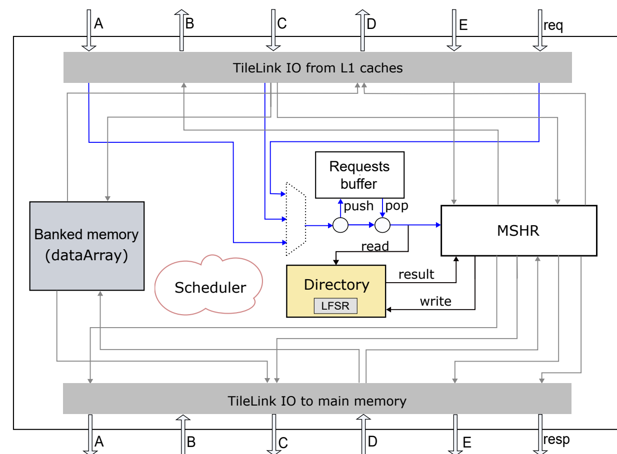


Figure 2: Key internal components of the Chipyard L2 cache.

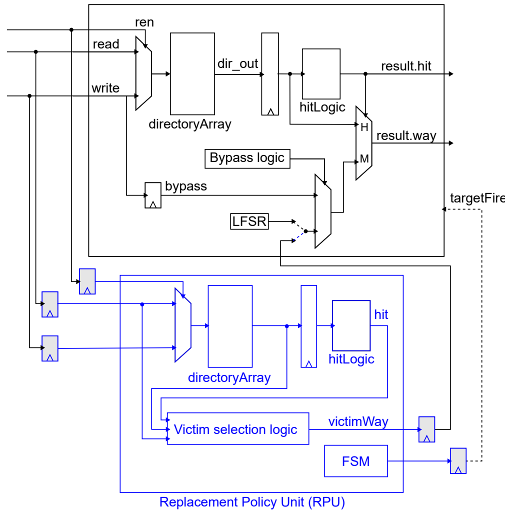
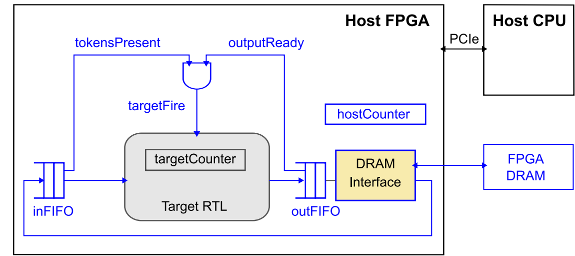


Figure 3: Block diagram of the original directory (black) and proposed extension (blue), which externalizes victim selection logic to enable modular replacement policy integration.



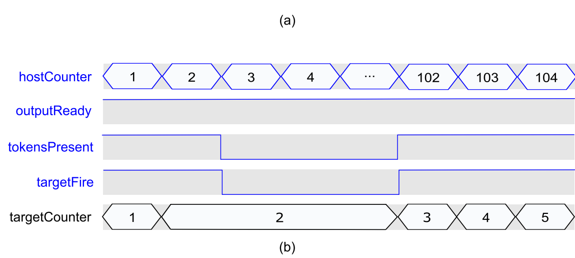


Figure 4: Latency-insensitive design in FireSim: (a) Host-target decoupling architecture, and (b) timing diagram showing architectural time advancement.

3.2 Isolating cache replacement functionality

To support modular integration of custom cache replacement policies, we isolate the portion of the L2 directory logic responsible for victim selection. The baseline design (black logic in Figure 3) provides a simplified view of the original directory logic, focusing on the tag comparison and replacement logic. It uses a Linear Feedback Shift Register (LFSR) to implement random victim selection. The directory logic is tightly coupled to the directoryArray, which stores internal metadata key to victim selection (step 3 of the cache-miss process described in the previous subsection).

To decouple the victim-selection logic from the rest of the directory and enable plug-and-play flexibility, we modify the RTL as in Figure 3. The replacement decision mechanism moves to an external *replacement policy unit* (RPU). This unit receives the same inputs as the directory and produces the victim way. To do so, the RPU replicates the necessary directory functions—e.g., hit detection and metadata access—to drive the *victim selection logic* block.

Through a ChampSim-inspired interface (Section 2), the RPU supports various replacement policies. It takes the inputs `set`, `current_set`, `way`, `paddr`, `type`, `PC`, and `hit`. The `set` and `tag` are already Directory inputs, and the `paddr` can be reconstructed from them. However, to include the `PC` of the cache-access instructions, we modified several Chipyard modules [3] to propagate the PC from the CPU to the L2. We also encode the `type` implicitly in the `PC` (i.e., setting the PC to an out-of-range sentinel value to indicate a write-back access). Finally, to provide `current_set` and `hit` inputs, we replicate part of the directory logic within the RPU.

The proposed RPU shows that key functions like cache replacement can be modularized in FPGA-accelerated simulators: users implement new policies by changing only the victim-selection logic. This boosts usability by hiding the L2 and system-level complexity. Beyond supplying the right inputs, correctness depends on timing. Next, we show how `targetFire` enables variable-latency RPU responses while preserving system behavior.

4 Latency-insensitive interface

To support variable-latency policies in cycle-accurate simulation, the RPU must preserve architectural timing (e.g., the L2 expects the victim way to be provided two cycles after the request arrives) regardless of internal delays. We leverage FireSim's host–target decoupling to make the L2–RPU interface latency-insensitive.

4.1 Host-target decoupling in FireSim

We base our simulation approach on FireSim [13], which employs a host-target decoupling paradigm [16] as discussed in Section 2. To implement decoupled execution, FireSim automatically transforms the target RTL code (i.e., system under simulation). The state updates of the target RTL are gated by a global `targetFire` signal, and inputs and outputs are connected via FIFOs. This formulation decouples target time from host time: unlike in FPGA prototypes, simulation advances only when all conditions for the target RTL to fire are satisfied. This allows the simulator to remain deterministic and independent of host-side latency (e.g., DRAM).

Figure 4 illustrates how the automatically transformed target RTL experiences a 1-cycle memory latency, despite accessing data that reside in the host DRAM memory. Figure 4b shows the corresponding time diagram for some key signals. The `hostCounter` increments on every host clock cycle, capturing the continuous advancement of host-side FPGA logic outside the target RTL. In contrast, `targetCounter`, which represents time from the target system's perspective, advances only when `targetFire` is high, indicating that the conditions to advance target execution are met.

These conditions are encoded by the signals `tokensPresent` and `outputReady`. The FireSim runtime automatically inserts FIFO-based interfaces that monitor input/output readiness. When both signals are asserted, the composite signal `targetFire` goes high, indicating that the target RTL is safe to advance. As shown in the illustrative waveform, although the host FPGA runs through 104 clock cycles, the target RTL part experiences only 5 cycles.

4.2 The L2-cache interface

To support variable-latency implementations of L2 cache replacement policies, we implement the RPU in the host part of the FPGA, similar to the DRAM interface block shown in Figure 4. We control the `targetFire` signal to pause the target simulation when necessary.

FireSim automatically inserts I/O FIFOs (shown as gray registers in Figure 3) to decouple the target and host sides of the FPGA. These FIFOs introduce a one-cycle delay on both input and output paths of the RPU: directory inputs take one cycle to reach the RPU, and RPU outputs take an additional cycle to return to the directory. To avoid observable side effects when the RPU generates the victim way, the original directory must exhibit a minimum latency of two cycles between input and output. This requirement can be satisfied by enabling an optional pipeline register within the directory—an existing configuration parameter in the Chipyard L2.

The resulting timing behavior is illustrated in Figure 5. When a directory read request occurs, the target RTL receives a one-cycle `ren` pulse. This pulse reaches the RPU one cycle later, which in turn immediately deasserts the `targetFire` signal to stall the target RTL. However, the gating takes effect one cycle later. The RPU can then take an arbitrary number of cycles to compute the victim way. Once the computation is complete, the `fire` signal is reasserted and the selected victim way is sent back to the target RTL. As expected by the original directory design, the target RTL receives the victim way from the RPU exactly two cycles after the initial `ren` pulse.

Additionally, integrating the RPU required restricting the original directory functionality. In the baseline implementation, the directory supports back-to-back read requests. However, to simplify the new latency-insensitive interface, we chose to enforce a one-cycle gap between consecutive read requests. This setup allows the RPU to handle only one request at a time by stalling the target RTL upon receiving a read enable signal (`ren`), resolving the victim, and returning the victim way before resuming the target RTL simulation, ensuring that the directory can output the victim way within the required two-cycle latency. In our experiments, the added overhead is negligible, with a maximum of 0.5% cycle overhead.

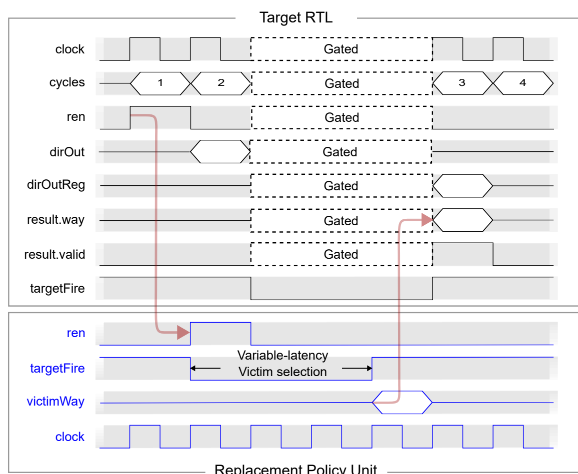


Figure 5: Timing diagram of our latency-insensitive policy.

4.3 Replacement policy unit implementations

Figure 4 illustrates the target-host partitioning of the FPGA and the accompanying CPU. The RPU is implemented on the host side, with most of its functionality residing in the FPGA. Notably, our RPU supports two distinct design variants for implementing the victim selection logic: (1) a hardRPU variant, where the victim selection logic is fully specified in RTL using a hardware description language or HLS, and (2) a softRPU variant, which implements the victim selection logic as a CPU-side driver written in `C++`. Communication between the FPGA and CPU takes place over a 32-bit AXI4-Lite bus, with the CPU accessing memory-mapped FPGA registers via standard `read`/`write` operations.

5 Case study: Advanced cache replacement with Hawkeye

In this section, we use Hawkeye [11] as a representative modern cache replacement policy. We first describe its core algorithm, then explore various strategies for integrating it into FireSim.

5.1 The Hawkeye cache replacement policy

Hawkeye is a practical cache replacement policy that leverages Belady's optimal algorithm to make near-optimal eviction decisions. Belady's MIN policy [4] minimizes misses by evicting the block reused farthest in the future, but it is impractical because it requires perfect future knowledge. Hawkeye addresses this limitation by using past behavior to predict future reuse, assuming that past eviction decisions under Belady's algorithm can guide future ones.

Figure 6 shows the two main components of Hawkeye: *OPTgen* and *PC-based predictor*. OPTgen is a hardware component that reconstructs Belady's optimal decisions for a subset of cache sets via simulation. OPTgen maintains two key structures: occupancy vectors and the Cache history sampler. The occupancy vector tracks whether a given cache line would be present under Belady's replacement at each point in time, while the Cache history sampler saves past memory accesses. These reconstructions are used to train a PC-based predictor, which classifies load instructions as either cache-friendly (likely to benefit from caching) or cache-averse (better to bypass or evict). When a line must be evicted, Hawkeye prioritizes cache-averse lines for replacement. If no such line is present, it falls back on the LRU cache-friendly line.

This predictive approach allows Hawkeye to approach Belady-like behavior using only backward-looking information and without explicitly predicting reuse distances or access patterns. The use of set sampling, which involves monitoring only a small, representative subset of cache sets, significantly reduces hardware overhead. According to the original paper, Hawkeye requires approximately 28KB of metadata and achieves a 17% reduction in LLC miss rate over LRU on memory-intensive workloads from SPEC CPU2006.

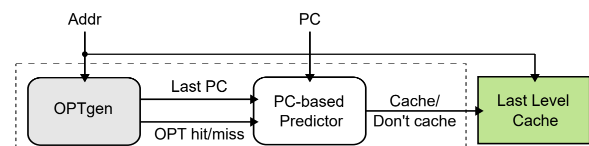


Figure 6: Block diagram of the Hawkeye algorithm.

5.2 SoftRPU: A software implementation

The SoftRPU variant implements the Hawkeye victim selection logic as a host-based software module, running on the CPU (see Figure 4). This approach offers high usability by enabling the direct reuse of the original C++ implementation of Hawkeye from the 2nd Cache Replacement Championship [9], with minimal modification.

On the FPGA side, the victim selection logic remains simplified: it communicates with the host CPU process through a memory-mapped AXI4-Lite register interface over PCI Express Gen3. For each L2 cache access, the inputs—including the address, program counter, hit or miss status, and set index—are written to registers. A polling driver running on the host CPU reads these registers, invokes the appropriate Hawkeye functions (i.e., the `GetVictimInSet` and `UpdateReplacementState` functions described in Section 2), and writes the resulting victim way back to an FPGA register. We use polling rather than interrupts for simplicity and faster synchronization.

This variant is highly user-friendly and even allows direct reuse of the original Hawkeye implementation from ChampSim. However, as we will show in Section 6, it introduces substantial simulation overhead, primarily due to the communication latency between the CPU and the FPGA: each L2 access triggering a cache replacement causes the simulation to pause until the CPU responds (see Section 4.2).

5.3 HardRPU-RTL: An RTL implementation

For the hardRPU-RTL variant, we manually design a fully synthesizable RTL version of Hawkeye's victim selection logic using the Chisel hardware description language (HDL). Figure 7 shows a block diagram illustrating the main internal components of the `GetVictimInSet` and `UpdateReplacementState` modules.

The `GetVictimInSet` module handles eviction decisions by consulting `RRIP` (Re-Reference Interval Prediction) counters [12], then checks if the set is sampled. If so, it retrieves the PC of the last evicted line from the `PC history table`. This PC is forwarded to the predictor for negative training, enabling the system to adaptively learn cache-averse patterns. As in the original Hawkeye policy, only 32 out of 1024 L2 sets are monitored.

The `UpdateReplacementState` module includes three submodules: the `Cache history sampler`, the `Occupancy vector store`, and the `PC-based predictor`. The Cache history sampler is used to reconstruct the history of past events, and it is accessed using the `sample set` and `sample tag` derived from the physical address. Each sample set comprises 8 ways, each storing a valid bit, tag, LRU metadata, the program counter corresponding to the last access to that line (`Last PC`), and a timestamp of the previous access (`Prev time`). On a hit, the previous PC and associated timestamp are retrieved. On a miss, the least recently used entry is evicted, and the new metadata is inserted accordingly. The `Occupancy vector store` uses `Prev time` and `Cur time` (i.e., current time) to compute liveness intervals, which indicate how long a cache line remains 'alive' or present in the cache. `OPTgen` determines whether the access would have been a hit under Belady's MIN policy by checking if the liveness interval exceeds the L2 cache capacity. This result trains the `PC-based predictor`, which maintains a table of saturating counters indexed by hashed PCs. If the access is classified as

Figure 7: RTL-level block diagram of the Hawkeye cache replacement policy implemented in Chisel.

cache-friendly, the predictor is trained positively by incrementing the corresponding counter. If the access is classified as cache-averse, the predictor is trained negatively by decrementing the counter. These updates guide the future classification of the predictor and influence the RRIP adjustments for subsequent eviction decisions.

Manually translating Hawkeye into RTL hardware is a tedious task that requires the careful design of a microarchitecture to balance latency and resource utilization. Several submodules, such as the `Cache history sampler`, `Occupancy vector store`, `PC-based predictor`, `RRIP` counters, and `PC history table`, benefit from low-latency memory structures. While Chisel's `Vec` abstraction offers ease of use, it synthesizes to flip-flops, which are costly in terms of area. To address this, we adopt Chisel's `SyncReadMem` for scalable memory mapping. However, this introduces extra access latency and necessitates explicit control over scheduling and data movement. Accordingly, we implement a finite state machine (FSM) that orchestrates prediction, sampling, and state updates across multiple cycles. Particular attention is also paid to efficient initialization and to minimizing delays along critical paths.

5.4 HardRPU-HLS: An HLS implementation

For the hardRPU-HLS variant, we develop a C-based version tailored for hardware synthesis based on the original C++ implementation of Hawkeye. To make the code more HLS-friendly, we replace dynamic containers (e.g., `std::map`) with fixed-size arrays, adopt the `ap_uint<N>` data type to control bitwidth and reduce resource usage, and insert HLS pragmas to enable loop unrolling and other optimizations. Then, using the AMD/Xilinx HLS tool, we automatically generate the corresponding RTL description from this C-code.

The resulting Verilog module operates on the same victim selection logic interface described in Section 4.2. It receives the address, program counter, set index, way number, and hit/miss status, and returns the selected victim way. Two control signals manage execution: `ap_start` triggers computation, while `ap_done` signals completion. This control mechanism enables an easy integration of this HLS module with the rest of the RPU. Finally, to integrate the Verilog module into FireSim, we define a Chisel`BlackBox`, exposing its interface with a matching `IO bundle`.

Compared to fully handwritten RTL, this HLS-based approach is more accessible to non-RTL experts, requiring only familiarity with HLS concepts.

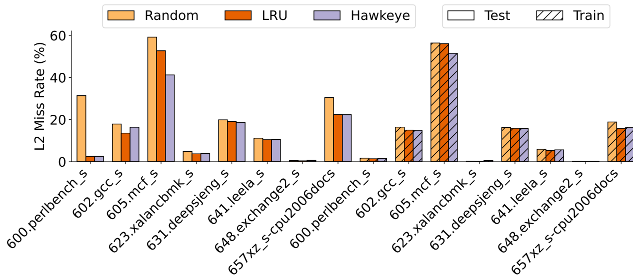


Figure 8: L2 miss rate.

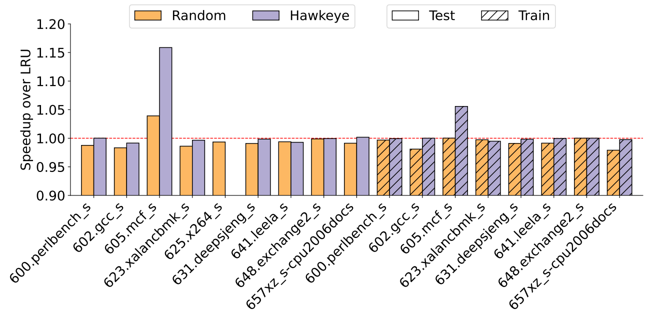


Figure 9: Speedup over LRU.

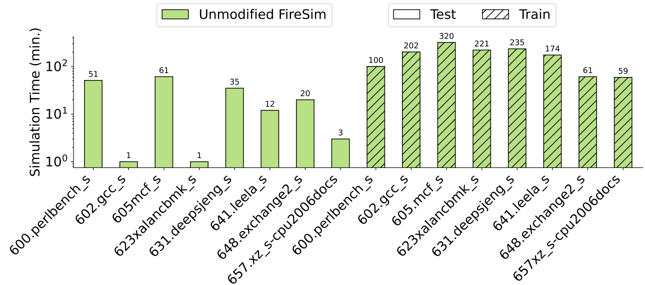


Figure 10: Reference simulation time of baseline FireSim.

6 Results

This section presents the evaluation of our RPU modular integration interface using three implementation variants introduced earlier: SoftRPU, HardRPU using RTL, and HardRPU using HLS. Using Hawkeye as a representative cache replacement policy, we assess each variant in terms of L2 cache miss rate, simulation speed, FPGA resource utilization, and development effort.

Experimental setup. We run experiments on FireSim v1.17.1, deployed on a server with an Intel Xeon Silver 4112 CPU @ 2.60GHz and an AMD Alveo U200 FPGA. The target system features a single in-order Rocket core with private L1 instruction/data caches and a shared, inclusive 512KB 8-way set-associative L2 cache.

We compare three L2 replacement policies: (1) baseline Random; (2) LRU implemented as HardRPU-RTL; and (3) Hawkeye implemented as SoftRPU (Hawkeye-CPU), HardRPU-RTL (Hawkeye-RTL), and HardRPU-HLS (Hawkeye-HLS). Policies are evaluated on memory-intensive SPEC CPU2017 speed workloads [14], using both *train* and *test* datasets; Linux boot time is excluded.

L2 miss rate and speedup. Figure 8 reports L2 miss rates under Random, LRU, and Hawkeye. On average, Hawkeye reduces the miss rate by approximately 7% on the test dataset and 3% on the train dataset compared to LRU. The corresponding speedups are shown in Figure 9, averaging 5% for the test dataset and 1% for the train dataset. For reference, in ChampSim, Hawkeye achieved a 17% LLC miss rate reduction and an 8.4% speedup over LRU [9]. While some benchmarks (e.g., `mcf`) benefit significantly from Hawkeye, our improvements are more modest than in the original paper—likely due to differences in core architecture, cache hierarchy, and workload characteristics. A deeper analysis is left for future work, as this paper focuses on simulation speed and usability.

Simulation speed. Simulation speed is a key advantage of FPGA-accelerated simulation. Figure 10 shows the simulation time needed to run the evaluated workloads end-to-end in the target compute system using baseline FireSim. While typical software-based simulators such as gem5 or ChampSim often rely on sampling methodologies (e.g., SimPoint) to feasibly simulate the train datasets of SPEC CPU2017, FireSim enables full end-to-end execution of these workloads within a few hours. In our setup, the baseline FPGA-accelerated simulation runs the target system at a frequency close to 60MHz.

Table 1 reports the average slowdown of the evaluated implementation variants relative to the baseline FireSim simulation. The HardRPU implementation of LRU results in a 4% slowdown due to a fixed 3-cycle stall of the target on every L2 access and requires RTL expertise (i.e., low usability). The Hawkeye-RTL sees a 15% slowdown, resulting from variable stall cycles depending on the internal state, up to 12 cycles, also offering low usability. Hawkeye-HLS, which offers moderate usability, stalls for 46 cycles per L2 access and results in a 1.57× slowdown. Hawkeye-CPU incurs nearly 20× slowdown due to FPGA–CPU communication overheads but remains the most user-friendly variant.

FPGA resource utilization. Table 2 compares the resource usage of the RTL and HLS variants of the Replacement Policy Unit (RPU) running Hawkeye on the AMD Alveo U200 FPGA (Virtex UltraScale XCU200). Both implementations use only a small fraction of available logic resources.

While the RTL implementation consumes more LUTs and FFs due to explicit control logic and distributed structures, the HLS version significantly increases BRAM usage. This is a result of the HLS tool's automatic inference of large memory blocks for array-based structures, particularly in the Occupancy Vector Store and PC predictor. Despite this tradeoff, the HLS implementation still uses less than 1% of the available LUTs and 6% of the BRAM, indicating that HLS-based integration is viable from a resource standpoint.

Table 1: Average slowdown relative to baseline FireSim

Simulation variant	User abstraction	Avg. slowdown
LRU-RTL	RTL	1.04×
Hawkeye-RTL	RTL	1.15×
Hawkeye-HLS	HLS-friendly C	1.57×
Hawkeye-CPU	C++	19.61×

Table 2: Resource utilization of Hawkeye in AMD Alveo U200

Resource	RTL	HLS	Available
LUT	5,161	1,864	1,182,240
LUTRAM	3,792	90	591,840
FF	2,017	1,049	2,364,480
BRAM	1	131	2,160

Tradeoff analysis. The three implementation variants differed notably in terms of development complexity and required expertise. The SoftRPU approach requires minimal development effort, as it reuses existing software code, but suffers from significant simulation slowdown due to communication overhead between the FPGA and the host CPU. The RTL-based HardRPU achieves the best performance with only minor slowdown, but requires substantial design expertise and development time. The HLS-based HardRPU offers a favorable tradeoff, achieving much of the simulation speed benefit and modest FPGA usage while requiring significantly less development effort and hardware expertise than handwritten RTL.

Our results suggest that HLS-based designs, when supported by appropriate interface abstractions, such as the proposed RPU, enable flexible ttradeoffs between performance and development effort, thereby making design-space exploration more accessible to non-RTL experts in FPGA-accelerated simulators.

7 Related work

Several FPGA-accelerated simulation platforms have been developed to overcome the speed limitations of traditional software-based simulators. Early systems such as RAMP [18] and ProtoFlex [8] demonstrated the feasibility of scalable, cycle-accurate simulation using FPGA hardware. More recently, FireSim [13] has become a widely adopted platform, achieving near-hardware simulation speeds while maintaining architectural fidelity. Building on FireSim, FASED [5] introduced flexible, FPGA-accelerated models for detailed DRAM system simulation, demonstrating how architectural timing models can be efficiently integrated into a fast FPGA-based simulation framework.

However, prior work has largely prioritized simulation speed and functional accuracy over flexibility and usability. As we have motivated in this paper, integrating or modifying microarchitectural components, such as cache replacement policies, remains challenging and typically demands deep RTL expertise. In contrast to software-based simulators like ChampSim [10], which offer convenient interfaces for targeted evaluations, FPGA-based simulation platforms have largely lacked similar modularity and ease of policy integration. To the best of our knowledge, our work is the first to enable modular integration of cache replacement policies into an FPGA-accelerated simulation platform, providing a latency-insensitive interface that supports RTL, HLS, and CPU-based software implementations.

8 Conclusion

This work explores usability ttradeoffs in FPGA-accelerated simulation through a cache replacement policy case study. We propose our RPU as a modular, latency-insensitive interface for integrating replacement policies into FireSim's L2 cache, supporting RTL, HLS, and CPU-based software implementations. Using Hawkeye as an example of an advanced cache replacement policy, we highlight ttradeoffs between simulation speed, resource utilization, and design effort. Our results suggest that FPGA-accelerated simulators should provide domain-specific latency-insensitive interfaces, such as our replacement policy unit, to lower the barrier to rapid architectural exploration. In particular, HLS appears as a promising approach by offering much of the simulation acceleration while maintaining a software-like development flow. We hope that this work and our open-sourced implementation of it [1] will inspire future ideas and research in user-friendly FPGA-accelerated computer architecture simulations.

9 Acknowledgments

This work was supported in part by the F3CAS project (ANR-20-CE25-0010) of the French National Research Agency, and by the AMD University Program through the donation of Alveo FPGA boards.

References

[1] ADAC Research Group. F3CAS simulator GitLab repository, 2025. Available: https://gite.lirmm.fr/adac/f3cas.
[2] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, et al. Chipyard: An integrated SoC research and implementation environment. In *Proceedings of the Design Automation Conference (DAC)*, 2020.
[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 1966.
[5] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanovic. FASED: FPGA-accelerated simulation and evaluation of DRAM. In *Proceedings of the Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.
[6] 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*, 2011.
[7] ChampSim. ChampSim. https://github.com/ChampSim/ChampSim. Accessed: 2024-04-10.
[8] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi. ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2009.
[9] crc2. The 2nd Cache Replacement Championship. (2017). [Online]. Available: http://crc2.ece.tamu.edu/. Accessed: 2024-04-12.
[10] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.
[11] A. Jain and C. Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2016.
[12] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH computer architecture news*, 2010.
[13] 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 *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2018.
[14] spec2017. SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/. Accessed: 2024-04-28.
[15] Synopsys Inc. Haps fpga-based prototyping systems, 2007. Available: https://www.synopsys.com/verification/prototyping.html.
[16] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proceedings of the annual international symposium on Computer architecture (ISCA)*, 2010.
[17] J. Wawrzynek, H. Wu, Y. Lee, D. Patterson, et al. The BEE2: A high-end reconfigurable computing system. *IEEE Design & Test of Computers*, 2005.
[18] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 2007.