A Cache Modeling Framework for Accelerator Systems

LINGFENG PEI, School of Computing, National University of Singapore, Singapore
WEI SIEW LIEW, Kahlert School of Computing, The University of Utah, USA
UDAREE CHATHURANGEE HIRANTHIKA KANEWALA, School of Computing, National University of Singapore, Singapore
TREVOR E. CARLSON, School of Computing, National University of Singapore, Singapore

Modern hardware accelerators increasingly rely on cache-based memory systems to improve modularity and tolerate irregular memory behavior in sparse and data-dependent workloads. However, configuring accelerator caches remains challenging: designers must choose cache capacity, Miss Status Holding Register (MSHR) count, subentry provisioning, and fill bandwidth without a fast model that captures the timing-dependent behavior of non-blocking caches. Existing analytical models often focus on cache residency or miss counts, while cycle-level simulation captures miss merging and resource contention at substantially higher cost.

This paper presents PlastiCache, a timing-augmented reuse-distance model for non-blocking accelerator caches. PlastiCache represents an accelerator memory stream as address–time pairs and extends reuse-distance analysis with predicted service intervals for outstanding misses. This formulation classifies each request as a cache-resident hit, an in-flight MSHR merge, or a new off-chip miss, while incorporating miss-handling capacity, memory-service timing, and return-path bandwidth as event constraints over the request stream. As a result, PlastiCache identifies whether a cache design is limited by cache capacity, MSHR availability, subentry pressure, DRAM service, or fill bandwidth.

PlastiCache targets early-stage cache design-space exploration by preserving the relative ordering of candidate configurations. Across broad sweeps of workloads and cache designs, PlastiCache achieves a mean Spearman ranking correlation of 0.93 with RTL on latency–area tradeoff curves, while achieving geometric-mean speedups of 545× over tree-based analytical models and 303× over Verilator-based RTL simulation. We further integrate PlastiCache into an end-to-end accelerator cache sizing flow with a Chisel-based sparse matrix multiplication processing element and a synthesizable cache generator. FPGA prototyping across real-world workloads, including SuiteSparse matrices and a large language model, shows that the PlastiCache-Balanced design reduces delay–area product by 45.6% and 25.9% on average compared with MOMS and MiCache, respectively, with maximum reductions of 73.8% and 37.3%.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Hardware accelerators**; *Electronic design automation.*

Additional Key Words and Phrases: cache modeling, hardware accelerators, FPGA, non-blocking caches, miss status holding registers, design space exploration

ACM Reference Format:
Lingfeng Pei, Wei Siew Liew, Udaree Chathurangee Hiranthika Kanewala, and Trevor E. Carlson. 2026. A Cache Modeling Framework for Accelerator Systems. *ACM Trans. Arch. Code Optim.* 1, 1 (January 2026), 26 pages. https://doi.org/10.1145/3838806

Authors' Contact Information: Lingfeng Pei, lpei@u.nus.edu, School of Computing, National University of Singapore, Singapore, Singapore; Wei Siew Liew, u1529306@utah.edu, Kahlert School of Computing, The University of Utah, Salt Lake City, UT, USA; Udaree Chathurangee Hiranthika Kanewala, udareek@u.nus.edu, School of Computing, National University of Singapore, Singapore, Singapore; Trevor E. Carlson, tcarlson@comp.nus.edu.sg, School of Computing, National University of Singapore, Singapore, Singapore.

1 Introduction

Hardware accelerators have become a central mechanism for improving performance and energy efficiency in machine learning [48, 53], scientific computing [58], graph analytics [10, 11, 59], and security [29]. As accelerator systems scale, two trends have made memory-system design increasingly important. First, the slowdown of Dennard scaling and the effective end of Moore's law have made data movement a dominant bottleneck. This problem is especially severe for sparse and irregular workloads, whose memory references are shaped by runtime data structures such as index arrays and graph topology [4, 46, 55, 57]. Second, both academia [20] and industry [52] increasingly emphasize reusable and modular accelerator components to reduce non-recurring engineering cost and improve design sustainability.

These trends have motivated a shift toward cache-based memory systems for accelerators [4, 5, 55, 57]. Compared with tightly managed scratchpads, caches provide a more modular interface between processing elements and off-chip memory. They exploit temporal and spatial locality without requiring designers to manually construct operation finite-state machines (FSMs) for every workload [42]. When equipped with many Miss Status Holding Registers (MSHRs), non-blocking accelerator caches can also sustain many concurrent memory requests and merge multiple requests to the same outstanding cache line [4]. This makes them attractive for irregular workloads whose reuse exists but is difficult to schedule statically.

Adopting caches shifts the problem to jointly choosing cache capacity, MSHR and subentry counts, memory-bank organization, and fill bandwidth. These parameters interact in non-obvious ways. A larger cache may reduce capacity misses but waste area if performance is actually limited by MSHR pressure. More MSHRs may expose additional memory-level parallelism but provide little benefit if subentries or fill bandwidth are the bottleneck. Although recent accelerator-cache designs demonstrate the benefits of cache-based orchestration, their cache parameters are often presented as fixed architectural choices or evaluated for a limited set of design points [4, 5, 44, 55, 57]. What remains missing is a lightweight, reusable methodology for jointly exploring cache capacity, MSHR count, subentry provisioning, and return-path bandwidth across workloads before committing to RTL implementation.

The key challenge is that modern non-blocking accelerator caches are governed not only by *what* addresses are reused, but also by *when* requests arrive. Traditional reuse-distance analysis (RDA) can estimate whether a line is likely to remain cache-resident under a least-recently-used (LRU) replacement policy, but it does not distinguish a cache-resident hit from a request that merges with an outstanding miss. Cycle-accurate simulation can capture MSHR allocation, subentry pressure, DRAM service time, and fill contention, but is too slow for broad design-space exploration. Accelerator cache exploration therefore needs a lightweight model that remains aware of request timing, miss overlap, and finite miss-handling resources.

Many accelerators execute structured kernels under designer-defined schedules, mappings, and templates that largely fix the ordering and nominal timing of memory events. Predictable execution does not imply regular memory access. Even when addresses are sparse or data-dependent, the order and nominal issue time of memory requests can often be recovered from the accelerator schedule, mapping, generator, analytical model, or profiling trace. We capture this property using an address–time stream, which preserves the cache-visible ordering and approximate issue timing needed for non-blocking cache modeling.

In this paper, we propose PlastiCache, an early-stage, workload- and timing-aware cache exploration framework for accelerator-style kernels. Given an address–time stream, PlastiCache predicts design-point trends, resource pressure, and bottlenecks before committing to RTL implementation.

The core of PlastiCache is a timing-augmented reuse-distance formulation. The model extends conventional reuse-distance reasoning with predicted service intervals for outstanding cache misses, allowing each memory request to be classified as a *cache-resident hit* (CH), an *in-flight MSHR merge* (MH), or a *new off-chip miss* (OFF). Building on this classification, PlastiCache expresses miss overlap, merge opportunities, and resource contention as timing constraints over the address–time stream. This allows the model to go beyond hit-rate estimation and identify whether a design point is limited by cache capacity, miss-handling concurrency, memory service, or return-path bandwidth.

We validate PlastiCache through RTL ranking sweeps, runtime comparisons, and FPGA evaluation of selected configurations. Across broad sweeps of workloads and cache configurations, PlastiCache achieves a mean Spearman ranking correlation of **0.93** with RTL on latency–area tradeoff curves, while reaching **0.985** in a detailed dblp-2010 sweep. It achieves geometric-mean speedups of **545**× over tree-based models, **303**× over Verilator, and **2.5**× over a sampled cache model that omits request overlap. To demonstrate end-to-end usefulness, we integrate PlastiCache into an accelerator cache sizing flow. The flow includes a Chisel-based sparse matrix multiplication processing element, a synthesizable non-blocking cache generator, and backends for Verilator simulation and Vitis RTL kernel integration. We prototype the generated systems on a modern FPGA platform and evaluate them across real-world workloads, including SuiteSparse matrices and a large language model. Compared with two state-of-the-art accelerator caches, MOMS and MiCache, the PlastiCache-Balanced configuration reduces delay–area product by an average of **45.6%** and **25.9%**, respectively, with maximum reductions of **73.8%** and **37.3%**.

In summary, this paper makes the following key contributions:

- We introduce an address–time event abstraction for accelerator cache modeling, capturing both the cache-line address and the expected issue time of each memory request so that non-blocking cache behavior can be modeled before RTL implementation.
- We develop an event-driven reuse model that intersects reuse information with predicted miss service intervals and finite miss-handling resources, allowing each request to be classified as a cache-resident hit, an in-flight MSHR merge, or a new off-chip miss.
- We validate PlastiCache against RTL across broad cache-design sweeps, achieving 0.93 ranking similarity on latency–area tradeoff curves with much lower runtime.
- We integrate PlastiCache into an end-to-end cache sizing flow and show that its bottleneck attribution and latency–area ranking can prune expensive RTL/FPGA exploration and select resource-efficient cache configurations.

  The source code for PlastiCache is publicly available at https://github.com/lingfengpei/plasticache-taco.

2 Motivation and Background

On-chip memory orchestration is a central design choice in accelerator systems, spanning scratchpad, cache, and hybrid organizations (Table 1). Here, manual control denotes how much the software, compiler, runtime, or hardware designer must explicitly manage data placement, movement, and sharing. The key trade-off is that manual orchestration offers predictability and efficiency, while automated caching improves modularity and portability for sparse and data-dependent workloads.

2.1 Accelerator Memory Orchestration

Scratchpad-based memory systems are common in accelerators because they provide predictable buffering and fine-grained control. Designers explicitly manage data placement, refilling, and sharing through tightly coupled datapath

Table 1. Classification of Memory Orchestration Architectures

Memory Orchestration Architecture	Purpose	Example	Degree of Manual Control Required
Scratchpad Only	Customized, predictable buffering	Eyeriss [13, 14], SCNN [41], others [12, 34, 37, 38]	**Very High**
Scratchpad + Programmable Walker	Programmable access control	CoRAM [15], CoRAM++ [51]	**High**
Abstracted **Scratchpad**	Reusable scratchpad blocks	Buffets [42], Tailors [56]	**Moderate**
Cache	Automated locality management	Leap Scratchpad [1], MOMS [4, 5], MiCache [55]	**Low**
Cache + Programmable Walker	Programmable access with caching	X-Cache [44], METAL [31]	**High**
Cache + **Scratchpad**	Hybrid orchestration	Trapezoid [57]	**Low to Moderate**

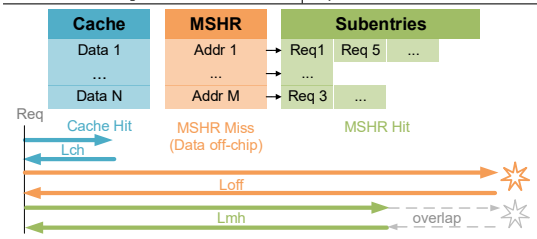

Fig. 1. Cache with MSHRs. Requests may become cache hits, off-chip misses, or MSHR-hit merges depending on cache residency and miss overlap.

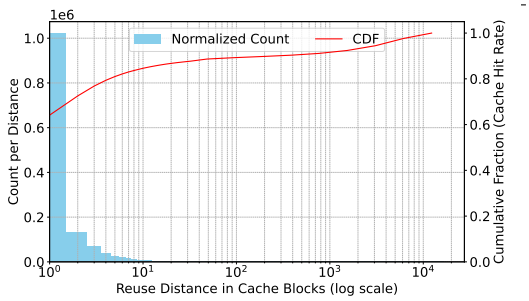

Fig. 2. Reuse distance distribution and CDF for the *DBLP-2010* dataset. The long tail indicates temporal locality across a wide range of reuse distances.

and control logic, enabling cycle-level determinism and aggressive optimization. However, this coupling increases engineering effort and limits adaptability: changes in workload structure or data format may require non-trivial redesign.

Prior work improves modularity through reusable abstractions or programmable access engines, including CoRAM [15], CoRAM++ [51], Buffets [42], and Tailors [56]. These systems reduce some manual data-movement burden, but designers must still reason explicitly about orchestration policies, which is difficult for sparse, graph, and data-dependent workloads whose references are shaped by runtime metadata.

Cache-based orchestration shifts this burden into the memory system by internalizing tag lookup, replacement, and refill logic. Caches provide a modular interface between processing elements and off-chip memory while exploiting temporal and spatial locality. Recent cache and cache-hybrid accelerators, including Leap Scratchpad [1], X-Cache [44], METAL [31], MOMS [4, 5], MiCache [55], and Trapezoid [57], reflect this trend.

However, adopting caches changes the design problem from manual movement to resource provisioning. Cache capacity, MSHR count, subentry capacity, memory bandwidth, and fill bandwidth interact: increasing one resource may not improve performance when another dominates. Accelerator cache provisioning therefore requires a timing-aware model that exposes both hit/miss behavior and non-blocking resource bottlenecks.

2.2 Non-Blocking Caches and MSHR Behavior

A key structure in non-blocking caches is the **Miss Status Holding Register (MSHR)**, which tracks in-flight misses and merges redundant requests. Accelerator caches often provision hundreds of MSHRs [4, 5] to sustain high miss concurrency.

As shown in Figure 1, a memory request can become a cache hit, an off-chip miss, or an MSHR-hit merge. A cache hit returns in L_{ch}, an off-chip miss allocates a new MSHR and returns after L_{off}, and an MSHR hit merges into an outstanding miss through a subentry. Because merging is possible only before the prior miss completes, identical address sequences can produce different cache behavior when request timings differ. Thus, modeling non-blocking accelerator caches requires reasoning about both address reuse and request timing.

2.3 Address–Time Predictability in Accelerators

Unlike general-purpose CPUs, many accelerators execute structured kernels under explicit schedules, mappings, and architecture templates. Given a workload, mapping, and accelerator template, the ordering of compute and memory events is often largely determined at design or compile time. This predictability underpins analytical tools such as Timeloop [40], MAESTRO [32], and template-based generators such as Gemini [8].

This does not require every memory address to be statically known. In sparse, graph, and input-dependent workloads, concrete cache-line addresses may be determined by runtime metadata such as index arrays, adjacency lists, or compressed tensor formats. PlastiCache only requires the cache-visible request order and approximate issue time, obtained from an accelerator schedule, mapping, analytical generator, or representative trace. We represent this information as an *address–time stream*, $\{(a_i, t_i)\}_{i=1}^{N}$, where a_i is the cache-line address of request i and t_i is its expected or profiled issue time. This stream abstracts away the internal generator while preserving the information needed to model cache hits, MSHR-hit merges, and miss overlap.

This scope targets accelerator-style kernels whose cache-visible address–time stream is directly available or can be constructed with low overhead. It differs from general CPU/GPU modeling, where deriving the cache-observed stream may require modeling instruction scheduling, warp scheduling, SIMT divergence, memory coalescing, occupancy, shared memory, and contention among concurrently resident contexts. For irregular accelerators, MSHR-rich non-blocking caches are especially attractive because they combine automatic locality capture with high-concurrency memory request service.

2.4 Why Reuse Distance Alone Is Insufficient

Classical reuse-distance analysis (RDA) characterizes locality by measuring the number of distinct cache blocks accessed between two references to the same block. It is effective for estimating cache residency under idealized replacement assumptions and can reveal whether a workload has a small or large effective working set.

Figure 2 shows the reuse-distance distribution and cumulative distribution for the *DBLP-2010* sparse matrix using 64-byte cache blocks. The long tail shows that many reuses occur at short distances, while others occur much later. Thus, cache capacity matters, but capacity alone is insufficient for accelerator cache provisioning. After short-distance locality is captured, additional cache capacity may provide diminishing returns, while performance may instead be limited by miss overlap, MSHR pressure, subentry availability, DRAM service, or fill bandwidth. Two configurations with similar RDA-predicted miss ratios can therefore perform differently if one suffers from MSHR exhaustion or fill-path congestion.

RDA answers only whether a line is likely to remain cache-resident. It does not determine whether a non-resident access becomes a new off-chip miss or merges with an outstanding miss, nor does it expose bottlenecks from finite MSHRs, subentries, DRAM banks, or fill bandwidth. These timing-dependent effects are central to non-blocking accelerator caches and motivate the timing-aware model introduced next.

3 Related Work

Prior cache and memory modeling techniques can be broadly grouped into locality-oriented cache models, detailed timing simulators, and architecture-specific analytical models. Table 2 summarizes representative approaches and their relationship to PlastiCache.

Table 2. Positioning of PlastiCache relative to representative cache, memory, and performance modeling approaches.

Approach	Input abstraction	Primary output	Timing-aware?	MSHR/subentry behavior?	Scope relative to PlastiCache
DineroIV [19]	Address trace	Hit/miss statistics	No	No	Studies capacity, associativity, and replacement, but not merged hits or miss/off-chip overlap.
RDA/AET [24, 28, 39]	Address trace / statistical reuse model	Miss-ratio curve	No	No	Captures locality trends, but omits miss timing and MSHR behavior.
Miniature simulation [50]	Sampled address/object stream	Miss-ratio curve	No	No	Efficient for capacity trends, but not accelerator timing or MSHR-resource pressure.
ChampSim [22]	Instruction trace	CPU performance/-cache statistics	CPU pipeline timing	CPU-oriented	Detailed for CPU studies, but not designed for accelerator cache/MSHR provisioning.
DRAMSim/Ramulator [18, 30, 35]	Memory request stream	DRAM timing statistics	Yes, for DRAM	No cache provisioning	Complementary memory models, but not cache/MSHR DSE models.
GPU analytical models [2, 33]	GPU kernel/hardware parameters	Kernel execution time	GPU-specific	Limited/not central	Useful for GPU throughput analysis, but not targeted at accelerator cache/MSHR sizing.
PlastiCache	Accelerator address–time stream	CH/MH/OFF, stalls, resource peaks	Yes	Yes	Early-stage ranking model for accelerator non-blocking cache DSE.

Locality-oriented cache models. Analytical cache models, such as Falcon [43] and the model of Bao et al. [6], predict cache residency or miss behavior from program structure such as affine accesses or loop nests. Trace-driven simulators such as DineroIV [19] replay address streams through configurable cache hierarchies, while statistical and reuse-distance-based methods, including AET [24] and RDA-based models [7, 28, 39], estimate miss-ratio curves from address streams or reuse distributions. Miniature simulation and synthetic trace generation [27, 50] further reduce evaluation cost by sampling or scaling the request stream. These approaches are effective for capacity, associativity, replacement, and miss-ratio studies, but they remain primarily locality-oriented: they do not directly model timing-dependent MSHR-hit merges, subentry pressure, or stalls from finite miss-handling resources.

Detailed timing and architecture-specific models. CPU microarchitectural simulators such as ChampSim [22] model instruction-trace-driven execution, including pipeline behavior, branch prediction, prefetching, and cache hierarchy effects. Although such tools could be adapted to approximate accelerator memory systems, they retain CPU-oriented machinery that is unnecessary for early-stage accelerator cache/MSHR provisioning. DRAM timing simulators such as DRAMSim and Ramulator [18, 30, 35] are complementary: they model memory-controller behavior, bank conflicts, row-buffer locality, and DRAM service timing after requests have been generated, but they do not perform cache/MSHR design-space exploration. GPU analytical models reason about GPU-specific effects such as SIMT scheduling, occupancy, divergence, and memory coalescing [2, 33]; PlastiCache instead assumes a cache-visible accelerator address–time stream and focuses on non-blocking cache resource sizing.

Positioning of PlastiCache. The key distinction is not simply that PlastiCache is trace-driven, nor that it adds MSHR parameters to a locality model. PlastiCache changes the modeling object from aggregate cache residency to event-level non-blocking cache classification. In miss-ratio or AET-style models, the primary output is a miss probability or miss count for a given capacity. In PlastiCache, the primary object is a timestamped request whose outcome depends jointly on reuse distance, outstanding miss intervals, and finite resource availability. This event-level formulation is necessary because MSHR merges, subentry pressure, DRAM-bank serialization, and fill-path contention are coupled through request timing.

4 Methodology

PlastiCache models non-blocking accelerator caches from an address–time stream. It combines reuse-distance locality with request timing to classify each access as a cache-resident hit (CH), an MSHR-hit merge (MH), or a new off-chip miss (OFF), while tracking finite MSHR, subentry, DRAM-bank, and fill-path resources for bottleneck attribution.

Request timing is a critical part of the modeling state. Consider two streams with different request timing but the same address order, $A, B_1, B_2, \ldots, B_K, A$, where K exceeds the modeled cache capacity. A reuse-distance-only model

assigns the same reuse distance to the second access to A in both streams, and therefore reaches the same capacity-based conclusion. In a non-blocking cache, however, the outcome depends on time: if the second access to A arrives before the first miss to A completes, it merges into the outstanding MSHR and becomes an MH; if it arrives after the first miss has completed and the line has been displaced, it becomes a new OFF. However, if the intervening B misses remain pending and have not yet displaced A, the second access to A can still be a cache hit once A has been filled. Thus, event classification depends on both cache residency and the service intervals of outstanding misses.

PlastiCache therefore treats each request as an address–time event and combines reuse information with predicted miss-service intervals and finite-resource constraints. This event-level formulation distinguishes CH, MH, and OFF events during early-stage non-blocking cache design-space exploration.

4.1 Applicability and Scope

Table 3. Applicability and current modeling scope.

Dimension	Supported now	Approximation	Not supported / future work
Cache	Fully associative analytical capacity proxy based on absolute reuse distance	Set associativity approximated by effective capacity and optional validation	Exact replacement and coherence
MSHR	Finite entries	Capacity constraint with completion events	Hashed collisions and lookup pipeline
Subentry	Finite entries	Per-request merge resource	Implementation-specific metadata layout
DRAM	Banks, row hit/miss behavior, and fill ports	Calibrated latency	Full DDR controller
PE	Scheduled request stream	Address–time input	Arbitrary out-of-order or control-heavy execution
Workload	Read-mostly predictable accelerator kernels	Trace-driven dynamic support	Coherent shared writes

PlastiCache targets accelerator kernels whose cache-visible request streams can be generated from a known schedule, mapping, analytical model, or representative trace. It is intended for early-stage design-space exploration, where the goal is to rank candidate cache organizations and identify dominant bottlenecks before RTL implementation. It is especially suited to read-dominant sparse and irregular workloads whose performance depends on cache capacity, MSHR availability, subentry pressure, and memory-return bandwidth.

Table 3 summarizes the scope. PlastiCache models finite MSHR and subentry resources and supports exploration over cache capacity, MSHR count, subentry count, and return-path resources. Multi-PE and multi-cache systems are within scope when their request streams can be partitioned across caches or interleaved into a shared memory-system view. The model does not target arbitrary instruction-level CPU/GPU execution, coherent shared writes, strongly cache-dependent control flow, or exact set-conflict behavior; these effects are better handled by RTL or detailed simulation after PlastiCache narrows the design space.

The abstractions in Table 3 prioritize early-stage ranking and bottleneck attribution over exact cache-organization simulation. PlastiCache therefore models the resources that most directly affect non-blocking accelerator caches: capacity, MSHR availability, subentry pressure, DRAM service timing, and fill-path bandwidth. The fully associative absolute-reuse-distance test is a lightweight capacity proxy that separates first-order locality and miss-overlap effects from lower-level set-conflict and replacement-policy details. This choice is not fundamental: set associativity can be added by maintaining per-set residency state over partitioned address–time streams, and other replacement policies can be incorporated by replacing the residency oracle. These higher-cost extensions can be conducted in later-stage validation after PlastiCache has pruned the design space.

4.2 Minimal Analytical Model and Assumptions

We use a minimal system model consisting of a processing element (PE), an on-chip cache with MSHRs and subentries, an off-chip DRAM, and a Request–Response Decoupling (RRD) buffer. This setting isolates the interaction among reuse

behavior, miss concurrency, and memory-service timing; Section 4.5 extends the same abstraction to larger accelerator organizations.

The RRD buffer is a conceptual request-tracking structure: each issued request allocates one entry and releases it only when the corresponding response returns. Its peak occupancy upper-bounds the request concurrency required to sustain the scheduled issue pattern without PE stalls, reflecting the decoupled access–execution style of latency-tolerant accelerators [9, 16, 23]. Unlike MSHRs, which track misses at cache-line granularity, the RRD buffer captures end-to-end outstanding requests independent of cache hierarchy details.

To separate intrinsic workload pressure from provisioning artifacts, we initially assume an ideal infinite-capacity RRD buffer. Under this assumption, the model input is an address–time stream, where each access contains a block address and the absolute timestamp at which the PE would issue it. The model still records peak RRD occupancy for practical sizing; the RRD becomes a bottleneck only when the implementation provisions fewer entries than this peak.

For the cache subsystem, PlastiCache explicitly models finite N_{mshr} MSHR entries and N_{sub} subentries. If a request arrives when the required resource is unavailable, the model advances the local time T_{cur} to the earliest completion event that frees resources and accumulates the resulting stall cycles. We use a fully associative capacity abstraction with absolute reuse distance, rather than exact stack distance, to keep the model lightweight.

Off-chip memory and return-path timing are modeled analytically with bank, row, and port constraints, as described next. Together, these assumptions provide a compact model that attributes performance loss to cache capacity, miss-handling structures, and memory-service bandwidth.

4.3 DRAM and Return-Path Timing Calibration

PlastiCache does not use a fixed off-chip miss latency. Instead, each OFF completion time is backend-calibrated to capture DRAM bank serialization, row-buffer hit/miss behavior, and finite return-path bandwidth. These parameters are effective cache-facing service intervals in accelerator/cache cycles and can be recalibrated for different frequencies, memory controllers, DRAM timing models, or return buses.

For cache-line address a, the model maps the request to a bank and row, $b = \texttt{bank\_index}(a)$ and $r = \texttt{row\_id}(a)$. Service begins when bank b is free:

$$t_{\text{start}} = \max(t_{\text{now}}, t_{\text{bank\_free}}[b]).$$

If the requested row is open, the access uses the calibrated row-hit service time; otherwise, it uses the row-miss/conflict service time and updates the open-row state:

$$t_{\text{dram}} = t_{\text{start}} + \begin{cases} t_{\text{row\_hit}}, & \text{if } r = \texttt{open\_row}[b], \\ t_{\text{row\_miss}}, & \text{otherwise.} \end{cases}$$

The returned line then traverses a bandwidth-limited fill path. With N_{fill} fill ports and per-line fill time t_{fill}, the final completion time is determined by the earliest available fill port. Thus, miss latency varies per request while remaining inexpensive to compute.

Table 4 reports the calibrated backend parameters used in our RTL/DRAMSim2 evaluation. Row-hit and row-miss values are effective cache-facing intervals, not direct copies of DRAMSim2 parameters such as `CL`, `tRCD`, or `tRP`. We obtain them by matching cache-line response spacing observed at the cache-facing interface in RTL simulation with DRAMSim2. For a new backend, the same parameters can be obtained from RTL measurements, a DRAM timing model, or frequency-scaled controller specifications.

Table 4. Calibrated backend parameters used in our RTL/DRAMSim2 evaluation. Timing values are expressed in accelerator/cache cycles and represent effective cache-facing service intervals, rather than direct copies of individual DRAMSim2 timing parameters.

Parameter	Fast-model value	RTL / DRAMSim2 reference	Interpretation
N_{bank}	8	DRAMSim2 `NUM_BANKS=8`	Matched to the DRAMSim2 bank configuration.
$t_{\text{row\_hit}}$	7 cycles	DRAMSim2 `CL=10`, `BL=8`	Calibrated effective row-hit service interval at the cache-facing interface; not directly equal to `CL`.
$t_{\text{row\_miss}}$	13 cycles	DRAMSim2 `tRCD=10`, `tRP=10`, `CL=10`, `BL=8`	Calibrated effective row-miss/conflict service interval at the cache-facing interface.
N_{fill}	1	RTL `numMemoryPorts=1`; single memory-return port in the reordering arbiter	Matches the RTL return-port structure.
t_{fill}	1 cycle/line	512-bit single-beat return path	One 64-byte cache line returns in one 512-bit beat.
Cache-line size	64 B	`memDataWidth=512b`	Consistent with the cache-line transfer width.
Return bus width	512 bits	`MEMDATAWIDTH=512`, `C_INVEC_DATA_WIDTH=512`	Matches the RTL/AXI-side data width.

This abstraction is simpler than a command-level DRAM simulator: it omits refresh, detailed DDR command scheduling, write-drain policies, and NoC-level interference, which are covered by the RTL/DRAMSim2 validation flow. PlastiCache instead focuses on the timing effects most relevant to cache/MSHR design-space exploration: bank serialization, row-buffer locality, and return-path bandwidth.

4.4 Modeling Algorithm

Listing 1 shows the event-driven model. Given an address–time stream, PlastiCache classifies each request as CH, MH, or OFF while enforcing finite MSHR/subentry resources, DRAM bank timing, and fill bandwidth.

Reuse measure. PlastiCache uses absolute reuse distance (RD): the number of logical requests since the previous request to the same cache block. Unlike stack distance, which counts distinct intervening blocks and gives an exact LRU test, absolute RD requires only the last logical position of each block. This reduces state and update cost for large traces, at the cost of approximating capacity behavior. This tradeoff matches PlastiCache's goal of preserving design-point trends and identifying dominant bottlenecks, rather than reproducing every replacement event.

State and classification. For each cache line, the model records the last request time, last logical index, residency state (CH or MH), and any in-flight miss completion time. Completed events are retired before processing each request. A first access becomes a compulsory OFF. Later resident accesses are CH if their RD is within the modeled capacity and OFF otherwise. If the line remains outstanding, the access merges with the outstanding miss and is classified as an MSHR hit (MH).

Timing, resources, and outputs. Timestamps advance T_{cur} and trigger event resolution. An OFF consumes one MSHR and one subentry, receives a DRAM completion time and then a fill completion time. An MH consumes only one subentry and completes with the corresponding base miss. If required resources are unavailable, the model jumps to the next completion event and records the stall. The final outputs include CH/MH/OFF counts, peak MSHR/subentry occupancy, and stall cycles, which attribute performance limits to capacity, miss concurrency, DRAM bank contention, or fill bandwidth.

4.5 System Scaling

PlastiCache scales to multi-PE and multi-cache systems by transforming system-level behavior into cache-visible address–time streams, then applying the single-cache model to each stream. This matches accelerator designs that use parallel PEs and partitioned data [4, 5, 55].

4.5.1 Multi-PE, Single Cache. When multiple PEs share one cache, PlastiCache merges their streams according to the scheduling policy. For example, a round-robin schedule can merge $(A_1, 1), (A_2, 2), \ldots$ and $(B_1, 2), (B_2, 3), \ldots$ into

Listing 1 Model Algorithm (pseudo code)

```
/* Inputs: AT[i]={time,addr}, NUM_BLOCKS, MSHR_MAX, SUB_MAX, DRAM_BANKS(2^k), T_HIT, T_MISS, FILL_PORTS, T_FILL // trace, cache/res, DRAM
 * State:  Status[a] in {CH,MH}, LastTime[a] (UNSEEN if first), LastIndex[a] (UNSEEN if first), InflightDone[a] // per-block status, last time/index
 * Res:    FreeMSHR, FreeSub (track peaks), EventQ(min-heap of (t,addr,type)); Banks/Fill states // resources and availability */
enum EvType { BASE_DONE, MERGE_DONE };      // base miss completion vs merged dependent completion
enum LineStatus { CH, MH };                 // resident line vs line outstanding (miss in flight)

inline bank_index(a){ return a & (DRAM_BANKS-1); } // power-of-two bank mapping using low bits
inline row_id(a){ return a >> log2(DRAM_BANKS); }  // row identifier above bank bits

DRAM_done(a, now){                          // schedule DRAM service and return DRAM completion time
  b=bank_index(a); r=row_id(a); ts=max(now,Bank[b].t_free); svc=(Bank[b].open_row==r)?T_HIT:T_MISS; // bank start & row hit/miss
  Bank[b].t_free=ts+svc; Bank[b].open_row=r; return ts+svc; // update bank state and output DRAM done time
}
FILL_done(now){                             // schedule return/fill path and return fill completion time
  p=argmin_p FillFree[p]; ts=max(now,FillFree[p]); FillFree[p]=ts+T_FILL; return ts+T_FILL; // serialize per-port and output fill done time
}
RESOLVE(t){                                 // retire all events completed by time t
  while(EventQ.min_time()<=t){ (td,a,type)=EventQ.pop(); FreeSub++; // process completions; each event releases one subentry
    if(type==BASE_DONE){ FreeMSHR++; Status[a]=CH; InflightDone[a]=+INF; } // base miss releases MSHR and makes line resident
  }
}
WAIT(need_mshr){                            // stall (jump time) until required resources exist
  while(FreeSub==0 or (need_mshr && FreeMSHR==0)){
    T_cur=max(T_cur,EventQ.min_time()); RESOLVE(T_cur); // advance to next completion and resolve events
    /* (optional) accumulate stall cycles here: delta = tnext - old_T_cur */ // attribute stalls to MSHR vs subentry
  }
}
ALLOC_BASE(a){                              // allocate a new base miss (off-chip miss)
  WAIT(true); FreeSub--; FreeMSHR--; d = FILL_done(DRAM_done(a,T_cur)); // consume MSHR/subentry; schedule DRAM and fill
  Status[a]=MH; InflightDone[a]=d; EventQ.push(d,a,BASE_DONE); cnt_OFF++; // record base-miss completion time
}
ALLOC_MERGE(a){                             // allocate a merged dependent (MSHR hit)
  WAIT(false); FreeSub--; d = InflightDone[a]; if(d==+INF) d=FILL_done(DRAM_done(a,T_cur)); /* rare fallback */
  EventQ.push(d,a,MERGE_DONE); cnt_MH++;    // merged request completes with the base miss
}

/* Main */
init all: FreeMSHR=MSHR_MAX, FreeSub=SUB_MAX, banks/fill free=0,
          Status[*]=CH, LastTime[*]=UNSEEN, LastIndex[*]=UNSEEN,
          InflightDone[*]=+INF, cnt_CH=cnt_MH=cnt_OFF=0;

for(i=1..N){                                // iterate over the logical request stream
  T_cur=max(T_cur,AT[i].time); RESOLVE(T_cur); a=AT[i].addr; // timestamps drive event resolution and resource timing

  first=(LastIndex[a]==UNSEEN);
  RD = first ? +INF : (i-LastIndex[a]);     // absolute reuse distance: logical requests since the previous access
  LastTime[a]=T_cur; LastIndex[a]=i;        // keep last timestamp for statistics/diagnostics; update logical position

  if(first or (Status[a]==CH && RD>NUM_BLOCKS)){ ALLOC_BASE(a); } // compulsory miss or exceeds modeled capacity
  else if(Status[a]==CH){ cnt_CH++; }       // resident line => cache hit
  else { ALLOC_MERGE(a); }                  // outstanding line => MSHR-hit merge
}
/* Outputs: cnt_CH/cnt_MH/cnt_OFF; peaks+stalls can be tracked in WAIT(). */ // final stats and sizing targets
```

one monotonically ordered cache-visible stream. The resulting model captures shared locality, MSHR pressure, and return-path contention from the combined access pattern.

4.5.2 Single PE, Multi-Cache. When data is partitioned across cache banks, PlastiCache splits the PE stream according to the partitioning or address-interleaving policy. Each per-bank stream is modeled independently, and system latency is bounded by the slowest cache bank, capturing partition imbalance without changing the single-cache abstraction.

4.5.3 Multi-PE, Multi-Cache. For multi-PE, multi-cache systems [4, 5, 55], PlastiCache first merges PE streams using the global schedule and then partitions or interleaves the merged stream across cache banks. Each bank is modeled independently, with overall performance again bounded by the slowest modeled bank.

Read/write sharing and coherence scope. This scaling strategy assumes read-dominant workloads with private cache partitions or disjoint address partitions across PEs. This matches our SpMV-based evaluation, where irregular

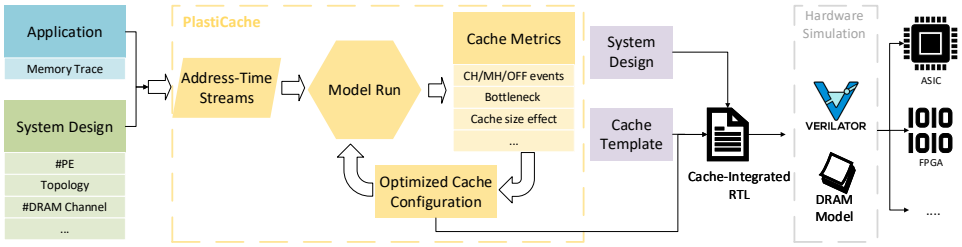

Fig. 3. System Overview.

vector accesses are read-only and the output stream is not reused through the cache. PlastiCache does not currently model true read–write sharing, dirty evictions, invalidations, or coherence transactions. Supporting these effects would require a multi-agent read/write/coherence-event stream, which is outside the scope of Table 3.

4.6 Overall Workflow

Figure 3 shows the four-step PlastiCache workflow. First, *model input construction* derives per-bank request streams from the address–time stream using system parameters such as PE count, cache banks, and interleaving policy. Second, *cache modeling and analysis* evaluates each bank and reports access events, bottlenecks, and hit rates for iterative exploration. Third, *RTL generation* emits AXI4-compliant cache modules from a parameterized template. Fourth, *system integration and validation* connects the generated cache RTL with PEs for cycle-accurate Verilator/DRAMSim2 simulation before deployment.

Integration contract. PlastiCache connects at the accelerator's cacheable memory boundary through standard AXI4 interfaces and does not require changes to the compute FSM or datapath. The designer provides only the address–time stream and system mapping configuration. For new workloads, PlastiCache is rerun to tune cache parameters, while the accelerator hardware remains unchanged unless unsupported semantics, such as coherent sharing, are introduced.

5 Implementation

5.1 Model Implementation

We implement PlastiCache as a lightweight, event-driven simulator that consumes an address stream (after the preprocessing described in Section 4) and produces (i) per-access classifications (**CH**, **MH**, **OFF**), (ii) stall-cycle breakdowns, and (iii) peak resource-usage targets.

Core data structures. Per-block state is maintained in a hash table (`std::unordered_map`) that records the last access time and the current outstanding status of each block. Completions are managed by a min-heap priority queue of future events, each containing a completion time and the associated block. We additionally track, for each inflight block, the completion time of its *base miss* (after the return/fill bottleneck) so that merged dependents can be resolved at the correct time.

Event-driven time and resource accounting. The model maintains a logical clock T (in cycles). On each input access, T is advanced and any events with completion time $\leq T$ are resolved. Resolving an event releases the corresponding miss-handling resources: **subentries** are returned for both **MH** and **OFF** events, while **MSHRs** are returned only when a **D** (base miss) completes and the block becomes resident.

When an access requires resources that are unavailable, the model stalls by jumping T forward to the next event time in the priority queue (i.e., the earliest completion), and accumulates stall cycles separately for MSHR exhaustion and subentry exhaustion. Peak MSHR/subentry usage is recorded online as a sizing target.

Per-access classification and scheduling. Per-access classification follows the analytical rules in Section 4.4: absolute reuse distance approximates cache residency, while request timing determines whether an access observes or merges with an outstanding miss.

OFF requests are scheduled through the calibrated banked-DRAM and fill-path model of Section 4.3; MH requests complete with their corresponding base miss.

Configurability and outputs. The DRAM bank count and hit/miss service times, as well as the return/fill width (number of fill ports) and per-line fill service time, are exposed as parameters for quick calibration. The implementation reports counts and rates of **CH**/**MH**/**OFF**, stall cycles attributed to MSHR and subentry shortages, peak MSHR/subentry usage, and the number of fill events observed.

5.2 Integrating with Real-World Hardware Implementation

To demonstrate the effectiveness of our model, we integrate it with two real-world cache implementations: Miss-Optimized Memory System, **MOMS** [4], and MSHR-inclusive Cache, **MiCache** [55]. Both are publicly available, FPGA-oriented designs written in *Chisel*, a high-level hardware description language based on Scala.

MOMS is an MSHR-rich cache architecture with thousands of MSHRs and subentries. It features a reorder buffer on the input side, functioning similarly to the RRD buffer in our model. However, this reorder buffer may become blocked when downstream components experience stalls. MOMS uses a set-associative cache design with a random replacement policy. **MiCache** builds upon MOMS with an MSHR-inclusive design, where both the MSHR metadata and cache tags are unified. This reduces BRAM usage and better adapts to the dynamically changing behavior of runtime workloads.

Integrating with real hardware allows us to evaluate our model end-to-end using RTL simulation and even FPGA prototypes. It also enables refinement of model parameters when available, such as the number of pipeline stages, and facilitates accurate BRAM usage estimation. If such parameters are incorporated, including L_{CH} and specific cache implementation features with the model, a rich design space can be formed to enable exploration of Pareto-optimal configurations that balance on-chip storage overhead and performance.

5.3 Building a Full Accelerator System

Both MiCache and MOMS are demonstrated using SpMV PEs. SpMV is a fundamental computation in many domains such as machine learning and graph analytics [10, 11, 46]. The data flow for this operation is illustrated in Figure 4.

The core computation in SpMV involves multiplying randomly distributed non-zero matrix elements with corresponding vector values and accumulating the results per row, as illustrated in Fig. 4(a). To compute the output vector $\mathbf{Y}$ from the matrix-vector product $\mathbf{Y} = A\mathbf{X}$ (Fig. 4(c)), the `row`, `column`, and `value` arrays are accessed sequentially, while the vector $\mathbf{X}$ is accessed irregularly, as shown in Fig. 4(d). When modeling a SpMV PE, its address-time stream is just one request per cycle in the ideal case, and therefore its absolute reuse distance is equal to the time interval.

The original MiCache and MOMS implementations build their PEs using Xilinx IP Integrator block designs. To enable cross-platform compatibility and full-system Verilator waveform simulation, we reimplemented the PE in Chisel to directly connect to Chisel-based caches, while preserving the original datapath, as shown in Figure 5. Our accelerator implementation accepts three AXI-Stream [3] input channels for the `row`, `column`, and `value` arrays, along with one AXI-Stream output channel for the result array. Additionally, an AXI4 interface is used to access the vector $\mathbf{X}$ data,

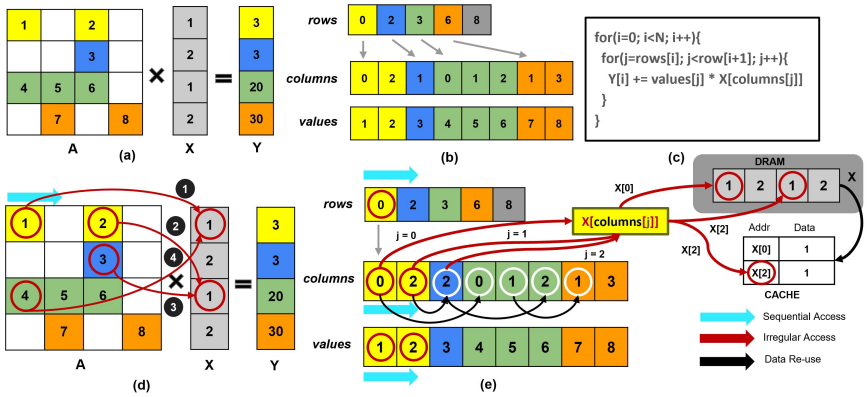

Fig. 4. SpMV CSR data flow. Shows: (a) basic SpMV computation, (b) CSR data format representation, (c) SpMV algorithm, (d) irregular data access pattern due to sequential traversal of matrix A and indirect access of vector X, and (e) data reuse opportunities in vector X that benefit from caching by reducing long-latency DRAM accesses.

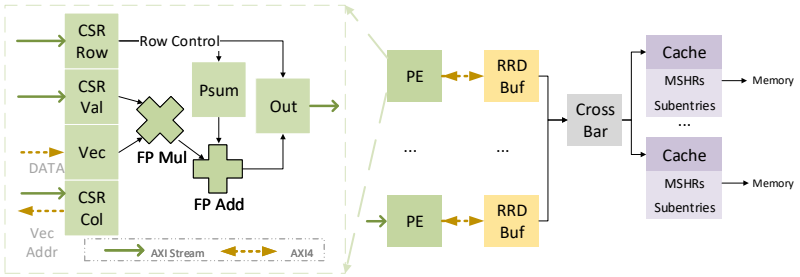

Fig. 5. Prototype Accelerator System

where the vector address is correlated with the column indices. Vector–value-pair consumption is synchronized using row-boundary information. Ultimately, the vector AXI interface is connected to the cache input AXI interface, enabling us to evaluate caching strategies in a full-system context.

The PE is implemented using a carefully designed FSM controller and queuing logic. We also introduce key circuit-level reliability features. During development, we found that certain designs could pass synthesis and RTL-level simulation, yet still fail in real hardware due to tighter timing constraints. We use an open-source 32-bit floating-point multiplier and adder based on Chisel.

After building the system, we use the Chisel compiler to emit synthesizable RTL code. For initial testing, we employ Verilator [45], a high-performance, cycle-accurate RTL-level simulator. To support simulation, we implement all necessary wrappers, signal control logic, data transfer infrastructure, and result checking mechanisms. A detailed DRAM model is integrated using DRAMSim2 [18], allowing us to accurately evaluate memory interactions and performance bottlenecks during simulation.

5.4 FPGA Prototyping

We further prototype the full hardware system on an FPGA and compare it against MiCache and MOMS. The experiments are conducted on the AMD Alveo U250 Data Center accelerator card using the Vitis toolchain. We adopt an AMD-recommended custom RTL bottom-up design flow [54]. This approach is non-trivial, as Vitis is primarily optimized for High-Level Synthesis (HLS) designs rather than RTL-based ones. The engineering effort to set up the system, including the configuration of control logic, wrapper code for interface compliance, and adherence to the host-kernel

Table 5. Characteristics of Benchmark Matrices

Benchmark Set	Matrix	Rows	Columns	Non-Zeroes (M)	Vector Size (MB)
SuiteSparse	rail4284	4,284	1,096,894	11.30	4.18
	dblp-2010	326,186	326,186	1.62	1.24
	pds-80	129,181	434,580	0.93	1.66
	amazon-2008	735,323	735,323	5.16	2.81
	flickr	820,878	820,878	9.84	3.13
	eu-2005	862,664	862,664	19.20	3.29
	webbase-1M	1,000,005	1,000,005	3.10	3.81
	com-Youtube	1,134,890	1,134,890	5.97	4.33
	in-2004	1,382,908	1,382,908	16.90	5.28
	ljournal	5,363,260	5,363,260	79.00	20.5
	mawi-12345	18,571,154	18,571,154	38.00	70.8
	road-usa	23,947,347	23,947,347	57.70	91.4
LLaMA Pruned Weights	llama_up02_weight	4,096	11,008	8.91	0.04
	llama_down04_weight	11,008	4,096	17.99	0.02
	llama_gate06_weight	4,096	11,008	2.70	0.04

communication protocol, has been automated. Specifically, control, memory, and streaming interfaces have been considered in order to conform to the Vitis kernel IP specification. Our scripts generate interface wrappers, host-side code, and corresponding TCL scripts. This automation supports flexible configurations, including varying the number of accelerators and adapting to different AXI interface types. Even the simplest one-PE/one-cache design requires more than **3,500 lines** of generated support code. The host application, written in C++ using Xilinx's native APIs, manages the lifecycle of the RTL kernel. It allocates memory on both the host and device, sets up kernel arguments, monitors execution time, and performs result verification against expected outputs. Finally, we use Vitis 2021.2 [25] to build the host executable and compile the RTL kernel into a platform-specific binary (xclbin). This binary is deployed and executed along with the host program on the Alveo U250 FPGA card [26], completing an end-to-end integration workflow from RTL to hardware prototype.

6 Evaluation

6.1 Experimental Setup

6.1.1 Comparison Setup. To assess the accuracy and efficiency of our cache modeling framework, we compare against three representative baselines:

1. Traditional RDA-based modeling (no timing). We use an open-source reuse-distance analysis (RDA) implementation [36], which employs a 2–3–4 self-balancing tree. This data structure supports efficient $O(\log n)$ update operations and is not tied to a particular ISA (CPU or GPU), enabling flexible tuning of the number of tracked entries to control memory footprint. This baseline, which is also used in recent GPU cache models [39], models cache hit/miss behavior but does *not* model non-blocking effects such as MSHRs.

2. AET modeling with sampling. AET [24] is a state-of-the-art analytical cache model. We evaluate the AET model under three modes: (i) *full* mode that processes the entire trace, (ii) *random sampling* that probabilistically selects cache blocks to monitor, and (iii) *reservoir sampling* that maintains a fixed-size set of monitored blocks to reduce memory usage. As expected, sampling reduces runtime at the cost of lower accuracy. Similar to RDA-based modeling, the AET model captures cache hits and misses but does not model MSHR behavior or request overlap.

3. Cycle-accurate RTL simulation (ground-truth baseline). We construct a cycle-level baseline using Verilator [45] and a complete RTL implementation. Specifically, we integrate our generated cache configurations with the MOMS repository to produce an RTL cache with MSHR support (Section 5.2). We also implement a Chisel-based non-blocking accelerator that continuously issues memory requests without stalling for prior responses (Section 5.3). DRAM is modeled using DRAMSim2 [18] with a detailed DDR configuration. We verify functional correctness via

output checking and collect performance statistics, including simulation runtime and internal event counters (e.g., cache hits). Finally, we calibrate refined model parameters (e.g., L_{OFF}) to match the Verilator-based setup.

To further validate end-to-end effectiveness, we compare our FPGA-prototyped accelerator against two state-of-the-art cache systems, MiCache [55] and MOMS [5]. All designs are implemented on the AMD Alveo U250 Data Center Accelerator Card using the Vitis 2021.2 toolchain. For a fair comparison, we use the same accelerator frontend and our automated generation flow (Section 5.4) to produce the processing element, platform wrapper, host interface, and integration logic across all designs.

6.1.2 Benchmark Setup. We evaluate our design using real-world datasets from the SuiteSparse Matrix Collection [17], covering diverse application domains. Key properties of the matrices used are listed in Table 5. These datasets are also employed in the MiCache and MOMS evaluation, enabling direct performance comparison.

Additionally, we include sparse weight matrices derived from the Llama2 [49] large language model. While the original weights are dense, recent sparsification techniques such as SparseGPT [21] and Simple Pruning [47] have demonstrated the effectiveness of structured magnitude-based pruning. We use pruned versions of Llama2 weights at various sparsity levels to assess the adaptability of our cache modeling framework to realistic, sparsity-driven workloads.

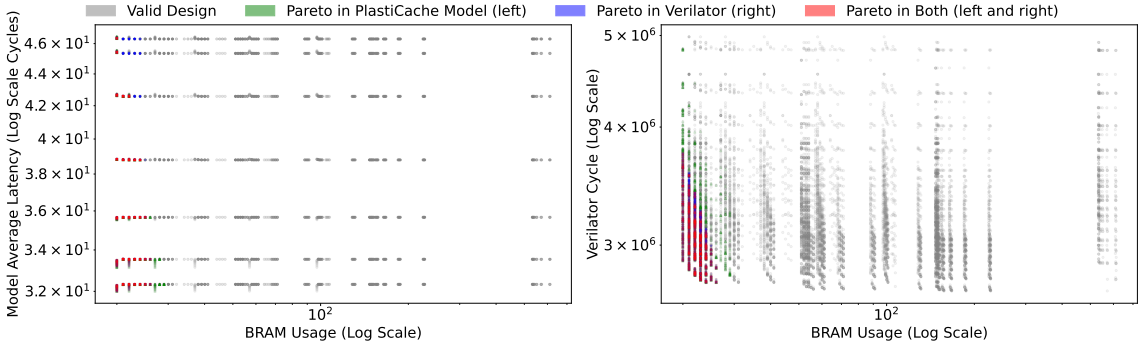

Fig. 6. Detailed design-space exploration for *dblp-2010*. Each point is a valid RTL-generated cache configuration that satisfies generator constraints, passes Chisel/Verilator elaboration and build, and completes RTL execution. The sweep varies cache capacity as well as MSHR table/entry and subentry row/entry organization. The x-axis is BRAM usage and the y-axis is latency. Green/blue mark the top 2000 latency–area configurations from PlastiCache/Verilator; red marks overlap. The overlap shows that PlastiCache preserves high-quality rankings despite RTL-specific variance.

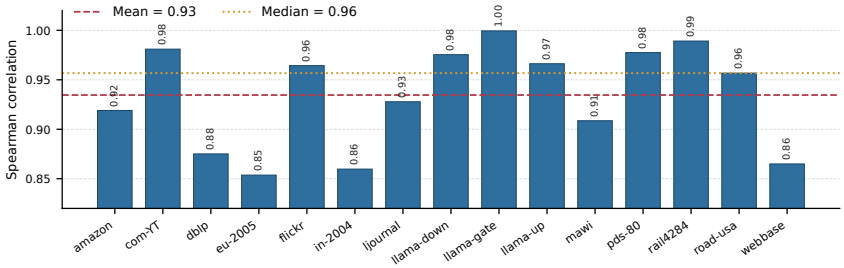

Fig. 7. Per-workload Spearman rank correlation between PlastiCache fast-model ranking and Verilator RTL simulation ranking across workloads, demonstrating that PlastiCache preserves the relative ordering of high-quality configurations across diverse workloads.

6.2 Design Space Exploration

We evaluate whether PlastiCache can guide cache design-space exploration by comparing its latency–area rankings with Verilator-based RTL simulation. Each configuration is ranked by latency–BRAM product, reflecting the goal of reducing both runtime and on-chip storage cost.

Figure 6 shows a detailed sweep on *dblp-2010*. Each point is a valid cache configuration, i.e., one that satisfies the cache generator's structural constraints, elaborates and builds in the Chisel/Verilator flow, and completes RTL execution without timeout or missing artifacts. The left plot shows PlastiCache predictions, and the right plot shows Verilator results from PlastiCache-generated RTL. Green and blue mark the top 2000 configurations ranked by PlastiCache and Verilator, respectively; red marks their overlap. We exclude unsynthesizable or invalid configurations from RTL ranking, including MOMS configurations whose subentry settings trigger hardware-level stalls or overflow checks. Despite these constraints, the high-quality regions identified by PlastiCache and RTL largely agree.

The detailed *dblp-2010* sweep intentionally covers a broader RTL-generator design space than the later cross-workload sweep. In addition to cache capacity, it varies the MSHR organization and subentry organization exposed by the cache generator. The MSHR organization is specified by the number of MSHR tables and the number of entries per table, while the subentry organization is specified by the number of subentry rows and the number of entries per row. These choices determine model-visible resources such as total cache capacity, total MSHR availability, and total subentry availability, but they also change the concrete RTL structure generated by the cache generator.

Configurations that fail to build, complete RTL execution, or pass overflow/stall checks are excluded from ranking.

Because the detailed sweep includes generator-level structural choices, the RTL measurements can vary even among configurations with similar model-visible resources. The model-predicted space is therefore more clustered than RTL because PlastiCache intentionally abstracts low-level effects such as data-port contention, MSHR hash-table collisions, signal-level timing, arbitration effects, and BRAM access constraints. Nevertheless, on *dblp-2010*, PlastiCache achieves a Spearman rank correlation of **0.985** against RTL latency–area rankings, showing that it preserves the ordering of promising design points despite RTL-specific variance.

After this detailed single-workload sweep, we evaluate cross-workload robustness using a representative subset of the broader design space. This larger DSE sweep contains 8640 design points across 15 workloads and 576 representative hardware configurations. The sweep focuses on high-impact parameters: `cacheSizeBytes` $\in \{0, 2^6, \ldots, 2^{12}\}$, `num_mshr_table` $\in \{1, \ldots, 4\}$, `mshr_per_table` $\in \{128, 256, 512\}$, and `num_subentry` $\in \{128, 256, 512, 1024, 2048, 4096\}$. Lower-level RTL choices, such as subentry row organization and BRAM mapping, are fixed in this cross-workload sweep to isolate ranking agreement over the selected high-impact DSE parameters and to keep RTL build and simulation cost manageable.

For RTL evaluation, each hardware configuration is built once and reused across all 15 workloads, avoiding a separate Chisel/Verilator rebuild for every workload–configuration pair. The fast model evaluates all 8640 points, while Verilator produces valid cycle counts for 6475 points, or 74.94% coverage. Invalid builds, missing artifacts, and timeout cases are excluded from Spearman calculation. Figure 7 reports per-workload correlations between PlastiCache and Verilator rankings by latency–BRAM product. Across all workloads, the coefficient remains above **0.85**, with mean **0.9345** and median **0.9568**, supporting PlastiCache as a practical proxy for RTL-based DSE.

6.3 Modeling Comparison

Table 6. Runtime, MSHR-Hit and OFF-miss prediction accuracy across benchmarks. The table compares simulation time (left), MSHR-Hit (middle) and absolute OFF-miss error relative to ▼ Verilator (right) for ● TreeRDA, AET-based models (■AET-Full, ▲AET-Random and ◆AET-Reservoir), and ★ PlastiCache. MH and OFF error is reported as absolute error with ▼ Verilator serving as the ground-truth reference. Shaded entries indicate the best-performing approach for each benchmark and metric (minimum runtime or minimum error). GeoMean and ArithMean summarize performance across all workloads. Entries marked ✗ denote benchmarks where the model fails to generate a prediction.

	Run Time (s)						**|Δ| MSHR Hit**		**|Δ| OFF Miss**				
Benchmark	●	■	▲	◆	★	▼	Others	★	●	■	▲	◆	★
dblp-2010	130.58	1.35	0.56	0.74	0.26	143.89	✗	0.01	0.02	0.01	0.06	0.27	0.05
amazon-2008	405.21	4.60	1.76	2.54	1.10	227.55	✗	0.00	0.04	0.04	0.06	0.22	0.07
com-Youtube	470.98	5.67	2.03	2.86	1.97	510.58	✗	0.03	0.20	0.20	0.21	0.17	0.19
eu-2005	1319.34	14.88	6.47	8.49	1.77	364.60	✗	0.29	0.06	0.01	0.03	0.35	0.13
flickr	756.54	8.85	3.34	4.35	2.63	871.40	✗	0.08	0.21	0.21	0.18	0.00	0.21
in-2004	1178.95	12.90	5.67	7.44	1.09	298.37	✗	0.21	0.01	0.00	0.02	0.18	0.07
ljournal-08	5291.08	78.73	27.11	38.61	25.48	2718.55	✗	0.07	0.20	0.21	0.21	0.12	0.21
mawi-12345	2560.71	30.47	12.78	17.18	3.80	906.33	✗	0.22	0.01	0.00	0.00	0.29	0.01
pds-80	76.07	0.73	0.31	0.50	0.14	149.63	✗	0.01	0.38	0.38	0.46	0.41	0.34
rail4284	823.45	9.99	3.81	5.54	2.29	476.65	✗	0.05	0.01	0.01	0.01	0.04	0.01
road-usa	3845.66	47.20	19.56	28.72	8.28	1393.55	✗	0.01	0.11	0.11	0.12	0.14	0.14
webbase-1M	243.17	2.46	1.03	1.42	0.41	156.74	✗	0.08	0.02	0.00	0.01	0.22	0.03
llama-d04	1287.97	13.07	5.83	6.29	1.09	506.53	✗	0.00	0.00	0.00	0.01	✗	0.00
llama-u02	662.34	6.81	2.94	3.32	0.68	308.85	✗	0.06	0.15	0.15	0.17	0.04	0.15
llama-g06	1859.91	20.07	8.95	9.99	1.50	565.89	✗	0.02	0.00	0.00	0.01	✗	0.00
GeoMean	802.58	9.03	3.68	4.91	1.47	445.40	–	0.04	0.00	0.00	0.04	0.12	0.00
ArithMean	1394.13	17.19	6.81	9.20	3.50	639.94	–	0.08	0.09	0.09	0.10	0.19	0.11

We evaluate each model by runtime and by how well it matches the ground-truth RTL simulation from ▼ Verilator (Table 6). In addition to OFF-miss accuracy, we report MSHR-hit prediction accuracy, since MSHR merges affect off-chip traffic, resource pressure, and cache configurations during design-space exploration.

For the ▼ Verilator baseline, each configuration undergoes Chisel elaboration, Verilator compilation, and cycle-accurate execution, with waveform dumping disabled to avoid excessive overhead. For ● TreeRDA, runtime increases rapidly with trace length and can take multiple days on the largest workloads; therefore, we cap the maximum recorded reuse distance to the vector size, as larger effective cache capacities are irrelevant for our target accelerator.

For accuracy, we focus on off-chip misses (OFF), which dominate SpMV-like workloads. ▼ Verilator event counters provide the ground-truth OFF count, and we report the absolute error $|\mathrm{OFF} - \mathrm{OFF}_t|$, rounded to two decimals. All experiments use a 1,KiB cache with sufficient non-blocking resources to avoid RTL stalls from resource exhaustion: 2048 MSHR entries and 6144 subentries. A single global timing-parameter set is used for all workloads and configurations, without per-benchmark tuning.

Runtime. ★ PlastiCache is consistently the fastest method. By GeoMean, it runs in 1.47 s, achieving a **545**× speedup over ● TreeRDA and a **303**× speedup over ▼ Verilator. Compared with AET-based models, ★ PlastiCache is **6.1**×, **2.5**×, and **3.3**× faster than ■ AET-Full, ▲ AET-Random, and ◆ AET-Reservoir, respectively. Across individual benchmarks, its speedups are 207.66×–1239.94× over ● TreeRDA, 106.69×–1068.79× over ▼ Verilator, and up to 13.38× over AET-based models.

MH prediction. Table 6 reports MSHR-hit (MH) absolute error relative to ▼ Verilator. ● TreeRDA and AET-based models do not model non-blocking overlap and therefore cannot predict MH behavior, marked as ✗. In contrast, ★ PlastiCache tracks outstanding requests, enforces MSHR/subentry limits, and schedules merged dependents to complete with the base miss. It thus predicts MH behavior directly, with GeoMean/ArithMean MH error of 0.04/0.08.

OFF prediction. Under the non-stalling configuration, all analytical models achieve small OFF errors. By ArithMean, ★ PlastiCache has an average OFF error of 0.11, comparable to ● TreeRDA and ■ AET-Full at 0.09, while running much faster. It is within 0.01 of the ▼ Verilator OFF reference on workloads such as `mawi-12345` and `rail4284`, with the worst-case error of 0.34 on `pds-80`. Entries marked ✗ indicate failed predictions.

Constant-window MSHR diagnostic. We also evaluate a simplified fixed-window model for MSHR events. Each miss opens a 67-cycle window, and later requests to the same line within the window are counted as MSHR hits. Across workloads, the fixed-window model has an arithmetic-mean MH error of 0.17 versus 0.08 for ★ PlastiCache, and an OFF error of 0.08 versus 0.02 in the same diagnostic run. Its OFF error is therefore 3.2× higher, and its MH error is larger and more workload-dependent. This shows that a constant reuse window cannot replace timing/resource state for modeling variable miss completion, cache-fill timing, and miss-handling interactions.

AET models capture cache residency but have two limitations here. First, sampled AET variants can fail when sampled addresses do not produce enough eviction events. Second, even ■ AET-Full cannot model OFF-to-MH transitions because it lacks timing information for overlapping in-flight misses. ★ PlastiCache addresses both issues: it is **6.1**× faster than ■ AET-Full by GeoMean while producing timing-resolved CH/MH/OFF classifications and miss-handling resource information.

These timing-resolved outputs support the non-blocking cache provisioning and DSE pruning evaluated in Sections 6.2 and 6.4.

Analytical MH estimates may diverge from hardware simulation on some workloads. Under MSHR or subentry saturation, hardware throttling and serialization reduce miss concurrency and can inflate measured MH rate relative to the analytical upper bound. These effects are intentionally not modeled. However, OFF miss counts remain accurate across benchmarks, suggesting that OFF behavior is mainly governed by structural reuse rather than realized MH rate under contention.

6.4 FPGA Prototype Results: Model-Guided Performance–Size Tradeoffs

Table 7 summarizes the hardware configurations of MOMS, MiCache, PlastiCache-Small, and PlastiCache-Balanced used in the FPGA comparison.

For the evaluated sparse workloads, PlastiCache allocates FPGA memory according to the target point on the performance–size curve. At the smallest-area end, it predicts limited marginal benefit from data-cache storage. Even small data caches require tags, valid bits, metadata, lookup logic, and banking structures, all allocated with coarse BRAM granularity; in multi-bank designs, bank-level replication and workload imbalance further increase this overhead. Under this objective, PlastiCache assigns the storage budget to miss-handling resources rather than data-cache capacity.

Miss-handling resources show stronger benefit before reaching the largest baseline configurations. Useful MSHR demand saturates before 4K entries, while MSHR storage is allocated in 512-entry BRAM-granular chunks. Similarly, once the subentry structure captures most same-line outstanding requests, additional subentries mainly increase storage without reducing off-chip traffic or PE stalls. The area-oriented design, PlastiCache-Small, therefore uses no data cache, 512 MSHR entries, and 512 subentry rows with three entries per row, preserving enough concurrency and merge capacity while reducing FPGA memory footprint. We also implement PlastiCache-Balanced as a cache-equipped point on the same model-guided performance–size curve. It keeps the same miss-handling resources as PlastiCache-Small—512 MSHR entries and 512 subentry rows with three entries per row—but adds a 64 KiB data cache. This moderate cache capacity captures residual locality and reduces execution time, while remaining much smaller than the 256 KiB baseline caches used by MOMS and MiCache. Thus, PlastiCache-Small targets the low-area region, whereas PlastiCache-Balanced trades modest additional storage for improved performance.

We compare the two PlastiCache-selected designs against two state-of-the-art accelerator cache systems, MOMS [5] and MiCache [55]. All designs use the same accelerator frontend with four processing elements and four interleaved

	MOMS	MiCache	PlastiCache-Small	PlastiCache-Balanced
RRD	1024	1024	1024	1024
DATA Cache	256 KiB, 2-way set-associative cache, random replacement	Shared tag BRAM for cache and MSHR using cuckoo hashing, 4 × 1024 metadata entries. Shared data BRAM for cache data and subentries, 256 KiB, dynamically allocated. Up to 29 subentries per data space.	None; low-area endpoint where tag/metadata overhead dominates the marginal latency benefit of very small caches under multi-banking.	64 KiB; captures additional residual locality while remaining much smaller than the 256 KiB baseline caches.
MSHR	4 tables, 1024 entries per table, cuckoo hash		512 entries; miss-concurrency demand saturates early and matches the 512-entry BRAM allocation granularity.	512 entries; same miss-concurrency target as PlastiCache-Small, while the balanced point spends additional RAM on data capacity rather than larger MSHR tables.
Subentry	4096 BRAM rows, 3 entries per row		512 rows × 3; same-line merging benefits saturate, and additional subentries increase footprint without performance gains.	512 rows × 3; retains merge capacity while exposing the performance effect of moderate cache capacity.

Table 7. Hardware architecture configuration parameters used in the FPGA comparison. For PlastiCache, we report both an area-oriented no-data-cache point and a balanced 64 KiB-cache point, showing two model-guided locations on the performance–size Pareto frontier.

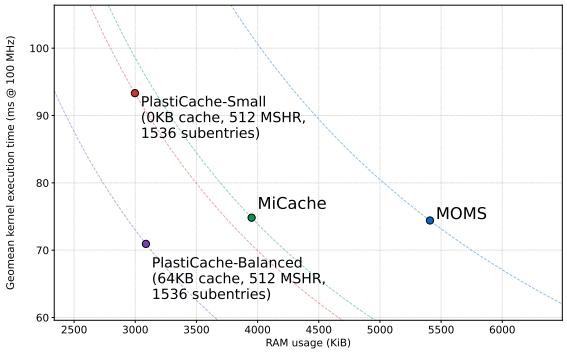

Fig. 8. Performance–RAM tradeoff of FPGA-implemented cache configurations. Each point uses the same accelerator frontend and four interleaved cache banks. PlastiCache identifies both an area-oriented point with no data cache and a balanced 64 KiB-cache point. The balanced point uses only modestly more RAM than PlastiCache-Small but substantially reduces execution time, while remaining smaller than the MOMS and MiCache baseline configurations.

Resource	MOMS	MiCache	PlastiCache Small	PlastiCache Balanced
LUT	169491	177262	166008	144915
LUTRAM	22741	23777	22151	22351
FF	265961	248282	253744	260245
BRAM	562	750	538	590
URAM	80	16	16	12
DSP	31	47	23	19
Freq (MHz)	97.0	93.6	94.4	100.0

Table 8. FPGA resource utilization for MOMS, MiCache, PlastiCache-Small, and PlastiCache-Balanced.

cache banks. For MOMS and MiCache, we use the smallest design points reported in their papers, both of which include a 256 KiB cache and 4K MSHR entries.

Table 8 summarizes post-implementation resource utilization using Vitis 2021.2 with default synthesis, placement, and routing settings. PlastiCache-Small uses fewer BRAMs than both baselines and substantially fewer URAMs than MOMS, while maintaining a comparable operating frequency. PlastiCache-Balanced spends additional RAM on its 64 KiB data cache, but its total footprint remains below the baseline cache configurations. The post-implementation results place PlastiCache-Small and PlastiCache-Balanced at distinct hardware-realizable points in the performance–size tradeoff space.

We deploy each design on the FPGA and measure total kernel execution time. The reported time includes kernel control operations such as setting arguments, but excludes host-device synchronization. Figure 8 reports geometric-mean kernel execution time versus total RAM usage. PlastiCache-Small achieves the smallest RAM footprint, but its execution time is higher because it relies primarily on miss-handling concurrency rather than cache residency. PlastiCache-Balanced adds only modest RAM relative to PlastiCache-Small, yet substantially reduces execution time. Compared with MOMS and MiCache, the balanced point occupies a more favorable delay–RAM region: it is smaller than both baselines and has comparable geometric-mean execution time.

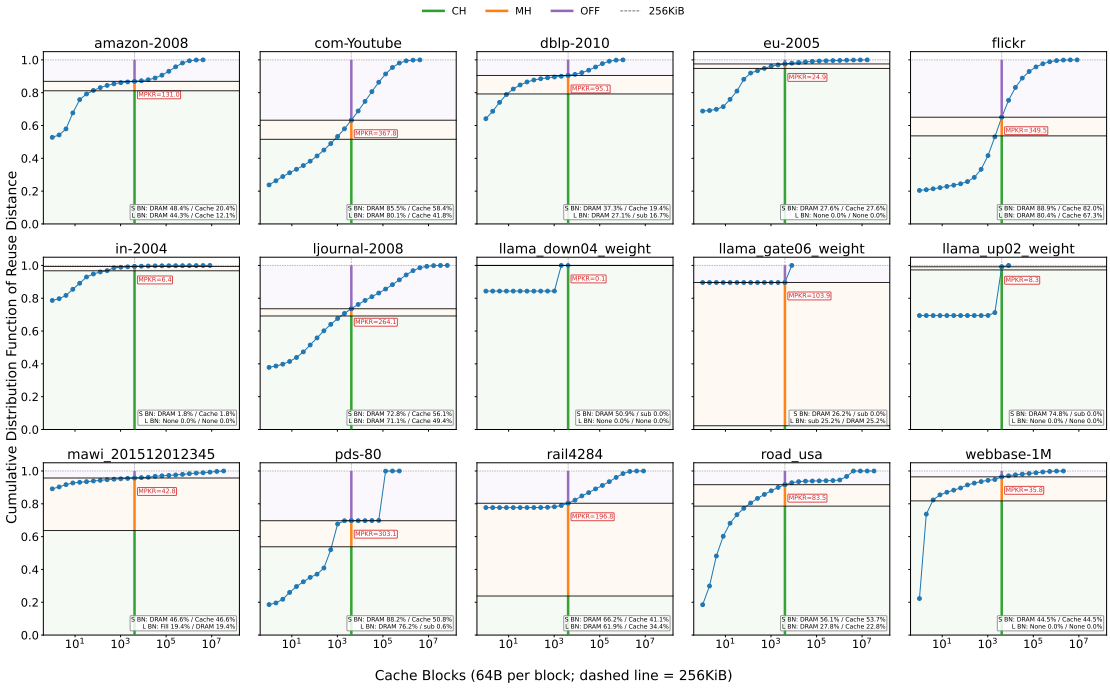

Fig. 9. Locality characterization and bottleneck attribution across sparse benchmarks. Each subplot shows the cumulative reuse-distance distribution, with the vertical dashed line marking the 256 KiB cache capacity. MPKR reports misses per thousand requests at this point. The colored bands separate cache-resident hits (CH), MSHR-hit merges (MH), and off-chip misses (OFF). Each subplot also reports the top two bottlenecks under a small baseline (S: 32 KiB cache, 32 MSHRs, 32 subentries) and a large baseline (L: 256 KiB cache, 2048 MSHRs, 4096 subentries), obtained by idealizing one resource at a time and measuring latency reduction.

6.5 Characterizing Real-World Sparse Benchmarks

Using PlastiCache, we characterize temporal locality and resource bottlenecks in real-world sparse matrix workloads. We use 256 KiB, or 4096 cache blocks with 64-byte lines, as a practical on-chip cache reference point [4, 55]. We distinguish irregularity from locality: irregularity describes how addresses are generated, while locality describes whether the resulting stream reuses cache lines within a finite window.

Figure 9 plots the cumulative reuse-distance distribution, which approximates cache hit rate as cache capacity varies from one 64-byte line to the full input-vector footprint. At the 256 KiB reference point, MPKR quantifies the remaining capacity-miss pressure, and the CH/MH/OFF partition separates cache-resident hits, MSHR-hit merges, and true off-chip accesses.

To determine the causes of bottlenecks, we use a one-at-a-time resource-relaxation analysis. For baseline latency T_{base}, we rerun PlastiCache after idealizing one resource r while keeping all others fixed, and compute $\text{gain}(r) = \frac{T_{\text{base}} - T_{\text{ideal}(r)}}{T_{\text{base}}}$. The two largest gains are reported as the top bottlenecks. We evaluate a small baseline with 32 KiB cache, 32 MSHRs, and 32 subentries, and a large baseline with 256 KiB cache, 2048 MSHRs, and 4096 subentries. Both use one PE, one memory bank, 64-byte cache lines, and the default DRAM/fill timing. The idealized runs remove cache-capacity pressure, MSHR limits, subentry limits, DRAM bank/row conflicts, or fill-path latency.

The benchmarks show diverse locality profiles, confirming that no single cache configuration is efficient for all workloads. The MPKR marker captures residual miss pressure at a realistic cache budget, while CH/MH/OFF further distinguishes locality, MSHR-level overlap, and off-chip traffic.

The bottleneck labels show that limiting resources depend on both workload and configuration. Under the small baseline, many workloads are dominated by DRAM-service serialization, often with cache capacity as the second bottleneck. Workloads such as `com-Youtube`, `flickr`, `pds-80`, `ljournal-2008`, and `rail4284` benefit from both DRAM and cache idealization, while `eu-2005`, `mawi`, `in-2004`, and `webbase-1M` show similar cache/DRAM gains because their residual misses are closely tied to backend service time. Under the large 256 KiB baseline, cache-friendly workloads such as `llama-down04`, `llama-up02`, `eu-2005`, `in-2004`, and `webbase-1M` show no dominant bottleneck, whereas high-MPKR workloads such as `com-Youtube`, `flickr`, `ljournal-2008`, `pds-80`, `rail4284`, and `road-usa` remain sensitive to DRAM banks, sometimes with cache capacity as a secondary bottleneck. The pruned LLaMA gate matrix is instead limited by subentry/DRAM effects due to frequent MSHR merging, while `mawi` shows comparable DRAM and fill sensitivity once cache and miss-handling resources are sufficiently provisioned.

These results clarify the role of the CH/MH/OFF partition. CH/MH/OFF rates describe locality and overlap, but they do not by themselves identify the limiting resource. Resource-relaxation labels provide this attribution by measuring which idealized resource most reduces latency. Thus, Fig. 9 connects benchmark locality directly to the model parameters: cache capacity, MSHR/subentry availability, DRAM banking, and fill bandwidth.

Overall, bottlenecks are properties of workload–configuration pairs rather than workloads alone. Cache-friendly applications may run well with small caches, while cache-unfriendly workloads such as `flickr` can justify larger capacity. With PlastiCache, these behaviors can be profiled quickly, enabling workload-aware cache sizing and identifying when additional cache capacity, larger miss-handling structures, or stronger DRAM/fill resources are most useful.

6.6 Robustness Evaluation

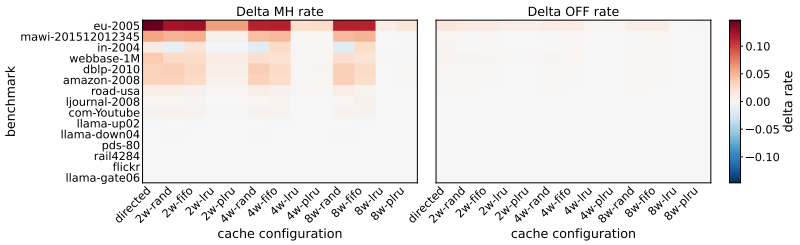

Fig. 10. Sensitivity of dynamic event rates to cache associativity and replacement policy. Each cell reports the change relative to the fully associative LRU configuration used by the RDA-based model. The left and right panels show changes in MSHR-hit (MH) rate and off-chip access (OFF) rate, respectively. Event rates are generally stable across the evaluated cache organizations, with eu-2005 showing the largest deviation.

Table 9. Synthetic address-stream evaluation. Deltas are computed as Model−RTL. MH denotes MSHR-hit rate and OFF denotes off-chip access rate.

Trace	Construction	Isolated behavior	Parameter	Δ MH	Δ OFF
Streaming	$l = 0, 1, 2, \ldots$	No temporal reuse	–	0.000	0.000
Bursty same-line	8 accesses per line	MSHR merging	$B = 8$	0.000	0.000
Zipf	$P(l) \propto 1/(l+1)^{\alpha}$	Uniform random	$\alpha = 0.0$	+0.138	−0.134
Zipf	$P(l) \propto 1/(l+1)^{\alpha}$	Medium hotspot	$\alpha = 0.8$	+0.068	−0.054
Zipf	$P(l) \propto 1/(l+1)^{\alpha}$	Strong hotspot	$\alpha = 1.2$	+0.039	−0.018

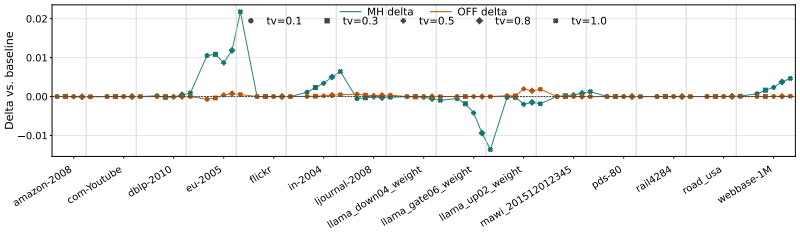

Fig. 11. Impact of timestamp perturbations on MSHR-hit and off-chip-miss rates. The address sequence is kept unchanged while request issue timestamps are perturbed with magnitude (tv). Each point reports the change relative to the unperturbed address–time stream using the same cache and memory-system parameters.

We further evaluate the robustness of PlastiCache along three dimensions: cache organization, input patterns, and request-timing noise. These checks stress the main simplifying assumptions of the model and characterize when its early-stage rankings are stable.

Sensitivity to cache associativity and replacement. PlastiCache uses a fully associative absolute-RD-based capacity abstraction to support fast design-space exploration. Fig. 10 compares the resulting event-rate predictions with hardware cache variants that use different associativity and replacement settings. Across the evaluated workloads, the MH and OFF rates remain largely stable, indicating that the dominant trends are captured by cache capacity, miss-handling resources, and memory-return timing. The largest deviations occur for eu-2005, where set-mapping and replacement effects have a stronger impact on the access stream. The cache-organization sensitivity study is therefore intended to validate that this lightweight capacity abstraction preserves early-stage CH/MH/OFF trends across common associativity and replacement-policy choices, while exact policy-specific effects remain part of later RTL validation.

Synthetic input evaluation. Table 9 evaluates PlastiCache on synthetic address–time streams that isolate basic access patterns. Streaming accesses produce no temporal reuse and match RTL exactly. Bursty same-line accesses isolate MSHR merging and are also captured exactly. For Zipf-distributed streams, the model increasingly aligns with RTL as locality becomes stronger, while the uniform-random case shows larger MH/OFF deviations because many misses overlap without stable reuse structure. These results confirm that the model behaves as expected on controlled access patterns beyond the real workloads used in the main evaluation. The remaining error in the uniform-random case highlights the expected limitation: when reuse is weak and ordering effects dominate, representative traces or detailed simulation are needed for final validation.

Sensitivity to timing perturbation. Because PlastiCache operates on an address–time stream, its predictions depend on the quality of request-timing estimates. Fig. 11 perturbs the request issue timestamp, i.e., the arrival time consumed by the model. For a timing-variation magnitude tv, each inter-request interval is multiplied by an independent factor sampled uniformly from $[1-tv, 1+tv]$, and the perturbed timestamps are reconstructed by cumulative summation. This preserves the original address order while changing the spacing between requests, thereby stressing the overlap windows that determine MSHR hits and off-chip misses. We evaluate $tv \in 0.1, 0.3, 0.5, 0.8, 1.0$; under this normalization, $tv = 1.0$ represents a substantial timing perturbation because the sampled scaling factor spans the full range from zero spacing to twice the nominal inter-request spacing. For each workload and perturbation level, we run three random seeds and report the mean change relative to the unperturbed baseline, using ΔMH and ΔOFF to denote the changes in MSHR-hit and off-chip-miss rates.

The results of this study show that the event rates remain stable across the evaluated perturbation range. In particular, ΔOFF remains close to zero for nearly all workloads, while ΔMH shows visible but still small deviations in a few cases such as `eu-2005` and `llama_up02_weight`. These cases correspond to requests whose reuse occurs near miss-completion

boundaries, where timing shifts can change whether a later request observes an outstanding miss or a completed cache fill. Overall, the experiment shows that PlastiCache does not require cycle-perfect timestamps, as long as the address–time stream preserves the main request ordering and miss-overlap structure. Very large timing distortions that no longer represent the accelerator schedule remain outside the intended scope and should be validated with detailed RTL or trace-based simulation after PlastiCache narrows the design space.

7 Conclusion

This paper presented PlastiCache, a timing-augmented reuse-distance model for early-stage exploration of non-blocking accelerator caches. PlastiCache represents accelerator memory behavior as an address–time stream and extends conventional reuse-distance analysis with predicted service intervals for outstanding misses. This allows the model to distinguish cache-resident hits, in-flight MSHR merges, and new off-chip misses while accounting for finite miss-handling capacity, memory-service timing, and return-path bandwidth.

The key insight is that many accelerator kernels expose enough scheduling and mapping structure to construct a cache-visible request stream before detailed RTL implementation, even when the addresses themselves are sparse or data-dependent. This makes it possible to reason about non-blocking cache behavior at a level that is more timing-aware than miss-count models but substantially faster than cycle-level simulation.

Across broad cache-design sweeps, PlastiCache closely preserves RTL-observed latency–area rankings, achieving a ranking similarity of 0.93 and geometric-mean speedups of 545× over tree-based analytical models and 303× over Verilator-based RTL simulation. We further integrated PlastiCache into an end-to-end accelerator cache sizing flow with a Chisel-based processing element, a synthesizable non-blocking cache generator, and FPGA prototyping. Across real-world workloads, including SuiteSparse matrices and a large language model, the PlastiCache-Balanced configuration reduced delay–area product by 45.6% and 25.9% on average compared with MOMS and MiCache, respectively.

Overall, these results show that timing-augmented reuse-distance analysis can provide a practical bridge between lightweight analytical modeling and detailed hardware validation. PlastiCache is intended to guide early cache sizing, identify dominant bottlenecks, and prune the design space before RTL implementation. Future work includes extending the model to richer coherence protocols, additional replacement policies, and more detailed off-chip memory-controller effects.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This project is supported by A*STAR under the RIE2025 Energy-aware Accelerated Computing (EAC) program (Award H25-MSR3439), the RIE2025 Innovation & Enterprise (I&E) Industry Alignment Fund - Industry Collaboration Projects (IAF-ICP) (Award H25-MCP3442), the Ministry of Education, Singapore, Tier 3 (Award MOE-MOET32024-0003), the National University of Singapore under FD-fAbrICS: Joint Lab for FD-SOI Always-on Intelligent & Connected Systems (I2001E0053), and the NUS Artificial Intelligence Institute (NAII) (Award NAII-SF-2024-004).

References

[1] Michael Adler, Kermin E Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 25–28.

[2] Yehia Arafa, Gopinath Chennupati, Atanu Barai, Abdel-Hameed A Badawy, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Gpus cache performance estimation using reuse distance analysis. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–8.

[3] ARM. 2010. *AMBA 4 AXI4-Stream Protocol.* Technical Report. ARM.

[4] Mikhail Asiatici and Paolo Ienne. 2019. Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 310–319.

[5] Mikhail Asiatici and Paolo Ienne. 2021. Large-scale graph processing on FPGAs with caches for thousands of simultaneous misses. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 609–622.

[6] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and Ponnuswamy Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–26.

[7] Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 225–236.

[8] Jingwei Cai, Zuotong Wu, Sen Peng, Yuchen Wei, Zhanhong Tan, Guiming Shi, Mingyu Gao, and Kaisheng Ma. 2024. Gemini: Mapping and architecture co-exploration for large-scale dnn chiplet accelerators. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 156–171.

[9] Tao Chen and G Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[10] Xinyu Chen, Yao Chen, Feng Cheng, Hongshi Tan, Bingsheng He, and Weng-Fai Wong. 2022. ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1342–1358.

[11] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 609–622.

[13] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[14] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[15] Eric S Chung, James C Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. 97–106.

[16] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 924–939.

[17] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[18] DRAMSim2 Contributors. 2023. DRAMSim2: A cycle accurate DRAM simulator. Retrieved Jul 27, 2023 from https://github.com/firesim/DRAMSim2

[19] Jan Edler. 1994. Dinero IV: Trace-driven uniprocessor cache simulator. *http://www. cs. wisc. edu/˜ markhill/DineroIV* (1994).

[20] Lieven Eeckhout. 2024. Toward sustainable computer systems. *Computer* 57, 2 (2024), 101–104.

[21] Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning*. PMLR, 10323–10337.

[22] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324* (2022).

[23] Tae Jun Ham, Juan L Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture*. 191–203.

[24] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:16848204

[25] Advanced Micro Devices Inc. 2022. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393). Retrieved Jul 24, 2023 from https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis

[26] Advanced Micro Devices Inc. 2023. Alveo U250 Data Center Accelerator Card. Retrieved Jul 24, 2023 from https://www.xilinx.com/products/boards-and-kits/alveo/u250.html

[27] Fan Jiang, Rafael KV Maeda, Jun Feng, Shixi Chen, Lin Chen, Xiao Li, and Jiang Xu. 2022. Fast and accurate statistical simulation of shared-memory applications on multicore systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2022), 2455–2469.

[28] Mohsen Kiani and Amir Rajabzadeh. 2018. Efficient cache performance modeling in GPUs using reuse distance analysis. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 4 (2018), 1–24.

[29] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[30] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[31] Anagha Molakalmur Anil Kumar, Aditya Prasanna, Jonathan Balkind, and Arrvindh Shriraman. 2024. METAL: Caching Multi-level Indexes in Domain-Specific Architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 715–729.

[32] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 754–768.

[33] Jan Lemeire, Jan G Cornelis, and Elias Konstantinidis. 2023. Analysis of the analytical performance models for GPUs and extracting the underlying Pipeline model. *J. Parallel and Distrib. Comput.* 173 (2023), 32–47.

[34] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 369–381.

[35] Haocong Luo, Yahya Can Tuğrul, F Nisa Bostancı, Ataberk Olgun, A Giray Yağlıkçı, and Onur Mutlu. 2023. Ramulator 2.0: A modern, modular, and extensible dram simulator. *IEEE Computer Architecture Letters* 23, 1 (2023), 112–116.

[36] mlaurenzano. [n. d.]. Reuse Distance. https://github.com/mlaurenzano/ReuseDistance

[37] Quan M Nguyen and Daniel Sanchez. 2021. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1064–1077.

[38] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 416–429.

[39] Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 37–48.

[40] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[41] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news* 45, 2 (2017), 27–40.

[42] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. 2019. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 137–151.

[43] Arjun Pitchanathan, Kunwar Grover, and Tobias Grosser. 2024. Falcon: A scalable analytical cache model. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1854–1878.

[44] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. 2022. X-cache: a modular architecture for domain-specific caches. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 396–409.

[45] Wilson Snyder. 2022. Verilator User's Guide. Retrieved Jul 27, 2023 from https://verilator.org/guide/latest/index.html

[46] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE design automation conference*. 211–216.

[47] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2023. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695* (2023).

[48] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[50] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 487–498.

[51] Gabriel Weisz and James C Hoe. 2015. CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing. In *2015 25th International conference on field programmable logic and applications (FPL)*. IEEE, 1–8.

[52] Carole-Jean Wu, Bilge Acun, Ramya Raghavendra, and Kim Hazelwood. 2024. Beyond efficiency: Scaling AI sustainably. *IEEE Micro* 44, 5 (2024), 37–46.

[53] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. 2023. Highlight: Efficient and flexible dnn acceleration with hierarchical structured sparsity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1106–1120.

[54] AMD Xilinx. 2024. Vitis Bottom-RTL-Kernel-Design-Flow. https://docs.amd.com/r/en-US/Vitis-Tutorials-Hardware-Acceleration/Bottom-RTL-Kernel-Design-Flow-Example

[55] Shaoxian Xu, Sitong Lu, Zhiyuan Shao, Xiaofei Liao, and Hai Jin. 2024. MiCache: An MSHR-inclusive Non-blocking Cache Design for FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2024, Monterey, CA, USA, March 3-5, 2024*, Zhiru Zhang and Andrew Putnam (Eds.). ACM, 22–32. doi:10.1145/3626202.3637571

[56] Zi Yu Xue, Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. 2023. Tailors: Accelerating sparse tensor algebra by overbooking buffer capacity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1347–1363.

[57] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 931–945.

[58] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.

[59] Bingyi Zhang, Hanqing Zeng, and Viktor K Prasanna. 2023. Graphagile: An fpga-based overlay accelerator for low-latency gnn inference. *IEEE Transactions on Parallel and Distributed Systems* 34, 9 (2023), 2580–2597.