VAMOs: Versatile Atomic Memory Operations for High-Performance Many-core Architectures

Joaquín Ferrer, Juan M. Cebrian, Ricardo Fernández-Pascual, Alberto Ros, and Manuel E. Acacio
Computer Engineering Department, University of Murcia, Murcia, Spain
Email: joaquin.ferrerc@um.es, jcebrian@um.es, rfernandez@ditec.um.es, aros@ditec.um.es, meacacio@um.es

***Abstract*—Atomic Memory Operations (AMOs) constitute a key mechanism for supporting synchronization in many-core architectures. These operations can be executed *near* the core (i.e., when the data is at first-level private cache) or *far* (i.e., at the shared cache). While far AMOs can mitigate coherence bottlenecks under high contention, they are suboptimal in scenarios of low contention and substantial data reuse. Consequently, achieving near-optimal performance requires a prediction-based dynamic adaptation of the execution location of the AMO. Current approaches to dynamic AMO execution address this challenge partially, since they fail to robustly track and respond to changing execution conditions.**

This work makes two primary observations to improve the near-far execution decision for AMOs. First, the behavior of AMOs can be more consistently determined by using their Program Counter (PC) rather than their target memory address, as currently done in the state of the art. Second, a global, system-level perspective is necessary to effectively determine where to execute each AMO. Building on these insights, we propose *Versatile Atomic Memory Operations* (VAMOs), a novel dynamic AMO execution approach that (i) employs a per-core, compact, PC-indexed predictor to infer the best location to execute each AMO, (ii) uses both local and global information to update the local predictors, and (iii) bypasses the private cache levels entirely for those AMOs predicted as far. This unique combination enables efficient execution of AMOs and the ability to react rapidly to changes in the sharing patterns.

Across diverse benchmarks and applications with varied AMO access and contention patterns, our simulation-based evaluation shows that VAMOs improves average performance by 21.82% and reduces storage requirements by 88.96% over the state-of-the-art DynAMO design.

***Index Terms*—atomic memory operations, microarchitecture, memory subsystem, parallel architectures**

I. Introduction

The continuous evolution towards large-scale many-core architectures and intricate parallel workloads has elevated synchronization primitives to a key determinant of system performance. Within this synchronization landscape, Atomic Memory Operations (AMOs) play a key role. AMOs are hardware instructions that perform a *read-modify-write* sequence as a single, indivisible unit to prevent data races. They are fundamental components of virtually all modern Instruction Set Architectures (ISAs), including x86 [1], Arm [2], and RISC-V [3], and constitute the core building blocks for mutexes, barriers, and lock-free data structures. These atomic instructions are available to programmers through compiler *built-in* macros [4] or specific libraries [5].

Historically, AMOs have been executed *near* the processing core, typically within the first-level data cache (L1D). In such a near AMO execution model, the cache coherence protocol ensures that the core obtains exclusive ownership of the corresponding cacheline prior to performing the modification. Although this approach is highly efficient for data that experience little or no contention, it becomes a significant performance hurdle when multiple cores repeatedly access and update the same synchronization variable (i.e., under high contention) [6]. In these cases, the associated cacheline is continuously migrated, or "ping-ponged", across the participating cores. This behavior induces substantial coherence activity, which results in significant latency overhead in obtaining the cacheline.

To address this limitation, *far* AMOs emerged as a clever alternative for these scenarios. Rather than migrating data to the requesting core, a far AMO transfers the operation itself to dedicated modules [7]–[13], or, more commonly, to the first shared level of the memory hierarchy (e.g., the Last-Level Cache or LLC), where the target data reside.[1] This design is conceptually simple yet highly effective: by executing atomic operations at a fixed location in the memory hierarchy, the performance overhead associated with repeated cacheline migrations is mitigated.

The practical relevance of this approach is evidenced by its adoption in commercial architectures. Armv8.1+ systems implement atomic instructions via the Large System Extensions (LSE), while the AMBA 5 CHI (Coherent Hub Interface) protocol [14] supports both near and far AMOs. More recent Arm's Neoverse CPUs [15], [16] support near and far AMOs, based on a pre-configured machine register [17]. However, Arm recommends a near AMO configuration due to different pitfalls [18]. ARM Cortex-A77 also implements far atomic operations to enhance scalability [19].

The RISC-V ISA provides native support for atomics through the 'A' Standard Extension, which defines AMOs such as `amoadd.w` or `amoxor.d`. While the ISA defines

This work has been funded by the Spanish Ministry of Science, Innovation and Universities (MCIN) MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS project, grant PID2022-136315OB-I00), and by the FSRM/10.13039/100007801 (SCALE project, grant 23039/GERM/25), Spain. Joaquín Ferrer is a PhD student funded by the Fundación Séneca, Región de Murcia, Spain, under grant 22723/FPI/24.

[1]The request message includes the operand, operation, and offset within the cacheline, and an acknowledgment to the requesting core is returned once the operation has been completed. Coherence actions may be required to ensure exclusive access to the cacheline at the LLC. Simple arithmetic logic is also added to each LLC bank.

the semantic behavior, the microarchitecture determines where to execute. High-performance RISC-V implementations (e.g., CVA6 [20] or BOOM [21]) are designed to support a near execution model. In a standard RISC-V TileLink [22] or ACE-based [23] system, AMOs can be intercepted by the L1D controller for near execution or encapsulated into a message and sent to the LLC for far execution.

Likewise, contemporary Graphics Processing Units (GPUs) make extensive use of far atomics at the shared second-level cache (L2) to sustain extremely high throughput demands imposed by thousands of concurrent threads [24]–[30].

Despite their potential, far atomics are not always the best solution. Under conditions of low data contention and high reuse of cachelines it is preferable to perform the AMOs locally. By doing so, the slow traversal to the LLC is replaced by fast L1D hits.

Recent prior work has sought to dynamically determine the best way to execute a specific AMO (near vs. far) through hardware-based prediction mechanisms. DynAMO [6] represents the current state-of-the-art (Sec. II-A). Located at the L1D, it dynamically selects where to execute AMOs based on the observed behavior of previously accessed data. Subsequently, Delegato [31] extended this idea to a multi-chiplet scenario, specifically targeting the reduction of cross-chiplet coherence activity by delegating atomic ownership to designated coherence controllers.

Although DynAMO highlights the benefits of enabling such dynamic AMO execution, we have identified four drawbacks that still need to be addressed to make this kind of prediction even more appealing (Sec. II-B). First, predictions in DynAMO rely on thresholds whose optimal values depend on each application. Second, DynAMO only uses local information, disregarding valuable global information available at the LLC. Third, it is overly conservative and biased towards near AMOs in order to avoid the *forever far* phenomenon, in which once an AMO is performed far, it cannot be returned back to near. Fourth, achieving the full benefits of far atomics is hindered by futile traversals of the private cache levels.

This work proposes *Versatile Atomic Memory Operations* (VAMOs), a new approach to dynamic AMO execution that overcomes the aforementioned limitations. VAMOs employs a per-core, compact predictor indexed by the Program Counter (PC) to anticipate the best way to execute each AMO. The predictor is updated not only with local information, but also with global information to precisely detect contended cachelines. Furthermore, AMOs that are executed far can bypass the private cache levels entirely, thus alleviating L1D pressure and substantially reducing their latency, which lies in the critical path of highly contended synchronization operations.

Our approach is motivated by two key observations. First, the behavior of AMOs exhibits a strong correlation with their associated PC. Thus, using the PC to index the predictor enables accurate behavioral prediction even for previously unseen memory addresses, and substantially reduces the required metadata footprint, since the number of distinct atomic instructions is typically much smaller than the number of unique atomic memory addresses they access. In addition, it allows AMOs prediction to be performed earlier in the pipeline, fully bypassing the private cache levels for far AMOs on their way to the LLC for processing.

Second, we observe that the coherence directory is aware of the sharing patterns of all cachelines, so this global information can be leveraged to assist each core's predictor in deciding whether it is better to execute the AMO near or far. Specifically, cores become effectively *blind* to sharing patterns for AMOs executed far, so it is essential to rely on global information to ensure that AMOs can be reclassified as near when their access pattern changes.

Our evaluation, conducted using a comprehensive set of benchmarks with varying AMO activity and behaviors, executed on a cycle-level simulator, provides empirical evidence for the superiority of VAMOs over the best implementation of DynAMO. Specifically, we find that VAMOs attains average performance improvements of 21.82% over DynAMO, which comes from more accurate AMO predictions and reduced latency for far AMOs. All of the above comes with a reduction of 88.96% in storage requirements, mainly due to the more compact PC-based prediction structures.

In summary, this work makes the following contributions:

- We identify the *shifting threshold values* and *forever far* issues as fundamental limitations of contemporary state-of-the-art dynamic AMO schemes.
- We introduce a PC-based prediction mechanism that improves accuracy, reduces storage requirements, and—by not using the L1D cache state information for prediction—enables early bypass of private cache levels.
- We propose a directory-assisted feedback loop that detects far AMOs quickly and effectively allows cores to dynamically recover from far-executing states.

II. Background and Motivation

An effective prediction of near-far AMO execution must generalize the behavior of atomic instructions with maximum accuracy while simultaneously minimizing storage requirements. This section analyzes the limitations of state-of-the-art proposals and discusses design alternatives for the predictor.

A. State-of-the-art near-far AMO prediction

DynAMO [6] constitutes the state-of-the-art approach to dynamic near-far AMO prediction. Specifically, Soria *et al.* identify two possible predictor implementations:

DynAMO-Metric compares the number of times each cacheline has been used to complete a near AMO and the number of times the coherence directory has invalidated it. If the result of dividing both values goes over a predefined threshold, the AMO accessing that cacheline is executed near; otherwise, far. This mechanism is only used when the cacheline is not in exclusive state in the L1D. In case of being in exclusive state, the AMO is always executed near.

DynAMO-Reuse predicts as near those AMOs for cachelines that are frequently reused in the L1D, whilst those that act on cachelines with low reuse are predicted as far. To measure the frequency of reuse, the predictor, indexed by cacheline address, includes a reuse bit that indicates whether the cacheline has been reused since it was fetched. In addition, it also includes a set of 5-bit reuse counters, which are initialized to their maximum value (31) for newly allocated cachelines. These counters are updated whenever a cacheline is invalidated or evicted from the L1D depending on the value of the reuse bit (increased if 1, decreased if 0). AMOs for cachelines obtaining a reuse counter greater than 0 are executed near. Otherwise, when the counter reaches 0, its minimum value, they execute far. When no reuse counter is found (predictor miss), the prediction falls back to a DynAMO-Metric-like policy. It accounts for the total number of cachelines brought into the L1D by near AMOs, and the total number of these cachelines that have been reused, and calculates a ratio. If this ratio exceeds a threshold, a near AMO is issued, and a far AMO is executed otherwise.

Additionally, DynAMO-Reuse can employ the state of the cacheline in the L1D to refine its prediction. Specifically, *DynAMO-Reuse-PN (Present Near)* is conservative and keeps AMOs near whenever the cacheline is already found in the L1D. *DynAMO-Reuse-UN (Unique Near)* does it only if the cacheline is found in an exclusive state, therefore being more aggressive than DynAMO-Reuse-PN in sending AMOs far.

B. Limitations of near-far AMO prediction

DynAMO is a pioneer in demonstrating that dynamic AMO execution can be valuable in general, yet we have identified four drawbacks in its approach. By addressing them, this work shows that the benefits of dynamic AMO execution can be even larger.

1) Reliance on predefined threshold values: We observe that the optimal threshold value used in DynAMO to decide between far or near depends on the workload under execution. Selecting an optimal threshold for a wide set of applications becomes a complex (if not impossible) task. This is illustrated for a subset of four applications with different AMO behaviors (for details on experimental setup and applications, see Sec. IV) in Fig. 1. The figure plots execution times, normalized to an all-near baseline, for DynAMO-Metric and DynAMO-Reuse-PN (DynAMO-Reuse-UN behaves identically in these cases) over a range of threshold values. As we can see, it is virtually impossible to find a good threshold value for all cases in DynaAMO-Metric. Larger threshold values tend to favor far-friendly workloads (*hist-3d* and *rsor*) but have an adverse effect on near-friendly workloads (*lir_fs* and *spmv*). DynAMO-Reuse-PN is less sensitive to the threshold value, although its performance still depends on it. Small threshold values preserve performance in near-friendly workloads, but only capture part of the performance benefits of far AMOs in far-friendly workloads.

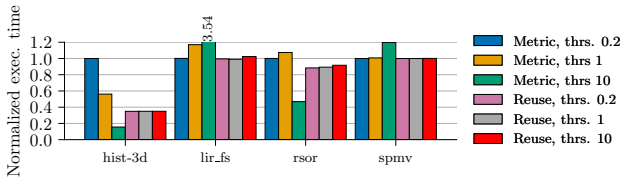


Fig. 1. Performance of DynAMO with different thresholds

2) Lack of system-wide perspective: DynAMO updates its predictor using invalidation requests as the only external event, so it indirectly infers contention on cachelines from them. However, invalidations are delayed local observations rather than direct measures of system-wide contention. More specifically, by the time an invalidation is processed by the L1D, the cacheline may already have been reused, and the feedback it provides may, therefore, be stale. For example, in DynAMO-Reuse, the increment or decrement of the reuse counter depends on a local condition (reuse bit) that may hide contention. Essentially, reuse counters are updated only when invalidating or evicting cachelines, regardless of whether the cacheline is highly contended or not. In this way, DynAMO-Reuse becomes sensitive to reuse distance: short local reuse distances mask contention, whereas long reuse distances expose it.

As a consequence, DynAMO is not able to precisely determine the global view of the actual contention of a cacheline, but only its local reuse. In this way, DynAMO misses some far-friendly behaviors that arise from global contention patterns and that are only observable by the coherence directory.

3) Lack of far-to-near recovery: DynAMO can remain in a suboptimal far prediction state after AMO transitions from a far to a near-friendly phase. For example, DynAMO-Metric uses the ratio of completed near AMOs over invalidation/eviction events to decide between near and far. However, once a cacheline is classified and executed as far, no new near AMOs are completed on it. The ratio therefore remains unchanged, and the predictor cannot transition back to the near state (the *forever far* phenomenon).

Similarly, DynAMO-Reuse predicts AMOs as far when the reuse counter reaches zero, and as near otherwise. Recovering from a far prediction state requires increasing the reuse counter, which only occurs when the corresponding cacheline is invalidated or evicted after having been reused. However, once a cacheline is predicted as far, it cannot be locally reused, no invalidation-driven update is generated, and the reuse counter never increases. Hence, the AMO remains in the far state until its corresponding entry is evicted.

More generally, both designs lack an effective mechanism to recover from far-state misclassifications. We believe that this is the reason why DynAMO-Reuse is strongly biased towards the near state at allocation time (i.e., setting the reuse counter to 31), requiring 31 decrement events before the associated cacheline can transition to far.

4) Far AMO latency is unnecessarily high: Since the most advanced DynAMO variants use the state of the cacheline at L1D to determine the prediction, AMOs to be executed far are

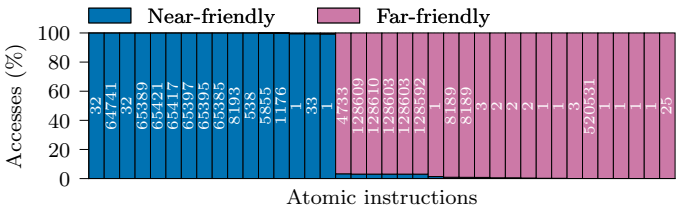


Fig. 2. Percentage of near- and far-friendly accesses for different AMOs

detected after the L1D access. Hence, they opt for traversing the private cache levels for both predicted near and far AMOs. This is problematic for the latter, as far AMOs do not need to perform any bookkeeping at the private cache levels and, thus, experience this extra latency uselessly (contrary to what happens with AMOs predicted as near). Obviously, removing this extra latency accelerates far AMOs.

C. PC- vs. target address-based prediction

Prediction mechanisms for memory operations can use the target memory address, the PC, or both to infer the behavior of the operation. As discussed, DynAMO employs the target memory address and the coherence state of the cacheline at L1D to perform the prediction. In this section, we advocate for *performing the prediction using only the PC of atomic instructions*.

Our decision is driven by the intuition that different cachelines accessed by the same atomic instruction within the same program region are likely to exhibit the same access pattern. To back up this intuition, we performed the study depicted in Fig. 2, in which we ran all our benchmarks and full applications with all AMOs issued as far to characterize each access as either near or far friendly. Upon receiving the request at the LLC, we classify it as far-friendly if the current accessing core is different from the previous, otherwise as near-friendly. Each bar represents the PC of a different AMO instruction in the evaluated applications, and reports the percentage of near-friendly and far-friendly accesses, demonstrating that most PCs are either fully near-friendly or fully far-friendly. Inside each bar, the number of different cachelines that each PC accesses is shown, to demonstrate that a single PC can capture the behavior of several cachelines.

Using the PC instead of the target address to drive the predictor has three important consequences. First, since the number of atomic instructions is typically much lower than the number of distinct cachelines accessed (see Fig. 2), the storage requirements of the predictor are smaller. Second, the predictor has a shorter learning period and is also able to anticipate the behavior of unseen cachelines. Third, predictions can be generated before the target address has been calculated at an earlier stage in the instruction pipeline, e.g., immediately after the completion of the decode stage. This opens up another important opportunity, not exploited before to the best of our knowledge: the private cache levels can be bypassed, and the AMO can be sent directly to the LLC for execution. As noted before, traversing the private cache levels for an AMO successfully predicted as far is futile and detrimental to performance.

III. VAMOs: Versatile Atomic Memory Operations

VAMOs improves upon the state of the art in AMO prediction in three different ways. First, the prediction accuracy is increased by using PC-based prediction (instead of using the target address) and, importantly, refining the prediction mechanism with global information gathered at the LLC about contention. Second, the latency of far AMOs is reduced by circumventing the private cache levels and proceeding straight to the LLC, thanks to an early prediction requiring only the PC. Third, the extra storage required to keep the prediction metadata is reduced, as PC-based indexing requires fewer entries to capture the necessary prediction information. Importantly, VAMOs does not entail any modification to the coherence protocol or existing memory structures (e.g. the L1D).

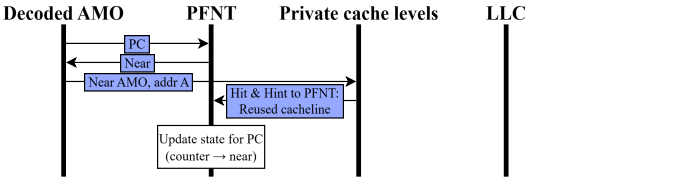
A. VAMOs overview

The near-far prediction in VAMOs is essentially performed by a PC-indexed table of saturating counters: the *Predict Far-Near Table* or PFNT (see Sec. III-C1 for organization details). AMOs predicted as far effectively bypass the private cache levels and are dispatched directly to the LLC. In contrast, near AMOs follow their course through the L1D. When a near AMO hits in any of the private cache levels (L1D or L2 in our configuration), VAMOs considers the prediction correct and updates the PFNT counter accordingly, consolidating the near prediction for the AMO. In case of a L2 miss, the request is forwarded to the LLC but the PFNT is not yet updated.

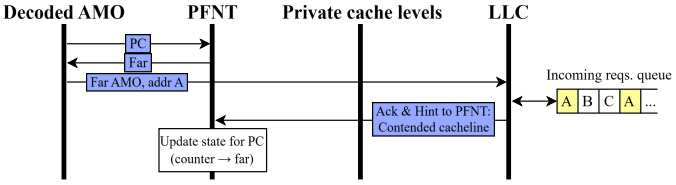
Both near AMOs missing in the private cache hierarchy and far AMOs bypassing the private cache hierarchy reach the LLC. There, the necessary coherence actions (e.g., invalidating remote copies, if any, including any copy that may be present at the requester's private cache) are carried out, ensuring that the copy in the inclusive LLC is the only valid copy. In the case of a near AMO an exclusive copy of the requested cacheline is returned to the L1D. For a far AMO, the operation is performed directly at an exclusive copy of the cacheline held in the LLC and an acknowledgment (Ack) message signaling the completion of the operation is sent to the requesting core, including the operation result. Note that the LLC does not change the decision indicated by the predictor on how to execute the current AMO.

Reuse of cachelines can be simply detected on L1D hits by near AMOs. However, detecting no reuse would require adding a reuse bit per L1D entry. As an alternative, we opt for complementing the PFNT with a *Reuse Tracking Table* or RTT (see details in Sec. III-C2). The RTT is indexed by cacheline addresses and tracks both the PC of the last AMO and if there is local reuse. Accesses to the RTT are not part of the critical path for either atomic or non-atomic memory accesses, since this table is only used to update the PFNT counter and it is not accessed at prediction time. Importantly, the PC field in the RTT is used to update the PFNT for any event in which only the cacheline, but not the PC, is known (e.g., on cache evictions/invalidations).

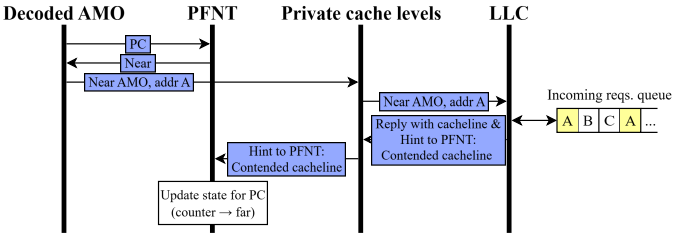
A key contribution of VAMOs is that it uses global information available at the LLC to provide valuable information



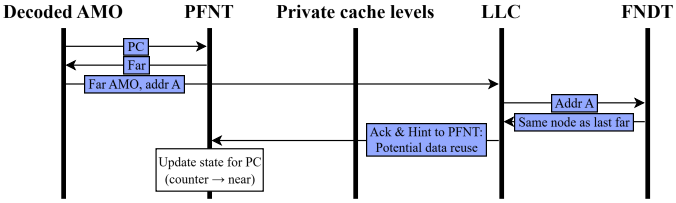
(a) Correct near prediction scenario in VAMOs



(b) Correct far prediction scenario in VAMOs



(c) Near misprediction scenario in VAMOs



(d) Far misprediction scenario in VAMOs

Fig. 3. Overview of VAMOs

to update the PFNT via *hints* transmitted along the response messages. Specifically, the LLC informs the local core about possible contention on the cacheline by examining the *incoming request queue* on the home node (details are provided in Sec. III-C3). These hints serve to update the PFNT counter for the AMO towards far.

Additionally, the LLC also checks for repeated far AMOs on the same cacheline generated by the same core. This allows the PFNT to update the counter towards near for AMOs that have been wrongly predicted as far. To detect these cases, VAMOs adds a small physical address-indexed table, the *Far-to-Near Detection Table* (FNDT), to each of the LLC banks. This table tracks the requesters of the cachelines most recently accessed via far AMOs (see Sec. III-C3 for more details).

B. Operation examples

To illustrate the interplay between the different tables employed in VAMOs, Fig. 3 shows four different AMO scenarios: two predicted as near and two as far, which we elaborate next.

Fig. 3.a illustrates a case of a correctly predicted near AMO that hits in the private cache levels. In this case, the PFNT counter is updated to reinforce this correct near prediction by using the PC kept in the RTT for the cacheline address.

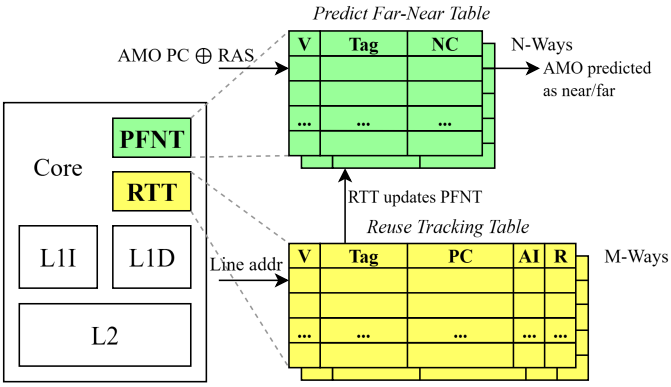


Fig. 4. Core extensions introduced by VAMOs

Additionally, the cacheline, if present in the RTT, is updated to indicate reuse.

Fig. 3.b shows a case of an AMO correctly predicted as far. In this case, the request is not forwarded to the private cache levels. Instead, the core inserts the request directly into the interconnection network and it is directly routed to the LLC. During the far AMO processing at the LLC, the incoming request queue is searched, revealing contention for that cacheline. The Ack message in response to the far AMO request includes now a hint informing about the contention observed at the LLC. This information serves the local core to confirm that the prediction was correct and the far prediction for the PC is reinforced in the PFNT.

Fig. 3.c presents a case of an AMO wrongly predicted as near. The AMO misses in the private cache levels and is forwarded to the LLC. Upon processing at the LLC, the incoming request queue is again searched, revealing contention for that cacheline. Since the AMO was originally predicted as near, the cacheline is forwarded to the requester's private caches, including the contention hint for the PFNT to update the counter towards far.

Finally, Fig. 3.d illustrates a far misprediction. As before, the incoming request queue at the LLC is checked when processing the far AMO request. This time no contention is found. Moreover, on accessing the FNDT, the LLC detects potential data reuse for that node. Hence, the LLC informs the requesting node of this situation by including this hint in the Ack message. The hint is subsequently used to update the PFNT counter towards near.

C. Implementation details

1) Predicting AMOs: the Predict Far-Near Table (PFNT): The PFNT (depicted in Fig. 4 in green) is a set-associative cache, indexed with the PC, that stores a saturated n-bit *Nearness Counter* (NC) per entry. It also includes a per-entry *valid bit* (V). To add context to the predictor, the PFNT is not indexed directly with the PC of the AMO, but with a XOR of the PC and the Return Address Stack (RAS) entries. This context is needed for AMOs used in synchronization libraries, where the same instruction can be called from

different program sites having different contention and locality properties. Including information from the RAS in the PFNT index differentiates these scenarios for the same AMO.

The PFNT is accessed prior to the initiation of the cache access. In this way, its access latency can be completely hidden and does not penalize AMO instructions with a private cache access to perform the prediction.

The NC values are updated under several conditions, as will be detailed in Sec. III-C4, to increase or decrease the confidence of the AMO associated with that PC to be executed near or far. The upper half of the counter range encodes a near AMO prediction (values from 2^{n-1} to $2^n - 1$), whereas the lower half encodes a far AMO prediction (values from 0 to $2^{n-1} - 1$). When a new entry is allocated in the PFNT, its NC is initialized to the weakest near value (2^{n-1}) for a fast transition to far.

2) Updating the PFNT: The Reuse Tracking Table (RTT): The RTT (illustrated in Fig. 4 in yellow) is a set-associative cache indexed with cacheline addresses. Entries are only allocated by AMO instructions. Each entry, apart from the valid bit (V) and Tag, records the PC of the most recent AMO that accessed it, as well as two bits that track its reuse behavior. Specifically, the *Atomic Insert* (AI) bit is set whenever a (near) AMO inserts the corresponding cacheline in L1D, regardless of whether the RTT entry is already assigned or is a new entry. The *Reuse* (R) bit is set when the cacheline is accessed by any other memory operation, signaling that it has been reused.

Upon allocation of a new entry in the RTT by an AMO instruction, its PC stored and both the AI and the R bit are set to 0. Subsequent near AMOs targeting the same cacheline and different than the one stored in the PC field, update the PC and reset the AI and R bits to track the reuse since this new near AMO. AMOs predicted as near missing in L2 set the AI bit, even if it was cleared before due to an access to the same address by a different PC. Thereafter, if any other memory access references the same cacheline and results in an L1D hit, the R bit is set. If the AI bit is not set, setting the R bit is unnecessary, because the RTT is designed to record the absence of reuse of cachelines inserted by atomic instructions.

When a cacheline is evicted or invalidated from L1D, VAMOs consults the RTT. If the AI bit is set and the R bit is not set, the AMO should have been executed in far mode, since the associated cacheline did not exhibit any temporal reuse. In this case, VAMOs uses the PC stored in the RTT to update the NC in the corresponding entry in the PFNT towards far. Conversely, if the cacheline was reused, the NC does not need to be updated at this point, as it has already been updated on each reuse event. All of the above also takes place every time an entry in the RTT is replaced. By using the AI and R bits in this way, VAMOs is able to effectively identify near predicted AMOs that do not benefit from data reuse, and therefore, that should be predicted as far in future occasions.

The RTT also serves to identify potential reuse in highly contended cachelines accessed with both atomic and non-atomic memory instructions. In this scenario, with a narrow view of only AMOs, they should be predicted as far. However, this would have a detrimental effect on the non-atomic instruction following the AMOs, as they anyway bring the cacheline to L1D. With a wide view of both atomic and non-atomic memory instructions, near execution is the preferred choice for these AMOs.

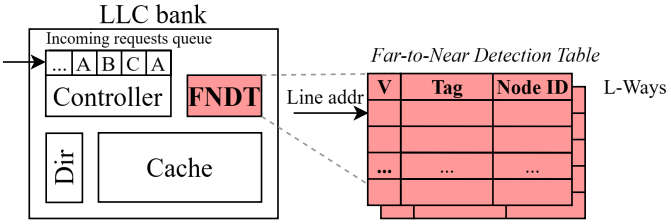


Fig. 5. LLC extensions introduced by VAMOs

The RTT also helps in this situation. When non-atomic memory operations execute, if an entry for the accessed cacheline is found, we update the PFNT, through the PC filed in the RTT, so that future AMOs with the same PC will be predicted as near. Essentially, the corresponding NC is set to the weakest near value if it was within the far values. In this way, if we later determine that it is best to predict the AMO as far, the counter can quickly switch to far.

Finally, VAMOs also employs the RTT when updating the NC after receiving the hints included in the reply messages from the LLC. Through the PC stored in the RTT, we are able to "link" the cacheline address with the entry in the PFNT containing the NC used in the prediction. The NC is then increased or decreased as appropriate.

3) Global information to improve accuracy: One of the key observations not explored in previous works that we make in VAMOs, is that the accuracy of AMO predictions can be increased if the local predictor is updated with global information provided by the LLC. VAMOs does this in two different ways.

1) The LLC recommends near for an AMO predicted as far. One of the weaknesses that we have detected in previous approaches is that when an AMO is locally predicted as far, future predictions cannot be reversed back to near (see Sec. II-B). The fact that the local core does not keep a copy of the cacheline in its private levels, prevents it from observing (through coherence messages) the activity of this cacheline. Thus, a core that predicted an AMO as far and is now the only core repeatedly accessing the cacheline would maintain the far prediction even when a near prediction would be now preferable to exploit local data reuse.

VAMOs solves this limitation by equipping the LLC with the *Far-to-Near Detection Table* (FNDT), as illustrated in Fig. 5. The FNDT is a fully associative cache that, indexed by the cacheline physical address, stores for each entry the identifier of the core that most recently accessed a cacheline via a far AMO.

On receiving a request at the LLC, and in parallel with its processing, the LLC controller accesses the FNDT. If the

request is for a far AMO, an entry is found for the target cacheline, and the core ID stored in the FNDT matches the ID of the core that issued the current far AMO request, the LLC sets a bit in the Ack reply (the *far-to-near* bit) to inform the requester of the potential local reuse. Conversely, if the request is for a non-atomic instruction, an entry is found for the target cacheline, and the stored core ID does not match the ID of the core issuing the request, the entry at the FNDT is deallocated. Upon reception of the Ack reply from the LLC, when the far-to-near bit is set, the core can infer the potential local reuse for that cacheline, updating the corresponding NC towards near.

2) The LLC recommends far for an AMO predicted as near. When several AMOs issued by different cores compete for the same cacheline, the best approach is to execute the AMOs as far. DynAMO detects this scenario *locally* at L1D, based on the received coherence messages. On the contrary, VAMOs adopts a more efficient solution based on detecting this scenario *globally* at the LLC and informing the PFNT so that it can update the NC towards far.

To this end, whenever a new request is inserted at the tail of queue of incoming requests at LLC, the queue is searched to locate the most recent preceding AMO request targeting the same cacheline and originating from a different core. If such an AMO request is found, a specific *contention bit* is set in the request to inform about the detected contention. If the new incoming request is itself an AMO, its contention bit is also set. When an AMO request reaches the head of the queue and is processed, an active contention bit is propagated back in the response from the LLC. Upon receiving a response from the LLC, if the contention bit is set, the NC is updated towards far. Note that our approach searches the queue of incoming requests at *enqueue* time rather than at *dequeue* time, when the request is going to be processed. The reason is to avoid affecting the processing time of AMOs at the LLC[2].

4) Putting it all together: updating NC values: VAMOs considers four different scenarios when updating the Nearness Counter (NC) for a predicted AMO: (i) upon completion of the AMO operation; (ii) on the eviction of a cacheline from L1D; (iii) when an entry in the RTT is evicted; and (iv) on non-atomic accesses to cachelines already accessed by far AMOs. Below, we examine each case.

(i) Update after execution of an AMO. Fig. 6 presents a flow diagram illustrating how the NC is updated upon completion of an AMO. Near AMOs hitting L1D or far AMOs where the current owner of the cacheline is the requesting core or with the FNDT determining that the requesting core is the same that performed the last far AMO for that cacheline increment the NC (towards near). Conversely, near AMOs missing in L1D and far AMOs hinted with contention from the LLC decrement the NC (towards far). In all other cases, the NC

[2]Note that, to accelerate contention detection at insertion time, the incoming request queue could be partitioned into Q distinct queues [32]. Each request could be dispatched to a particular queue based on the address of the access. This partitioning would reduce the number of requests that have to be examined during insertion, thereby improving the efficiency of the enqueue operation.

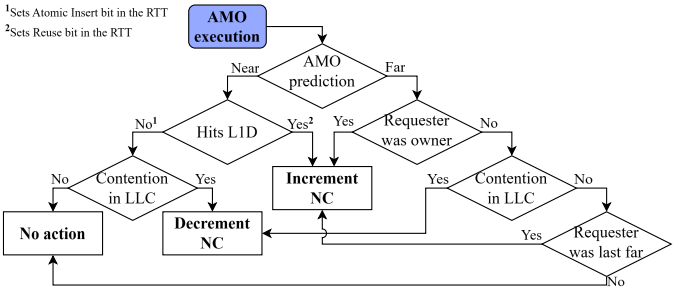


Fig. 6. Flow diagram of the NC update after AMO operation

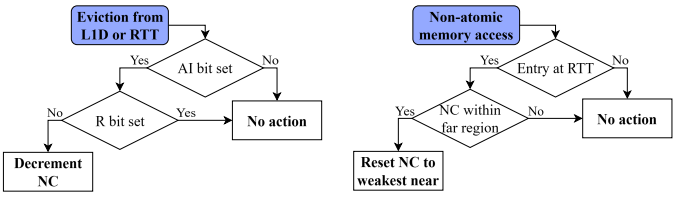


Fig. 7. Flow diagram of the NC update after an eviction from the L1D or the RTT, or after executing a non-atomic access on a cacheline previously accessed with a far AMO

remains unchanged. Note that detection of potential contention has higher priority when deciding how to update the NC.

The diagram also illustrates that near AMOs missing in L1D set the AI bit in the RTT, while near AMOs hitting in L1D set the R bit in the RTT if the AI bit was already set. The same behavior is exhibited by any non-atomic requests that access cachelines originally inserted by AMO operations.

(ii) & (iii) Update on evictions from L1D or RTT. Fig. 7 (left side) condenses how the NC is updated in these two scenarios. If the AI bit in the RTT entry is set but the R bit is not, the corresponding NC is decremented. This favors far accesses for unused cachelines.

While updating the NC after an L1D eviction is always a good decision, performing the same update on an eviction from the RTT may, in some situations, seem detrimental, specifically, when the temporal reuse window of the cacheline exceeds the effective lifetime (live window) of its corresponding entry in the RTT. In this case, the PFNT will not be able to observe and record potential reuse and future AMOs on the cacheline may be predicted as far. This could be a consequence of an underprovisioned RTT (since VAMOs is not completely insensitive to the number of addresses accessed by AMOs), or due to low temporal locality on the references to the involved cacheline. However, in the latter case, it may be, in fact, preferable to execute these AMOs as far, given that no short-term reuse is observed.

Our experiments with a large set of benchmarks (see Sec. IV) corroborate the advantages of updating the NC on RTT evictions as it enables VAMOs to filter out unused cachelines even before the line itself is evicted from L1D. This allows the predictor to transition more rapidly to far.

(iv) Update on non-atomic accesses for cachelines previously targeted by far AMOs. The final scenario in which the

NC must be updated is depicted in Fig. 7 (right side). This case corresponds to cachelines accessed with both atomic and non-atomic instructions. Since non-atomic instructions cannot execute far, the most advantageous strategy in this case is to execute the AMO near, and subsequently, exploit the resulting data locality for the non-atomic access. Unfortunately, if AMOs are executed far, there is no local information available to identify this potential reuse. Consequently, upon each non-atomic access to a cacheline previously accessed by a far AMO, the predictor sets the NC to the weakest near value.

D. Storage requirements

Finally, we specify the structural parameters for the components of VAMOs. We conducted a series of experiments in which we evaluated multiple design configurations for the PFNT, the RTT, and the FNDT, as well as different bit-widths for the NC values.

Focusing, first, on the PFNT and the size of the NC, the results indicate that the configuration delivering the highest performance employs 3 bits for storing each NC and a PFNT organized as 16 entries, 4-way 4-set-associative. In total, each PFNT entry requires one valid bit, 10 bits for the PC tag, and 3 bits for the NC, for a total of 14 bits per entry. This corresponds to a storage requirement of just 28 bytes per PFNT instance.

As for the RTT, the configuration comprising a total of 4 sets of 4 ways each (16 entries in total) yields the highest performance. Each RTT entry requires a valid bit, 20 bits for the address tag, 12 bits to store the PC, and 2 bits for the AI and R fields. This results in a total of 35 bits per entry and 70 bytes per RTT.

The FNDT is implemented as a fully associative table, with the optimal configuration consisting of only 4 entries. Each FNDT entry requires a valid bit, 20 bits for the address tag, and 9 bits for the node identifier (allowing high-scale configurations with up to 512 nodes). Hence, each entry requires a total of 30 bits, corresponding to 15 bytes per FNDT.

Overall, for each node in the system, the combined storage overhead of the PFNT, the RTT, and the FNDT amounts only to 113 bytes.

IV. Evaluation Methodology

We evaluate VAMOs using a cycle-level, in-house simulator that faithfully models an out-of-order processor implementing the RISC-V instruction set architecture. The core is driven by an instruction stream generated by Spike [33], the official RISC-V ISA simulator. The memory hierarchy is modeled using SLICC [34], coupled with the GARNET interconnection network model [35]. The simulated system approximates an Intel Alder Lake–class microarchitecture [36]. The system parameters are shown in Table I.

We compare VAMOs with the state-of-the-art AMO placement predictor, DynAMO [6]. Specifically, we consider its three variants: DynAMO-Metric (Dyn_M), DynAMO-Reuse-UN (Dyn_R_UN), and DynAMO-Reuse-PN (Dyn_R_PN). For the threshold parameters used in these variants, we explored multiple threshold values for each predictor and selected, in each case, the one that delivered the best average performance for the set of benchmarks that we employ. All other aspects of the DynAMO predictor configuration follow the specification provided by its authors. The main configuration parameters for both VAMOs and DynAMO are summarized also in Table I. We also compare VAMOs against three static AMO placement policies: (i) the All Near policy, which serves as the baseline, with all performance results normalized to this configuration; (ii) the All Far policy, which executes all AMOs far, representing the opposite extreme of the All Near policy; and (iii) the All Far Bypass policy, which executes all AMOs far while also bypassing the private caches and issuing far AMO requests directly to the LLC, analogous to the behavior of VAMOs.

We evaluate the predictors using a large variety of AMO-intensive parallel applications:

Splash-4 [41] is an improved version of the Splash-3 benchmark suite [42], which in turn, builds on the classic SPLASH-

TABLE I
System configuration

Processor (32 out-of-order cores)	
Front end	Decoupled, 64KiB TAGE-SC-L
Back end	512-entry ROB, 192-entry LQ, 114-entry SQ
Width (instr./cycle)	6 decode/alloc, 12 issue, 8 commit
Memory hierarchy	
Private L1I	32 KiB, 8 ways, 4-cycle latency
Private L1D	48 KiB, 12 ways, 5-cycle lat., stride prefetcher
Private L2	1.25 MiB, 16 ways, 10-cycle latency
Shared L3	3 MiB/core, 24 ways, 45-cycle latency, inclusive
Coherence protocol	Sparse directory-based MESI protocol
Memory	32 banks, 100-cycle latency
Interconnection network	
Topology	8 × 4 2-dimensional mesh
Latency	1 cycle route, 1 cycle link
VAMOs	
PFNT & RTT (per core)	16 entries, 4 ways in both cases
FNDT (per LLC bank)	4 entries, fully associative
DynAMO	
AMT (per L1D)	128 entries, 4 ways
Metric & Reuse thresholds	0.2 & 0.5 resp.

TABLE II
Benchmark inputs and properties

	Name	Code	Input	APKI	Near/Far
Splash-4	**FMM**	fmm	16 K particles [37]	0.3	Near
	Ocean_cp	ocean_cp	514×514	1.0	Near
	Radiosity	radiosity	room [37]	1.4	Near
	Raytrace	raytrace	car [37]	0.3	Far
CircusTent	**Random Access**	rand	8 MiB	166.6	Far
	Stride-1	stride1	8 MiB	333.3	Near
	Stride-N (N=9)	strideN	72 MiB	249.9	Near
	Pointer Chase	ptrchase	8 MiB	200.0	Far
	Central	central	8 MiB	333.3	Far
	Scatter	scatter	8 MiB	374.9	Near
	Gather	gather	8 MiB	374.9	Near
	Scatter/Gather	sg	8 MiB	399.9	Near
	Histogram-3D	hist-3d	IMG [38], [39]	124.9	Far
	Histogram	hist	32k iters	250.0	Far
	Radix Sort	rsor	2 MB vector	108.6	Far
	SPMV	spmv	rma10 [40]	69.6	Near
	Local Inc-Reuse	lir	32k iters	99.9	Near
	lir False Sharing	lir_fs	32k iters	99.9	Near
	Local Inc-No Reuse	linr	32k iters	199.9	Near
	linr False Sharing	linr_fs	32k iters	199.9	Far
	Local-to-Global Inc	lgi	32k iters	235.4	Far
	Global-to-Local Inc	gli	32k iters	114.2	Near
	Multi-Address	multi_addr	32k iters	125.0	Far
	Multi-Call	multi_call	32k iters	142.8	Far

2 [37] and eliminates important performance bugs, data races, and improper synchronization. Splash-4 applies modern programming techniques to improve scalability on contemporary hardware by replacing all possible critical sections with atomic operations or other lock-free constructs. We selected the four applications with higher APKI: *FMM*, *Ocean_cp*, *Radiosity* and *Raytrace*. For the rest, AMO execution does not impact performance.

CircusTent [43] is an open source benchmark suite for shared and distributed memory systems designed to measure the performance of a target architecture's memory subsystem with respect to AMOs. Its different kernels represent distinct memory access patterns.

Histogram-3D is a reduction kernel based on the OpenCV [44] color histogram program.

Histogram is a similar reduction kernel that only performs one atomic reduction (Histogram-3D does 3, one per RGB channel).

Radix Sort is a shared memory multithread sorting algorithm that uses a shared array to sort a vector of numbers, similar to the load-balanced parallel radix sort [45].

SPMV [46] is a sparse matrix-vector multiplication kernel where matrices use the compressed sparse column format.

Local Inc-Reuse, *Local Inc-Reuse-False Sharing*, *Local Inc-No Reuse*, and *Local Inc–No Reuse–False Sharing* represent a family of variants derived from the same *Local Inc* kernel, where each thread that performs an accumulation operation on a distinct variable. Each variant corresponds to a distinct configuration of data reuse and false-sharing behavior.

The *Local-to-Global Inc* and *Global-to-Local Inc* kernels are two-phase applications. In the former, the execution is partitioned such that the initial 25% of the iterations correspond to a near-friendly phase, whereas the remaining 75% correspond to a far-friendly phase. The latter kernel exhibits the inverse distribution of phases.

Finally, we present the *Multi-Address* and *Multi-Call* kernels. The *Multi-Address* kernel consists on an AMO performing reductions to two memory addresses: one highly local to the thread/core, and another address highly contended. The far-friendly and near-friendly accesses are interleaved, and we will test different interleave lengths.

The *Multi-Call* kernel aims to reflect the behavior of a synchronization library. It consists of a function with a single AMO which is called at various points in the code with different localities and contention scenarios. In this kernel, the near-friendly and far-friendly calls are completely interleaved (i.e., every near-friendly call to the function is followed by a far-friendly call, and vice versa).

Table II summarizes the characteristics of each application. The near- or far-friendly classification is derived from the relative performance of the All Near and All Far static policies for each benchmark. For all CircusTent applications, each thread executes 16k iterations. All the dynamic behavior benchmarks (*gli*, *lgi*, *multi_addr* and *multi_call*) will be evaluated to demonstrate the superior adaptability of VAMOs over DynAMO.

V. Results

A. VAMOs contention and reuse characteristics

Fig. 8 illustrates the classification that VAMOs assigns to each AMO, as well as the contention and reuse characteristics identified in the system. Applications on the X-axis are ordered from those exhibiting predominantly near-friendly behavior (left) to those exhibiting predominantly far-friendly behavior (right). The Y-axis denotes how atomic instructions were executed (Near/Far) and the fraction of the application's AMOs that fall into each classification category as described next. For an AMO predicted as near, three scenarios may occur: (i) the AMO hits in the L1D, indicating that the near prediction was accurate; (ii) the AMO is forwarded to the LLC, where the corresponding cacheline is found to be contended, in which case a far AMO might have been preferable to mitigate contention; or (iii) the AMO reaches the LLC and the cacheline is not contended, in which case the near AMO prediction was adequate.

For far-predicted AMOs, we adopt an analogous classification: (i) a far AMO whose requester is identical to the last requester for that cacheline, that indicates we may want to switch to near prediction. (ii) a far AMO targeting a contended cacheline, in which case the far AMO is expected to help alleviate contention; and (iii) a far AMO targeting a cacheline that is neither contended nor reused, where the far AMO may proactively contribute to preventing future contention on that cacheline and mitigate pollution on the private cache by data that will not be reused.

Fig. 8 shows how VAMOs adapts to near-friendly applications/phases (from left to right up to *fmm*), with a predominant near-predicted AMO that successfully hit in the L1D cache. This behavior arises because these applications experience negligible contention, such that the initial near prediction remains valid throughout the entire execution. Nevertheless, some specific benchmarks warrant a more detailed discussion.

For the Local Increment-Reuse False Sharing (*lir_fs*) benchmark we can see around 20% of contended lines classified as near. The presence of false sharing causes contention at the LLC, pushing the prediction towards far. However, the subsequent reuse by a load following the AMO restores the NC value to a near prediction value. By doing so, VAMOs can favor the local reuse observed for the cacheline in the next predictions for the AMO.

In the Scatter/Gather (*sg*) benchmark, there is a high number of requests to previously unseen cachelines (classified as near with no contention, as they correspond to *cold* cachelines). However, due to the temporal locality exhibited later in the execution, the predictor ultimately classifies most accesses as near AMOs with L1D hits, reducing the ratio of near with no contention to around 25%.

When analyzing the *Scatter* and *Gather* benchmarks separately, this reuse pattern does not occur for one of the three distinct memory access instructions in the application. Consequently, all AMOs issued by that particular instruction are consistently predicted as far, since no reuse is observed. In

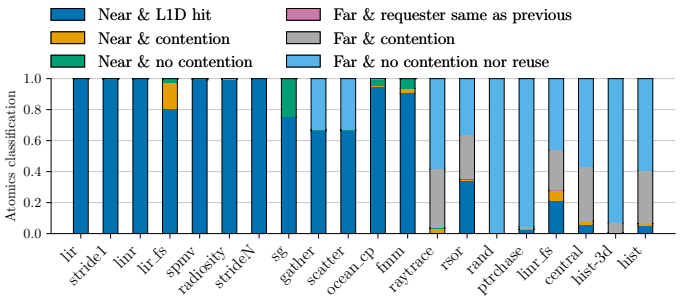


Fig. 8. Classification of AMOs performed by VAMOs and the corresponding scenarios observed in the system

contrast, the remaining two instructions exhibit sufficient reuse for their respective AMOs to be predicted as near, allowing these accesses to be satisfied by the L1D. The use of PC-based indexing in the prediction mechanism is crucial for exposing this behavior.

We now consider far-friendly applications, for which the predominant prediction outcome is far AMOs, typically targeting either contended or non-contended cachelines. Applications such as *rand* and *ptrchase* neither exhibit contention nor show data reuse. Consequently, their predictors transition to issuing far AMOs. By contrast, other applications such as *raytrace*, *rsor*, *central*, *hist-3d*, *hist*, *linr_fs*, target contended lines. In these cases, initially issued near AMOs encounter contention at the LLC, and VAMOs eventually switches to far AMO predictions. However, far AMOs do not necessarily experience the same level of contention upon reaching the LLC. This observation reinforces the conclusion that correctly identifying far AMOs not only reduces execution time under contention, but also mitigates the contention at the LLC itself.

This behavior arises because, when issuing a near AMO to a contended cacheline, the AMO must wait until the corresponding cacheline has been invalidated in the private cache of its previous owner. During this invalidation interval, additional near AMO requests targeting the same cacheline may arrive at the LLC, thereby exacerbating contention. In contrast, when AMOs are issued as far, by the time the request reaches the LLC, the preceding AMOs, also issued as far, will typically have been completed, therefore avoiding invalidations or waiting periods.

The AMO can then be executed directly at the LLC, reducing the possibility of another far AMO targeting the same cacheline to arrive at the LLC concurrently, and thus reducing contention at the LLC. As a result, many far AMOs are classified as far AMOs that access a cacheline that is effectively non-contended at the time of service. Although the cacheline is globally contended over the course of execution, the probability of contention at the specific instants when each far AMO is processed at the LLC is significantly reduced.

To substantiate this observation, we quantified the residence time at the LLC for each atomic access, i.e., the number of cycles an AMO request spends at the LLC request queue before being completed. We found that, for applications that issue AMOs to highly contended cachelines (e.g., *raytrace*, *rsor*, *linr_fs*, *central*, *lgi*, *hist-3d*, *hist*), the time spent in the LLC is substantially reduced. On average, the LLC residence time of AMOs decreases from 108.34 cycles to 18.42 cycles (83% reduction) under VAMOs compared with the All Near policy. Moreover, for these applications, as well as for applications that exhibit little or no data reuse (and therefore cause VAMOs to issue far AMOs), the time spent in the private caches is also reduced. Because far AMOs bypass the private cache levels, the average residence time at the L1D and L2 levels is lowered. Overall, when far AMOs are issued appropriately, the end-to-end latency of atomic accesses is reduced by 48% relative to the All Near policy.

B. Atomic distribution (DynAMO vs VAMOs)

We next analyze the distribution of AMOs comparing VAMOs against DynAMO-Metric (Dyn_M), DynAMO-Reuse-PN (Dyn_R_PN), and DynAMO-Reuse-UN (Dyn_R_UN), classifying them into near and far AMOs (Fig. 9). The first observation is the increased ratio of far predictions that VAMOs generally reach in far-friendly applications. Even in some cases where the different variants of DynAMO completely fail to anticipate far AMOs (namely, *linr_fs*, *central*, and *hist*), VAMOs successfully detect them. The second observation is that VAMOs preserves the same high prediction accuracy as DynAMO in near-friendly applications.

Overall, the observations above are a direct result of the fact that predictions in VAMOs closely match workload behavior and, as will be seen next, will lead to significant performance improvements (around 34% on average). This shows the adaptability of VAMOs to contention and reuse, outperforming all DynAMO variants.

Regarding the low adaptability of DynAMO, the threshold selected for Dyn_M avoids performance degradation in near-friendly applications, but it also prevents far-friendly applications from exploiting the benefits of far AMOs. Dyn_R_PN and Dyn_R_UN have similar issues for applications operating on highly contended cachelines (e.g., textitlinr_fs, *central*, *hist*), only showing good adaptability for *raytrace*, *rsor*, *rand*, *ptrchase* and *hist-3d*.

The limited adaptability of DynAMO-Reuse arises from both how contention is anticipated and how the DynAMO predictors are searched. In workloads that combine contention with short reuse distances, local reuse is observed first and the predictor remains biased towards near AMOs, even when far AMOs would be preferable. Consequently, for workloads such as *linr_fs*, *central*, and *hist*, it does not effectively transition to far AMOs, whereas in *raytrace*, *rsor* and *hist-3d*, which exhibit longer reuse distances, contention can be detected more accurately. Likewise, in *rand* and *ptrchase*, where reuse is limited, DynAMO-Reuse easily adapts to far AMOs. *Scatter* and *gather* expose a different limitation: DynAMO predictors cannot detect the absence of reuse of one of the instructions because they are indexed with the target memory address. Conversely, the fact that VAMOs employs a PC-indexed local predictor allows it to capture multiple cachelines accessed by

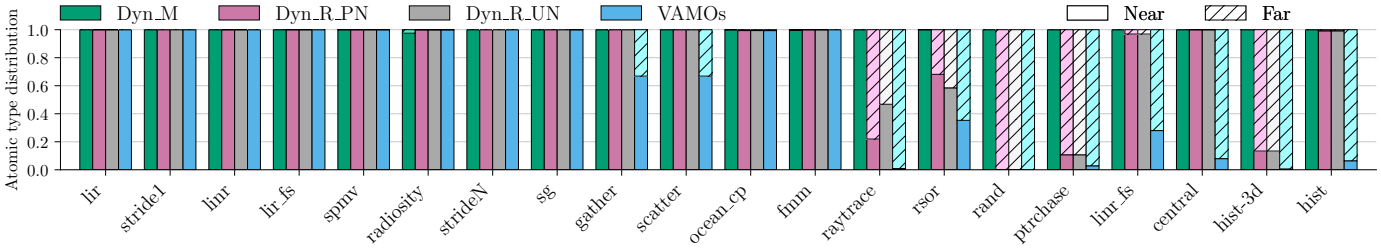


Fig. 9. AMOs distribution of the dynamic AMO placement policies

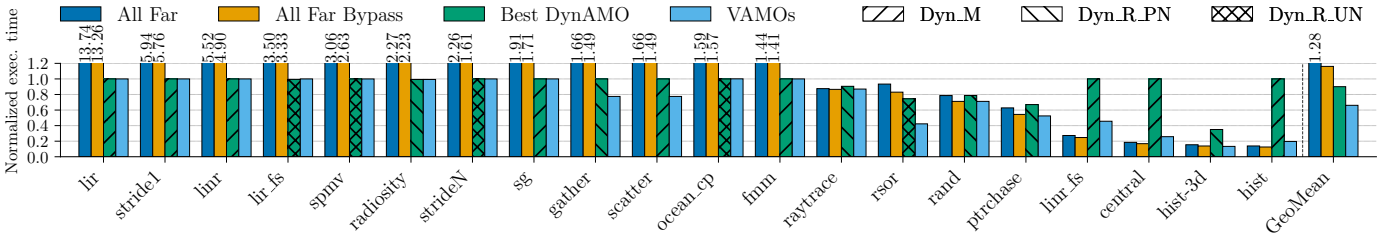


Fig. 10. Execution times of VAMOs compared with the best DynAMO variant for each benchmark, and All Far and All Far Bypass static policies. All execution times are normalized to All Near.

the same instruction within a single entry, and also leverage observed past behavior for the instruction on unseen cacheline addresses it accesses.

C. Performance impact

Fig. 10 reports the execution times for All Far, All Far Bypass, DynAMO, and VAMOs, normalized to the All Near static policy. For DynAMO, we select the variant that performs best for each benchmark (we mark its bars in the figure with the configuration used in each case). The All Far policy bars allow to discern if an application is near or far-friendly (near-friendly applications perform poorly with the All Far policy). It also enables a more precise assessment of the impact of bypassing private cache levels.

For near-friendly applications, VAMOs achieves performance comparable to All Near, without incurring any degradation due to mispredictions. In the *scatter* and *gather* benchmarks, the PC-based local prediction implemented in VAMOs enables anticipating absence of reuse in multiple cachelines accessed by the same AMO instruction, therefore resulting in reductions of around 20% in both cases.

As discussed in Sec. V-B, DynAMO behaves over-conservatively in most far-friendly benchmarks. Under high contention and short reuse distances (e.g., *linr_fs*, *central*, *lgi*, *hist*), it does not classify any AMOs as far. In contrast, VAMOs improves performance between 75% to 85% for *central* and *hist*, and 55% for *linr_fs*. For applications with longer reuse distances (*raytrace*, *rsor*, *hist-3d*), DynAMO-Reuse is able to identify contention and issues far AMOs, yielding performance gains of up to 9%, 25% and 65% for *raytrace*, *rsor* and *hist-3d*, respectively. Yet, VAMOs outperforms DynAMO-Reuse because it transitions more rapidly to far and bypasses the private caches, achieving improvements of 13%, 58% and 87% for *raytrace*, *rsor* and *hist-3d*.

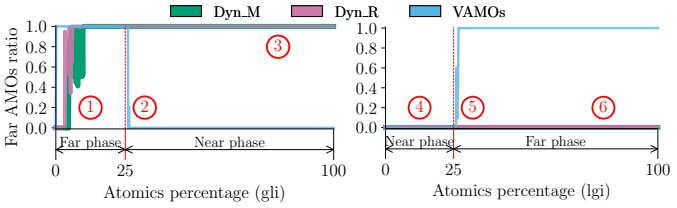


Fig. 11. Far AMOs ratio along *gli* and *lgi* executions

Finally, in applications that exhibit negligible data reuse (*rand*, *ptrchase*), DynAMO-Reuse matches the performance of the All Far configuration, with improvements of 22% and 33% for *rand* and *ptrchase*, respectively. Due to its bypassing mechanism, VAMOs matches the performance of the All Far Bypass configuration for these two applications, further increasing performance gains up to 29% and 48%, respectively.

DynAMO yields average performance improvements of 10% when the best-performing variant is selected for each benchmark, whereas VAMOs consistently attains average performance gains of 34%.

D. Adaptability of VAMOs

Finally, we evaluate the kernels whose atomic access patterns change dynamically though their execution: *gli*, *lgi*, *multi_addr* and *multi_call*. These kernels exhibit different access patterns through different execution phases. These different access patterns may be due to the atomic instruction accessing different memory addresses with different contention and locality behaviors, or due to a function containing an atomic instruction being called in different points in the code, which would resemble a synchronization library.

First, we consider *gli* and *lgi*. Fig. 11 illustrates the temporal evolution of the fraction of far AMOs issued by DynAMO-Metric, DynAMO-Reuse (note that DynAMO-Reuse-UN and

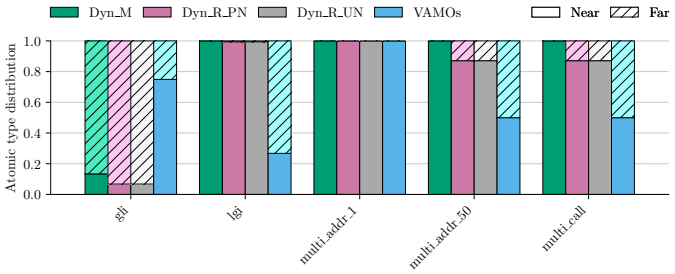


Fig. 12. AMOs distribution for the kernels with changing AMO patterns

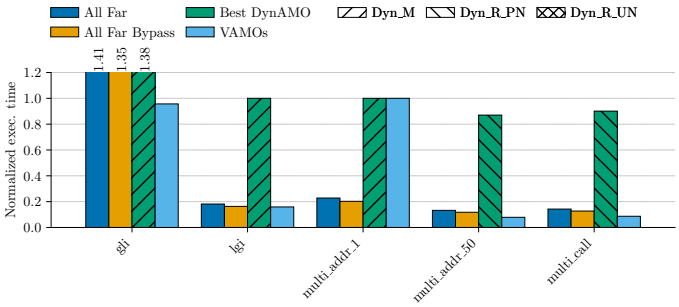


Fig. 13. Execution times for the kernels with changing AMO patterns

DynAMO-Reuse-PN exhibit very similar behavior), and VAMOs. The Y-axis reports the fraction of far AMOs issued in each time interval, while the X-axis shows the cumulative fraction of AMOs executed during the application.

In *gli*, all DynAMO variants adapt correctly during the initial global-increment phase ① and issue far AMOs, although they take slightly more time to adjust than VAMOs. When the application transitions to the local-increment phase ②, DynAMO remains biased towards far execution ③. DynAMO-Metric can return to near execution only if enough near AMOs are issued to reduce its metric below the threshold, whereas DynAMO-Reuse requires a cacheline to be reused after being inserted by a near AMO. Because the predictor remains in the far state after the phase transition, neither condition is satisfied: near AMOs are not issued, and cachelines are not reused under near execution. As a result, DynAMO becomes trapped in the far state and fails to recover (*forever far* phenomenon). VAMOs does not suffer from this limitation. It quickly detects that each cacheline is repeatedly accessed by the same node (via the FNDT), increases the corresponding NC, and rapidly transitions back to near AMOs for the remainder of the execution.

In *lgi*, the short reuse distance prevents DynAMO variants from adapting to the far phase ⑥. VAMOs, in contrast, tracks contention explicitly and prioritizes it over local reuse. It therefore transitions correctly from the initial near-friendly phase ④ to the subsequent far-friendly phase ⑤.

Now, we analyze the near/far atomic distribution and execution times of these two kernels (*gli* and *lgi*), as well as the *multi_addr* and *multi_call* kernels. For the *multi_addr* kernel, we evaluate two variants: one with a 1-1 interleaving (i.e., one near-friendly access, one far-friendly access), and another with a 50-50 interleaving (i.e., 50 near, 50 far).

Fig. 12 reports the distribution of AMOs in each case. First, for *gli* and *lgi*, we observe an expected 75%-25% distribution between near and far predictions, depending on the kernel. For the *multi_address_1* kernel, we observe that, as expected, no far AMOs are predicted by VAMOs. This is an extreme case in which, because near and far-friendly accesses are completely intermixed, the predictor is unable to detect the access pattern of the atomic instruction. Thus, it remains stuck at the initial near prediction.

The situation is different for the *multi_address_50* kernel. In this case, the near-friendly and far-friendly micro-phases have a length of 50 accesses. While both DynAMO-Metric and DynAMO-Reuse opt for a near prediction, VAMOs is able to adapt quickly to each micro-phase and predict the correct execution placement for each one, leading to the 50%-50% near-far distribution.

Finally, the *multi_call* kernel behaves similarly to the former. Despite the near and far-friendly calls being completely interleaved, they are located at different parts of the code. Thus, the PC⊕RAS indexing of the PFNT is essential for using different entries to classify each AMO execution separately. In this way, VAMOs can successfully capture the near-friendly and far-friendly behaviors of each one. Every near access is correctly predicted as near, and every far access is predicted as far, leading again to a 50%-50% near-far distribution for VAMOs.

Fig. 13 summarizes the execution times of these kernels, normalized to the All Near execution. VAMOs obtains the best performance in both *gli* and *lgi* by its high adaptability to each phase, while DynAMO is unable to detect the phase change in *gli* and remains in a *forever far* prediction, and do not detect the far-friendly phase in *lgi*. In *multi_address_1* neither DynAMO nor VAMOs is able to detect the far-friendly behavior of one of the two addresses (DynAMO because of the short reuse distance, and VAMOs because of the PC index of the PFNT), and obtain the same performance as the All Near execution. In this benchmark, interestingly, All Far and All Far Bypass obtain the best performance, despite half of the accesses being near-friendly. This is because the benefit of executing far the far-friendly accesses, and thus avoiding the high contention at the LLC, is higher than the performance detriment of executing far the near-friendly accesses.

Finally, in the *multi_address_50* and *multi_call* kernels, VAMOs is able to adapt properly to each phase and behavior of each call and predicts the correct AMO placement for each access. Contrarily, DynAMO-Metric is unable to detect the far-friendly address, and in DynAMO-Reuse, only a small subset of nodes detect this far-friendly address. VAMOs extracts the full advantage of near and far AMOs in these two kernels, obtaining the best performance on each one.

E. Ablation study

To assess the impact that the mechanisms implemented in VAMOs have on performance, we incrementally deactivate each one of them and quantify their individual contribution to

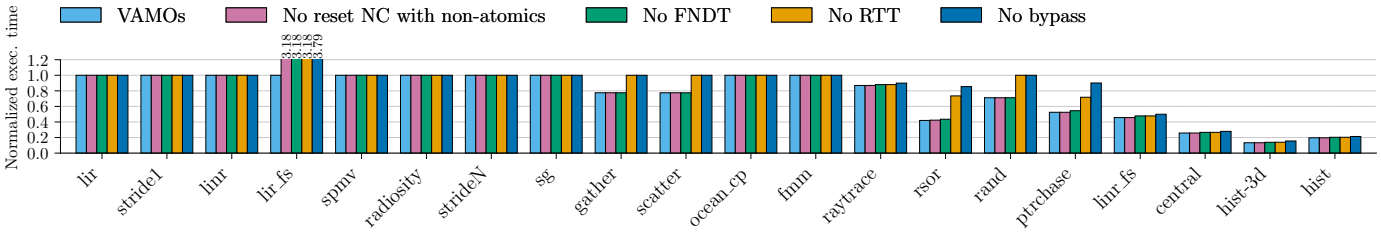


Fig. 14. Execution times of VAMOs and its restricted variants, normalized to All Near

overall performance. We progressively remove these components until obtaining a highly simplified variant of VAMOs that solely monitors hits of near AMOs in the L1D, and contention in the LLC. Fig. 14 reports the execution times for the full VAMOs design and for these progressively restricted variants. All execution times are normalized to the performance of the All Near policy.

We can examine how each component of VAMOs plays a crucial role in distinct benchmarks. The NC reset triggered by non-atomic accesses (Sec.III-C2) is critical for *lir_fs* to match the performance of All Near. The FNDT enables the transition from the far-friendly phase to the near-friendly phase in *gli* as shown in Sec. V-D. The RTT becomes critical to track non-reused data in *gather*, *scatter*, *rsor*, *rand* and *ptrchase*. Finally, the bypass of private cache levels provides extra performance gains in most of the far-friendly benchmarks.

VI. Related work

The academic literature has extensively investigated both hardware and software techniques aimed at reducing the cost of synchronization instructions and atomic updates. The majority of existing proposals rely on specialized hardware support, for example, dedicated accelerators or interconnection networks, or on the introduction of novel instructions to accelerate synchronization primitives [47]–[56]. However, these approaches typically incur substantial silicon area overheads, and most ISAs are reluctant to adopt additional, non-standard instructions.

Current research in many-core synchronization has looked at approaches that explicitly account for the physical and protocol-level constraints of on-chip interconnects. Delegato [31] introduces a delegation-based synchronization model specifically designed for multi-chiplet architectures, in which atomic operations are offloaded to dedicated owner entities to reduce cross-die traffic. Although this multi-chiplet orientation is interesting, it is largely orthogonal to the many-core focus of VAMOs.

At the microarchitectural level, prior work has concentrated on enhancing the speculative execution of the Read-Modify-Write (RMW) cycle. Free Atomics [57] and the No Rush (RoW) [58] scheduler refine the timing and fence constraints associated with atomic operations within the processor pipeline. These techniques remain fundamentally local or "near"-centric optimizations and therefore do not address the root cause of interconnect congestion arising from cache-line ping-ponging. In parallel, the expressiveness of atomic primitives has been extended through Multi-Address Atomic (MAD) operations [59], which provide atomicity guarantees across multiple, potentially non-contiguous memory locations.

VII. Conclusion

This work has proposed VAMOs, a lightweight yet highly accurate PC-based AMO prediction mechanism that dynamically determines the best way to execute each AMO: near or far. VAMOs is built on two key observations: (i) AMO behavior can be effectively characterized based on their program counter (PC), and (ii) global information exposed by the LLC can enhance the quality of near–far execution decisions. By leveraging these observations and bypassing private cache levels for far AMOs, VAMOs outperforms the state-of-the-art DynAMO predictor by 21.82% across a comprehensive suite of AMO-intensive benchmarks with varying behaviors. These results are achieved while reducing the storage requirements by 88.96% with respect to DynAMO.

References

[1] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2024, order Number: 325462-082US.

[2] Arm Holdings. (2022) Armv9 architecture reference manual for a-profile architecture. [Online, accessed 25-03-2026]. [Online]. Available: https://developer.arm.com/documentation/ddi0487/maa/?lang=en

[3] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[4] G. Project. (2023) Gcc __atomic builtins. [Online, accessed 25-03-2026]. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

[5] T. C. S. Committee. (2023) std::atomic library. [Online, accessed 25-03-2026]. [Online]. Available: https://en.cppreference.com/w/cpp/header/atomic.html

[6] V. Soria-Pardos, A. Armejach, T. Mück, D. Suárez-Gracia, J. Joao, A. Rico, and M. Moretó, "Dynamo: Improving parallelism through dynamic placement of atomic memory operations," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589065

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The mit alewife machine: architecture and performance," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 2–13. [Online]. Available: https://doi.org/10.1145/223982.223985

[8] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, p. 64–75, Apr. 1989. [Online]. Available: https://doi.org/10.1145/68182.68188

[9] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *High Performance Embedded Architectures and Compilers*, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 3–17.

[10] R. Kessler and J. Schwarzmeier, "Cray t3d: a new dimension for cray research," in *Digest of Papers. Compcon Spring*, 1993, pp. 176–182.

[11] J. Laudon and D. Lenoski, "The sgi origin: a ccnuma highly scalable server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 241–251. [Online]. Available: https://doi.org/10.1145/264107.264206

[12] S. L. Scott, "Synchronization and communication in the t3e multiprocessor," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII. New York, NY, USA: Association for Computing Machinery, 1996, p. 26–36. [Online]. Available: https://doi.org/10.1145/237090.237144

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[14] Arm Holdings. (2021) Amba 5 chi architecture specification. [Online, accessed 25-03-2026]. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba/amba-5

[15] M. Bruce, "Arm neoverse v2 platform: Leadership performance and power efficiency for next-generation cloud computing, ml and hpc workloads," in *2023 IEEE Hot Chips 35 Symposium (HCS)*, 2023, pp. 1–25.

[16] H. Wire. (2020) Aws graviton2-powered ec2 instances now available. [Online, accessed 25-03-2026]. [Online]. Available: https://www.hpcwire.com/2020/06/12/aws-graviton2-powered-ec2-instances/

[17] A. Holdings. (2020) Arm neoverse n2 core technical reference manual. [Online, accessed 25-03-2026]. [Online]. Available: https://developer.arm.com/documentation/102099/0001/The-Neoverse-N2--core

[18] ——. (2022) Do near or far atomics give the best performance on neoverse systems? [Online, accessed 25-03-2026]. [Online]. Available: https://developer.arm.com/documentation/ka004706/latest/

[19] Arm Holdings. (2019) Arm cortex-a77 core software optimization guide. [Online, accessed 25-03-2026]. [Online]. Available: https://developer.arm.com/documentation/swog011050/latest/

[20] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

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

[22] SiFive, Inc. (2023) Tilelink specification, version 1.9.3. [Online, accessed 25-03-2026]. [Online]. Available: https://www.sifive.com/document-file/tilelink-spec-1.9.3

[23] ARM Holdings. (2017) Amba axi and ace protocol specification. [Online, accessed 25-03-2026]. [Online]. Available: https://developer.arm.com/documentation/ihi0022/hc

[24] NVIDIA. (2025) Parallel thread execution isa. [Online, accessed 25-03-2026]. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[25] ——. (2026) Cuda programming guide, v13.2. [Online, accessed 25-03-2026]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/

[26] ——. (2022) Nvidia hopper architecture in-depth. [Online, accessed 25-03-2026]. [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[27] AMD ROCm Documentation. (2025) Hardware atomics operation support. [Online, accessed 25-03-2026]. [Online]. Available: https://rocm.docs.amd.com/en/latest/reference/gpu-atomics-operation.html

[28] M. Elteir, H. Lin, and W.-C. Feng, "Performance characterization and optimization of atomic operations on amd gpus," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, ser. CLUSTER '11. USA: IEEE Computer Society, 2011, p. 234–243. [Online]. Available: https://doi.org/10.1109/CLUSTER.2011.34

[29] V. Soria-Pardos, A. Armejach, D. Suárez, D. Martinot, A. Grasset, and M. Moretó, "Flama: Architecting floating-point atomic memory operations for heterogeneous hpc systems," in *2025 28th Euromicro Conference on Digital System Design (DSD)*, 2025, pp. 435–442.

[30] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[31] V. Soria-Pardos, A. Armejach, T. Mück, D. S. Gracia, J. Joao, and M. Moretó, "Delegato: Locality-aware atomic memory operations on chiplets," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1793–1808. [Online]. Available: https://doi.org/10.1145/3725843.3756030

[32] A.-T. Nguyen and J. Torrellas, "Design trade-offs in high-throughput coherence controllers," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003, pp. 194–205.

[33] RISC-V Software Computing Organization, "Spike risc-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim, 2026, [Online, accessed 25-03-2026].

[34] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[35] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 33–42.

[36] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel alder lake cpu architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[38] J. Burkardt. (2021) Png files florida state university. [Online, accessed 25-03-2026]. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/data/png/bmp_24.png

[39] NASA. (2021) Nasa video and image library. [Online, accessed 25-03-2026]. [Online]. Available: https://images.nasa.gov/

[40] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[41] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Splash-4: A modern benchmark suite with lock-free constructs," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*. Austin, TX (USA): IEEE Computer Society, Nov. 2022, pp. 51–64. [Online]. Available: http://webs.um.es/aros/papers/pdfs/ejgomez-iiswc22.pdf

[42] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[43] B. Williams, J. Leidel, X. Wang, D. Donofrio, and Y. Chen, "Circustent: A benchmark suite for atomic memory operations," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 144–157. [Online]. Available: https://doi.org/10.1145/3422575.3422789

[44] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008. [Online]. Available: https://books.google.es/books?id=seAgiOfu2EIC

[45] A. Sohn and Y. Kodama, "Load balanced parallel radix sort," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 305–312. [Online]. Available: https://doi.org/10.1145/277830.277903

[46] G. Zhang, W. Horn, and D. Sanchez, "Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–25. [Online]. Available: https://doi.org/10.1145/2830772.2830774

[47] A. Ros and S. Kaxiras, "Callback: efficient synchronization without invalidation with a directory just for spin-waiting," *SIGARCH Comput. Archit. News*, vol. 43, no. 3S, p. 427–438, Jun. 2015. [Online]. Available: https://doi.org/10.1145/2872887.2750405

[48] S. Abadal, A. Cabellos-Aparicio, E. Alarcon, and J. Torrellas, "Wisync: An architecture for fast synchronization through on-chip wireless communication," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3–17. [Online]. Available: https://doi.org/10.1145/2872362.2872396

[49] B. E. Akgul and V. J. Mooney, "The system-on-a-chip lock cache," *Design Automation for Embedded Systems*, vol. 7, no. 1–2, p. 139–174, Sep. 2002. [Online]. Available: https://doi.org/10.1023/A:1019751632622

[50] A. Franques, A. Kokolis, S. Abadal, V. Fernando, S. Misailovic, and J. Torrellas, "Widir: A wireless-enabled directory cache coherence protocol," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 304–317.

[51] A. Kurth, S. Riedel, F. Zaruba, T. Hoefler, and L. Benini, "Atuns: Modular and scalable support for atomic operations in a shared memory multiprocessor," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[52] C.-K. Liang and M. Prvulovic, "Misar: Minimalistic synchronization accelerator with resource overflow management," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 414–426.

[53] J. T. Robinson, "A fast general-purpose hardware synchronization mechanism," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 122–130. [Online]. Available: https://doi.org/10.1145/318898.318910

[54] X. Tang, J. Zhai, X. Qian, and W. Chen, "plock: A fast lock for architectures with explicit inter-core message passing," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 765–778. [Online]. Available: https://doi.org/10.1145/3297858.3304030

[55] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero, "Architectural support for fair reader-writer locking," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. USA: IEEE Computer Society, 2010, p. 275–286. [Online]. Available: https://doi.org/10.1109/MICRO.2010.12

[56] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 35–45, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273440.1250668

[57] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free Atomics: Hardware Atomic Operations Without Fences," in *ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. Proceedings, A. . IEEE, Ed. New-York, United States: ACM IEEE, 2022, pp. 1–13.

[58] A. Asgharzadeh, J. Feliu, M. E. Acacio, S. Kaxiras, and A. Ros, "No rush in executing atomic instructions," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 1618–1630.

[59] E. J. Gómez-Hernández, J. M. Cebrian, R. Titos-Gil, S. Kaxiras, and A. Ros, "Efficient, distributed, and non-speculative multi-address atomic operations," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 337–349. [Online]. Available: https://doi.org/10.1145/3466752.3480073