DynAMO: Improving Parallelism Through Dynamic Placement of Atomic Memory Operations

Víctor Soria-Pardos
victor.soria@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Adrià Armejach
Barcelona Supercomputing Center,
Universitat Politècnica de Catalunya
Barcelona, Spain

Tiago Mück
Arm
Austin, Texas, USA

Darío Suárez Gracia
Universidad de Zaragoza
Zaragoza, Spain

José A. Joao
Arm
Austin, Texas, USA

Alejandro Rico
AMD
Austin, Texas, USA

Miquel Moretó
Universitat Politècnica de Catalunya,
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

With increasing core counts in modern multi-core designs, the overhead of synchronization jeopardizes the scalability and efficiency of parallel applications. To mitigate these overheads, modern cache-coherent protocols offer support for Atomic Memory Operations (AMOs) that can be executed near-core (*near*) or remotely in the on-chip memory hierarchy (*far*).

This paper evaluates current available static AMO execution policies implemented in multi-core Systems-on-Chip (SoC) designs, which select AMOs' execution placement (*near* or *far*) based on the cache block coherence state. We propose three static policies and show that the performance of static policies is application dependent. Moreover, we show that one of our proposed static policies outperforms currently available implementations.

Furthermore, we propose DynAMO, a predictor that selects the best location to execute the AMOs. DynAMO identifies the different locality patterns to make informed decisions, improving AMO latency and increasing overall throughput. DynAMO outperforms the best-performing static policy and provides geometric mean speedups of 1.09× across all workloads and 1.31× on AMO-intensive applications with respect to executing all AMOs *near*.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

KEYWORDS

multi-core architectures, microarchitecture, atomic memory operations, data placement

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589065

ACM Reference Format:
Víctor Soria-Pardos, Adrià Armejach, Tiago Mück, Darío Suárez Gracia, José A. Joao, Alejandro Rico, and Miquel Moretó. 2023. DynAMO: Improving Parallelism Through Dynamic Placement of Atomic Memory Operations. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589065

1 INTRODUCTION

Multi-core designs have dominated the CPU market since the end of Dennard's scaling [21]. Despite chip manufacturers getting closer to the end of Moore's Law [49], recent designs continue to deliver higher core counts. For instance, AMD's EPYC 7773X uses a chiplet-based architecture and 3D stacking to support up to 64 cores and a 768MiB Last-Level Cache (LLC) [15]. Other vendors have released even higher core counts, such as the 128-core Ampere Altra Max [26]. This trend towards many-core architectures makes synchronization operations critical for scaling the performance of parallel applications: a larger core count leads to longer communication latency between cores and more contention in the shared system interconnect.

Shared memory is the preferred programming model to parallelize applications in multi-core architectures. In this paradigm, parallel applications enforce correctness through explicit synchronization primitives, such as locks, barriers, and condition variables. These synchronization primitives use expensive atomic Read-Modify-Write (RMW) operations to update shared variables.

Modern Instruction Set Architectures (ISAs), such as x86, POWER, Armv9, and RISC-V, implement RMW operations as a single Atomic Memory Operation (AMO) instruction. AMOs can perform simple operations such as bit-wise arithmetic (OR, AND, XOR), fetch-and-add, swap, and Compare-And-Swap (CAS). Programmers can directly use AMO through compiler *built-in* macros [54] or using specific libraries [18]. Typically, on an AMO, the core fetches the target cache block with exclusive permissions into its private cache to modify it locally (*near* AMOs). Alternatively, systems have implemented mechanisms to execute AMOs on different locations on the chip (*far* AMOs); for example: on the directory, the LLC, the

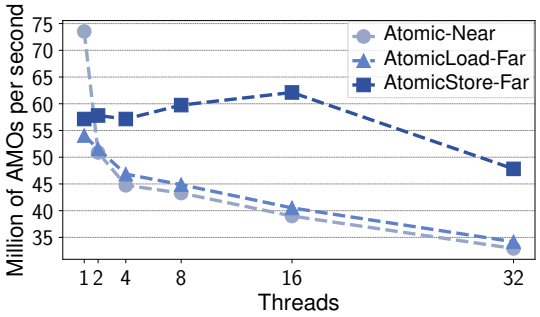

Figure 1: Throughput of *near* and *far* AMOs. Far AMOs are split into two sets: AtomicLoads are AMOs that return the previously stored value, AtomicStores do not return any value.

memory controller, or dedicated modules [5, 30, 33, 38, 42, 58, 63]. Recent processors [36, 43], GPUs [64], and the Advanced Microcontroller Bus Architecture (AMBA) 5 Coherent Hub Interface (CHI) standard [9] have reintroduced the concept of *far* AMOs, including mechanisms to support them.

Figure 1 illustrates that *near* and *far* AMOs are complementary mechanisms comparing the throughput of a memory update using *near* and *far* AMOs on an AMBA 5 CHI based system (see Section 6.1 for simulation details). In the case of *far* AMO, AMBA 5 CHI distinguishes two versions depending on whether the AMO transaction returns the previously stored value (AtomicLoad-Far) or not (AtomicStore-Far).

When near AMOs, Atomic-Near, run in single-threaded programs, they achieve the highest possible throughput, since their cache blocks are always present in the first-level cache. However, as the thread count increases, we observe how the throughput gradually degrades as threads compete to update the shared variable. When switching to far AtomicLoad and AtomicStore, the throughput of single-thread significantly reduces as AMOs execute in the shared LLC, increasing the latency of AMOs. However, as the thread count increases, the trend reverses specially for AtomicStore transactions (56% better than Atomic-Near for 32 cores), as the data does not move to private caches, and all the updates are centralized and serialized by the directory. Figure 1 reveals that *near* and *far* AMOs perform differently depending on contention and the memory access pattern.

This paper proposes and evaluates different static AMO placement policies that decide whether to execute the AMO *near* (L1D) or *far* (LLC) based on the current L1D cache block state. Our analysis and evaluation conclude that, while some policies best suit specific applications, there is no one-size-fits-all static policy. Therefore, we propose DynAMO, a simple predictor that dynamically decides the best location to execute the AMO (*near* or *far*). DynAMO identifies the different locality patterns of data accessed by AMOs to make informed decisions, improving AMO latency and increasing overall throughput.

This paper makes the following contributions:

- We investigate the design space of static AMO policies and propose three new static policies that target common access patterns and are not implemented in current chip designs.
- We implement and evaluate the proposed static policies as well as two state-of-the-art static AMO policies, available in current chips, using a detailed AMBA CHI model within the gem5 simulator [48]. We find out that one of our proposed static policies is the best performing, yielding speed-ups of 1.05× across all applications and 1.19× on AMO-intensive applications with respect to executing all AMOs *near*.
- We observe that there is no static policy able to outperform the rest on all applications. Therefore, we propose DynAMO, a dynamic AMO placement predictor that leverages data access locality patterns to decide whether AMOs should be executed *near* or *far*. We show that DynAMO can achieve better performance than the best static policy. DynAMO obtains speed-ups of 1.09× across all applications and of 1.31× for AMO-intensive applications with respect to executing all AMOs *near*.

2 BACKGROUND

2.1 Near and Far AMO Architectures

ISA specifications usually do not define where AMOs execute in the micro-architecture, *near* the core in the L1D cache or *far* in remote components such as the shared LLC or the memory controller, among others, where a small arithmetic logic unit performs the operation. *Near* AMOs implementations are typical due to their simple design and lower implementation and verification costs. However, *far* AMOs have been widely studied as one of the first examples of *Near Data Processing*. Rather than fetching cache blocks into private levels of cache to be updated, operations are sent to where the data resides, typically the LLC or a memory controller. Hence, this mechanism prevents the cache block from "ping-ponging" between cores, reduces the number of snoops issued by the directory, and may improve overall performance. While many systems only implement *near* AMOs, very few systems implement only *far* AMOs. Most were classic commercial systems like the NYU Ultracomputer [31], the Cray T3D [38], the T3E [58], or the SGI Origin [42]. Modern multi-core architectures have recently incorporated support for both *near* and *far* AMO execution, such as POWER9 [43] or Armv8 [34] through the AMBA 5 CHI standard [9] for cache coherent Networks-on-Chips (NoCs).

Several works have widely discussed the implementation of *near* and *far* AMOs [10, 24, 41]. Other works have tried to improve the performance of *far* AMOs by (i) reducing the latency between consumer and producer through data forwarding [24, 68], or (ii) speeding-up lock exchange [3, 20]. However, most of these proposals rely on specific communication NoCs or memory consistency models (e.g., Total Store Ordering), cause significant extra traffic, or use simple trace-based models to simulate AMOs. We further discuss other prior works in Section 7.

2.2 The CHI Cache Coherence Protocol

The AMBA 5 CHI is an open specification [9] for shared system interconnects targeting large-scale systems with a weak consistency model. CHI supports both *near* and *far* AMO transactions and is used on recent servers and cloud-grade systems-on-chip [7, 26, 36]. CHI implements a tunable and modular MOESI cache coherence protocol. CHI uses a different name convention from classic MOESI: Unique Clean (UC, Exclusive), Unique Dirty (UD, Modified), Shared

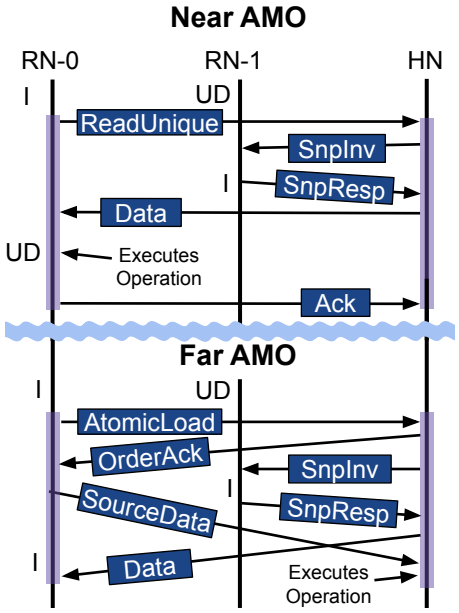

Figure 2: CHI *near* and *far* transaction specifications. The Request Nodes (RNs) encapsulate the cores and private caches. The Home Node (HN) is the point-of-coherence for the cache block and contains the directory and LLC.

Clean (SC, Shared), Shared Dirty (SD, Owned), and Invalid (I, Invalid). The protocol specifies multiple transactions with different message flows and semantics for each possible operation an agent in the system can perform. For instance, a cache controller can select which of these transactions is issued depending on the cache level, the state of the cache block, the type of operation, and the cache inclusion policy, among other parameters.

In the case of AMOs, the AMBA CHI specification states that an AMO can execute through two types of transactions, which we refer to as *near* and *far*. Figure 2 defines the message flow of these transactions, which we have implemented in gem5. A *near* AMO performed by a request node (RN-0) generates a ReadUnique transaction to gain exclusiveness of the cache block and perform the atomic operation locally. The directory, or home node (HN), is the point of coherence that issues the required snoops to invalidate other local copies of the cache block (e.g., the invalidation of RN-1's local copy in Figure 2). Finally, the HN answers by sending the cache block to the L1D cache of RN-0 in either UniqueClean or UniqueDirty state, where the AMO executes. If the block is already in Unique state at the L1D cache, the AMO is directly performed locally without issuing any transaction (not shown in Figure 2).

A *far* AMO always issues an atomic transaction to the HN. As in *near* AMOs, the HN must ensure all local copies are invalidated before modifying the block by sending snoops. In parallel to the snoop messages, the HN requests the RN-0 L1D cache controller to forward the data operands required to perform the atomic operation. Once all copies are invalidated, the HN answers with the original value stored in the accessed memory position (AtomicLoad) or by sending an acknowledgment message that ends the transaction (AtomicStore) before performing the AMO. Finally, once the HN receives the source operands, the atomic operation is performed in the HN.

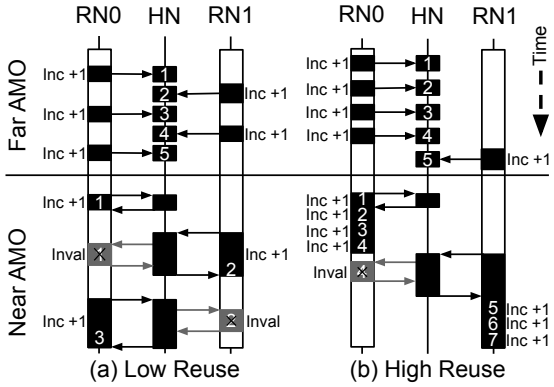

Figure 3: Simplified diagram that shows (a) low and (b) high reuse access patterns for *far* and *near* AMO executions.

There is a pathological case for *far* AMOs when the requestor has the targeted cache block in the L1D cache in UC or UD state. In this case, the HN must send a snoop message to the RN-0 L1D cache (the requestor). This snoop message increases the critical path of the flow and delays its completion, making *far* AMO impractical. Therefore, whenever the L1D cache has the block in UC/UD state, the preferred choice should always be to perform a *near* AMO.

Currently, existing processors implementing AMBA CHI usually execute AMOs *near* in the L1D cache. More recently, Arm's Neoverse CPUs support executing AMOs *near* or *far* when connected to a CMN-600 or CMN-700 mesh interconnect network. The configuration mode of AMO's can be changed through a machine register [8]. However, Arm recommends configuring AMOs to execute always *near* due to different pitfalls [35] that we explain in the next section.

3 MOTIVATION

This section elaborates on the possible use cases of *near* and *far* AMOs. They are presented as a possible solution to mitigate the growing synchronization bottleneck that shared memory multicores suffer. We also explain the main pitfalls that may limit the performance of AMOs.

3.1 Near and Far AMOs Use Cases

As shown in Figure 1, *near* and *far* AMOs can alternatively achieve the highest throughput in different situations. Figure 3 presents a simplified message diagram, omitting some messages, to illustrate two different access patterns. In this example, two threads on different cores or request nodes (RN0 and RN1) increment a shared variable through AtomicStore instructions. In access pattern (a), threads perform an AMO on the shared counter in turns. In this scenario, *far* AMOs executed at the home node (HN) perform better than *near* AMOs because they avoid the ping-pong effect of the cache block caused by the highly contended variable. Section 6.2 will show how benchmarks like GMETIS and RADIOSITY have this access pattern.

Meanwhile, in access pattern (b), each thread performs four AMOs on the shared counter, exhibiting locality. In this case, *near*

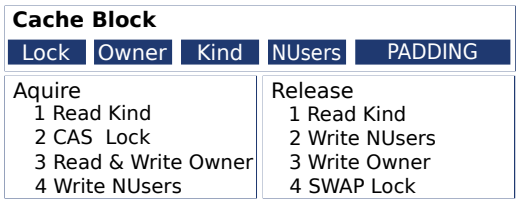

Figure 4: Top: *Pthread Mutex* data members distribution within a cache block. Bottom: *Pthread Mutex* operations performed to acquire and release the lock.

AMOs perform better, as bringing the cache block into the L1D is a better choice than paying the cost of *far* AMOs for each access. This access pattern is present in several applications, such as FLUIDANIMATE and SPT.

The main takeaway is that the decision to execute AMOs *near* or *far* impacts performance and depends on different application and system-level factors such as AMO locality and the memory access pattern. To maximize performance, computer architects must implement dynamic mechanisms to select the best placement for a given AMO.

3.2 Far AMO Pitfalls and Best Practices

Even though *far* AMOs aim to reduce on-chip data movement and improve system's overall throughput, their use in specific scenarios can cause significant global traffic and may increase the execution latency of AMOs. For example, forcing all AMOs to execute *far* in single-thread programs may prevent some variables from being fetched into private caches, increasing AMO access latency as a result. Additionally, other scenarios can trigger pathological behaviors that harm the overall system performance. We now discuss the main pitfalls and bottlenecks of AMOs, including solutions that can be adopted to mitigate some of these issues.

3.2.1 Consistency Model. Cores usually issue AMOs at *commit* stage to preserve memory ordering and sequential program semantics. AMOs are guaranteed to complete once sent to the memory subsystem. Moreover, AMOs that have both read and write (RMW) semantics, e.g., fetch-and-add (AtomicLoad), return the read value stored in the memory position before the update, which delays the completion of the AMO, potentially stalling the pipeline. These restrictions make *near* AMOs that miss in the L1D cache, and *far* AMOs in general, quite expensive, as they can block the commit stage for multiple cycles. On top of that, branches are the most frequent instructions that consume the return value of an AMO, exacerbating the problem.

The core can speculatively issue a read before the AMO arrives at the *commit* stage to mitigate the long latency of AMOs with a return value. However, since AMOs are typically used to implement synchronization primitives and update shared variables, the speculative read will likely be invalidated, thereby increasing the number of messages and invalidations. Therefore, computer architects must apply this optimization judiciously.

Several ISAs, such as RISC-V, Armv9, and Power9, have included atomic-no-return instructions, also known as *AtomicStore*. These instructions read-modify-write without returning the original read value to the requestor, that just needs a data-less acknowledge message to complete the transaction. These instructions have lower consistency constraints, allowing cores to commit AMOs earlier. Therefore, replacing AtomicLoads for AtomicStores when possible is critical to improve performance, especially for *far* AMOs. However, while AtomicLoads are supported and generated by most compilers and libraries, we have observed that AtomicStores are typically not generated unless explicitly expressed by the programmer via inline assembly. Most compilers do not even have built-in macros yet for AtomicStores. Thus, we have manually replaced AtomicLoads for AtomicStores whenever possible in the benchmarks used in Section 6.1 to take into account this insight.

3.2.2 Home Node Throughput. *Far* AMOs centralize the execution of synchronization instructions at the HN level to deliver high throughput, thereby increasing parallelism in contended scenarios. However, specific implementation details can hinder the throughput of *far* AMOs.

When an AMO request arrives at the corresponding HN slice and hits in the LLC, the cache controller reads the data SRAM cells that store the cache block to obtain the stored value. Then, an Arithmetic Logic Unit performs the AMO over the data. Finally, the result is written back into the SRAM cells. Accessing the LLC data arrays can take tens of cycles, as LLCs are optimized for storage rather than latency. Fortunately, cache blocks that receive AMOs are likely to experience temporal (mutex) or spatial (reductions) locality. Therefore, this bottleneck can be alleviated by having a small dedicated buffer to store the cache blocks that AMOs recently accessed, avoiding high-latency reads and writes into the SRAM cells and improving overall *far* AMO throughput. Thus, we adopt this approach as the baseline implementation for *far* AMOs.

3.2.3 Software Stack Support. Modern runtimes, libraries, and software stacks have been developed and optimized for several years for multiple systems with different synchronization instructions. Consequently, we have modular software that works on a wide range of systems, but does not always consider the particular characteristics of each machine.

The *Pthread Mutex* is one of the most used synchronization primitives by programmers. A simple analysis of the *Pthread Mutex* specification reveals that it is not ready to work with *far* AMOs in an efficient manner. Figure 4 shows the data members in blue that compose the *Pthread Mutex* struct, their distribution within a cache block, and how they are accessed to acquire and release the mutex. In both routines, we observe several read and write operations to elements placed on the same cache block. To understand how *near* and *far* AMOs interact with the rest of the memory accesses, we analyze these routines step by step.

In the case of *lock acquire*, the *Kind* field is read to determine the implementation version of the *Pthread Mutex*. This read operation fetches the cache block with read-only permissions. Then, a *far* AMO is issued to acquire the mutex modifying the *Lock* variable if free, invalidating the local copy. If the thread acquires the mutex, then it overwrites the fields *Owner* and *NUsers*, fetching the block again into the L1D cache, now with write permissions. In contrast, if the thread performs a *near* AMO, it fetches the block with read and write permissions from the beginning. Hence, the subsequent read and write operations would hit the L1D cache.

Table 1: Existing and proposed static AMO policies based on current L1D cache block state. Near (N), Far (F).

	Policy Name	UC	UD	SC	SD	I
Existing	All Near	N	N	N	N	N
	Unique Near	N	N	F	F	F
Proposed	Present Near	N	N	N	N	F
	Dirty Near	N	N	F	N	F
	Shared Far	N	N	F	F	N

For *lock release*, the *Kind* field is again read. Then two write operations are performed to *Owner* and *NUsers*. These two instructions fetch the cache block into the L1D cache and invalidate all the other copies. Then a SWAP instruction releases the *Lock* variable. As we have mentioned in Section 2.2, when the cache block is already present with exclusive permissions at the L1D cache, then the AMO should always be performed as a *near* AMO to avoid the extra cost of writing back the block and issuing a *far* AMO.

We can conclude that performing *near* AMOs in *Pthread Mutex* routines will consistently outperform *far* AMOs. In Section 6, we will see how this expected behavior matches the results of our experiments. Consequently, a new version of the *Pthread Mutex* struct should be devised so that *far* AMOs can match the performance of *near* AMOs. Simple approaches such as splitting the *Pthread Mutex* data members into different cache blocks may degrade the performance of *near* AMOs. Therefore, a different interface layout may need to be used, as current implementations hinder the performance of AMOs that execute remotely. However, this additional work is out of the scope of this research and we keep the default *Pthread Mutex* configuration.

4 STATIC AMO POLICIES

Section 2.2 describes the implementation of *near* and *far* AMOs in the AMBA 5 CHI protocol. However, the protocol does not enforce when the cache controller must issue a *near* or a *far* AMO – being implementation dependent. Some implementations, such as the Neoverse N1 [8], allow execution of AMOs to be configured depending on the cache block's state. However, only a limited number of policies are supported. We have performed a Design Space Exploration to cover a broader number of AMO policies. The controller can issue any of the two AMO transactions for each of the five cache block states in the MOESI coherence protocol. Thus, we can derive up to 32 different static AMO policy implementations. However, many of these static policies are very similar or do not make sense from a practical point of view. They may trigger pathological cases that add significant serialization; e.g., performing a *far* AMO on blocks already in unique state in the L1D cache (UC and UD states). As a result, the number of possible static AMO policies is reduced to just eight options. From these options, we select the five most representative implementations. We omit the remaining three policies because they show very similar performance.

Table 1 defines the five selected policies. For each policy, the table shows where the AMO is executed (*near* (N) or *far* (F)) for each L1D cache block state. We refer to these policies as *static AMO policies* as they always perform the same decision based only on the cache block coherence state. We have split the static AMO policies into two groups. The first one is that of existing implemented policies, and includes the two policies found in the Neoverse core family. The second group contains three static AMO policies we propose and evaluate in this paper. The next paragraphs describe each policy in detail, highlighting their rationale. Section 6.2 evaluates their performance.

The first existing static policy is *All Near*. It executes all AMOs in the L1D cache (*near* to the CPU). The *All Near* policy is the default AMO policy for those multi-core SoCs that do not have support for *far* AMOs, as well as for Arm's Neoverse CPUs, which support *far* AMO execution [8]. *All Near* performs well in scenarios where cache blocks have temporal locality and is worth fetching them into the L1D cache.

On the other extreme of the design space, we have the *Unique Near* policy. For *I*, *SC*, and *SD* cache block states, *Unique Near* issues an atomic transaction to execute the AMO *far* in the HN. As a result, the system avoids bringing the cache block into the L1D in *Unique* state and prevents future invalidations. If the cache block is already present in the L1 cache in *Unique* state (i.e., with exclusive permissions), it executes the AMO locally. The *Unique Near* static AMO policy is also available in Arm's Neoverse CPUs, although it attains lower performance than *All Near* [35]. Section 6.2 validates that this is the case for the benchmarks evaluated in this paper.

Between *All Near* and *Unique Near*, we consider three policies: *Present Near*, *Shared Far*, and *Dirty Near*. *Present Near* is similar to *All Near* as it performs all AMOs at the L1D cache except for those blocks that are not present. If the cache block is invalid, *Present Near* issues an atomic transaction to execute the AMO *far*. The rationale behind this policy is that there is some locality if the block is present, and it is worth upgrading the privileges to perform the AMO *near*. Otherwise, if the block is invalid, it may have been invalidated by the HN. Thus the core may be competing with other cores to update the block.

Shared Far executes AMOs *near* at the L1D cache for UC, UD, and I states. As its name suggests, it issues *far* AMOs only for shared states (SC and SD). This policy assumes that if the core shares the block with other cores, they may use it in the future. Therefore, executing the AMOs in the HN avoids future invalidations as other sharers reread the data block. Moreover, *Shared Far* fetches the targeted cache block if not present in the L1D because it could have been evicted from L1 to L2, and the best choice is to bring it back into the L1 cache.

Finally, *Dirty Near* executes *near* AMOs for UC, UD, and SD states. If the block is in SD state, the L1 cache was the last writer of the shared cache block. If the program has a consumer-producer pattern, it may be the next writer in the future and benefit from executing the AMO *near*.

Each of the introduced static policies presents advantages in different scenarios. For example, policies favoring *near* AMOs will perform better in applications with data locality. Meanwhile, in highly contended scenarios, *far* AMOs can increase AMO throughput execution. However, current systems rely on static policies, neglecting the need to adapt to each situation. Therefore, mechanisms that enable dynamic decisions based on system information can further improve AMOs' performance.

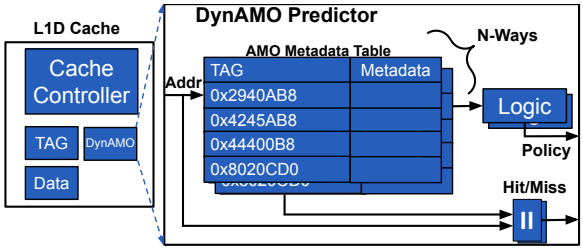

Figure 5: DynAMO predictor structure overview.

5 DYNAMO: DYNAMIC AMO POLICY

Next, we propose DynAMO, an AMO policy guided by a predictor that dynamically decides which policy to follow to execute AMOs. DynAMO aims to improve the latency and overall system throughput of AMOs while involving a small area and power overhead. To simplify our predictor, we have followed two main design principles: (i) the predictor only uses locally gathered information, and (ii) it has reduced hardware combinational and sequential logic.

Deciding whether an AMO should execute *near* or *far* is done at the L1D cache. However, AMOs usually update shared variables; therefore, core-local information can be insufficient to make accurate predictions. Global information about a cache block is only available at the HN, including the list of sharers and the current owner of the cache block. Forwarding this information to the private cache levels may be expensive and inaccurate. However, the L1D cache receives invalidation messages, which can partially replace the global view of the HN for our predictor, reducing its implementation cost. L1D caches can know through these invalidations if another core has requested a block for reading (downgrade to Shared) or writing (invalidation). Therefore, we have placed one DynAMO predictor at each L1D cache that uses these events to select the best AMO policy.

Figure 5 shows DynAMO's prediction mechanisms consisting of a look-up table that stores the predictor metadata and some logic that interprets the metadata to compute the prediction. To keep the predictor up-to-date, the Cache Controller registers the events in the L1D (AMOs, Snoops, Evictions, or Access Hits). Note that notifying these events is not in the critical path of L1D accesses. The cache controller queries the predictor when the AMO cannot be directly executed in the L1D (i.e., the cache block is not in Unique State) to decide the best static AMO policy to apply to the AMO. The AMO Metadata Table (AMT) is a set-associative look-up table with a few entries per way that stores the metadata associated with each cache block touched by an AMO. The AMT is indexed with the least significant bits of the physical cache block address and uses the most significant bits to check whether the indexed entry corresponds with the indexing address.

5.1 DynAMO Design Insights

An AMO, like any other memory access, follows common memory access patterns, such as *recency-friendly*, *thrashing*, *streaming*, or *mixed* [44]. The most adopted cache replacement policy is Least Recently Used (LRU), which mainly benefits the recency-friendly access pattern. LRU keeps those cache blocks likely to be reused in the cache. However, for *mixed* access patterns, LRU may evict blocks frequently referenced to insert other blocks used only once. *Near* AMOs, like LRU, work well for *recency-friendly* patterns because, in case of a miss, they fetch the cache block into the L1D cache, assuming it will be reused in the future. However, for other access patterns, *near* AMOs may evict frequently referenced cache blocks of the working set from the L1D cache to fetch blocks used only once. In those cases, issuing a *far* AMO avoids polluting the L1D by executing the AMO at the HN.

Besides the mentioned access patterns, parallel applications also exhibit different sharing patterns. Shared variables can be updated by multiple cores, triggering the snooping mechanism of the HN to forward the exclusivity rights from one core to another or invalidating all the read-only copies shared between multiple cores. To correctly understand the access patterns of a cache block, it is necessary to consider the access patterns of other cores. The most common sharing patterns are consumer-producer, migratory, and false-sharing. For these sharing patterns, MOESI-like cache coherence protocols may be sub-optimal [11] because they invalidate copies of other cores rather than updating them. *Near* AMOs behave like regular write operations in a MOESI cache coherence protocol so that they can suffer the same pathologies, such as cache block ping-ponging. In these cases, *far* AMOs can avoid the ping-pong effect by centralizing the updates at the HN.

5.2 Metric-Based DynAMO

The first DynAMO metric-based design predicts the placement for the AMO based on cache block statistics. The predictor counts the number of times the accessed cache block has been used to complete a *near* AMO and the number of times the directory has invalidated that same cache block. The predictor computes a ratio with these two values; a high ratio implies low contention on the cache block, i.e., more completed *near* AMOs than invalidations received. If the ratio exceeds a threshold, metric-based DynAMO predicts that future AMOs for that cache block should be executed *near*. Otherwise, the predictor selects *far* AMO execution, thus avoiding common invalidations and centralizing the execution of AMOs for that cache block at the HN.

As explained in Section 2.2, when the cache block is already present in the L1D cache in *unique* state, forcing the cache controller to perform a *far* AMO is more expensive than just executing the AMO locally. Hence, when the cache block is already in a *unique* state in the L1D, the AMO is always executed *near*. The predictor only selects between near and far execution for *I*, *SC*, and *SD* states. Since the prediction is made without distinguishing the cache state, when a cache block is predicted as *near* it behaves like *All Near* policy. When the cache block is predicted as *far* it behaves like *Unique Near* policy. We could make predictions for each one of the cache states, but that would increase the number of counters and the complexity of the predictor.

When a new entry is allocated in the AMT, the predictor must perform the first prediction without past history information. Based on the results obtained in Section 6.2, we know that *near* AMOs perform well in most cases. So, the first prediction is always a *near* AMO. The new AMT entry is initialized so the *near* AMO counter equals one and the invalidation counter equals zero. The main drawback of this design is that counters increment monotonically, so at some point, they can overflow. Moreover, in a short period

Table 2: gem5 configuration.

Processor	
Core count	32 out-of-order cores
Dispatch, issue width	14 insts/cycle
Fetch, decode width	8 insts/cycle
Commit width	8 insts/cycle
Reorder buffer	316 entries
Load and store queues	76 entries, 58 entries
Cache Memory Hierarchy	
Private L1 I&D caches	64 KiB/core, 4-way, 2 cycle data array access
L1I prefetcher	Tagged
L1D prefetcher	Stride 8 prefetcher
Private L2 cache	512 KiB/core, 8-way, 8 cycle access lat.
DynAMO	128 entries, 4-way
L2 prefetcher	Best Offset Prefetcher
Shared L3 cache	Exclusive, 32 slices of 1MiB, 8 ways, 10 cycles access lat.
Interconnect Architecture	
Coherence protocol	MOESI-like AMBA 5 CHI specification
Network topology	8×8 2D mesh
Router and link latency	1 cycle route, 1 cycle link
Main Memory	
Type	HBM3, 32GB storage
Channel	8 channels, 64 GB/s per channel

of time, one of the counters can increment its value drastically while the other stays constant. This conditions future predictions: although subsequent program phases may have different access patterns, the counters will keep their high values inherited from a previous phase. Thus, we adopt a simple solution: to shift the counters one bit to the right every certain number of cycles or before overflowing. This way, the counters have better confidence levels in different program phases and overflows are avoided.

5.3 Reuse-Pattern DynAMO

The second DynAMO design refines the previous metric-based predictor. The idea is to capture the reuse pattern of each cache block fetched by an AMO by tracking if any other memory access has reused the block during its lifespan in the L1D cache. The intuition behind this heuristic, DynAMO-Reuse, is to keep bringing cache blocks that are likely to be reused via *near* AMOs and avoid bringing cache blocks with no reuse via *far* AMOs.

To learn the reuse pattern of a cache block, the AMT has a *reuse bit* and a *reuse confidence counter* for each entry. When DynAMO-Reuse predicts a *near* AMO and fetches a cache block into the L1D cache, it resets the reuse bit. The predictor sets the reuse bit if that same cache block receives a subsequent hit by any other memory access. When the L1D cache block is evicted or invalidated due to a snoop message, DynAMO-Reuse increases the *reuse confidence counter* if the *reuse bit* is set. Otherwise, it decreases the confidence counter.

The reuse confidence counter encodes the reuse behavior of a cache block. A zero value means the cache block has lost all the reuse confidence. This indicates that future references to such cache blocks should be executed *far* in the HN, unless the cache block is already in *unique* state in the L1D cache. On the other hand, a positive counter implies that the cache block has reuse confidence. DynAMO-Reuse executes these AMOs *near* in order to fetch the cache block into the L1D and exploit the potential reuse, performing the AMO locally.

As we will see in Section 6, this version of the predictor, named DynAMO-Reuse-UN, performs better than any other static policy for some benchmarks. However, the decision to apply *far* execution for I, SC, and SD cache states when the reuse confidence counter arrives at zero is aggressive, and we find that always choosing *far* AMO execution in this situation can degrade the system's performance in some applications. Therefore, we have designed a variation of the predictor that applies the *Present Near* policy for zero reuse confidence cache blocks, which is more conservative. This version is named DynAMO-Reuse-PN.

When a new entry is allocated in the AMT, DynAMO-Reuse must perform an uninformed decision. This first decision of the predictor is fundamental for AMO performance and it is especially important for applications that exhibit a streaming/thrashing access pattern, i.e., bring new cache blocks into the L1D that will not be referenced again. Predicting *near* as the first decision for these cache blocks would evict other cache blocks that may be reused.

To avoid this issue, DynAMO-Reuse employs a heuristic that tracks the amount of local reuse seen by all AMOs. By counting the total number of cache blocks brought into the L1D cache by AMOs, and the total number of these blocks that have been reused, a global view of the amount of cache reuse can be obtained. This ratio determines the first decision: if reuse is low, the newly allocated AMO in the AMT will execute *far*, and *near* otherwise. We can identify and filter the streaming/thrashing access patterns with this approach.

After the first decision is taken, the entry is allocated, setting the confidence counter to its maximum value. Therefore, the next decision for that memory address will be to execute *near*. If the access pattern of AMOs executing on that address has low reuse, the subsequent fetch and eviction/invalidation actions will decrease the counter to zero. Then, the predictor will start selecting *far* AMO execution for that cache block.

6 EVALUATION

6.1 Methodology

Simulation Infrastructure: To evaluate the static policies and DynAMO predictors we use the gem5 (v20.1.0.0) performance simulator [48]. Gem5 includes cycle-approximate models of various system components, including cores, cache hierarchy with a detailed mesh-based network-on-chip, and memory controllers. We simulate a multi-core system consisting of 32 Neoverse-N1-like out-of-order cores, as detailed in Table 2. We model HBM3 memory following the high bandwidth trend in HPC systems [53]. We have extended the existing CHI protocol implementation in gem5 [28] to support the AMO transactions specified in the original AMBA 5 CHI specification [9], enabling the simulation of a NoC capable of executing *near* and *far* AMOs. The simulated system resembles recent architectures such as the AWS Graviton 3, and runs Ubuntu 20.04 with Linux kernel 5.4.65. We used McPAT 1.3 [46] with the enhancements Xi et al. [66] proposed to estimate dynamic energy

Table 3: Benchmark inputs & characteristics.

	Name	Code	Input	Sync. primitives employing AMOs	AMO Footprint
Splash-3	Barnes	BAR	16k [65]	POSIX mutex	84 KB
	FMM	FMM	16K [65]	POSIX mutex	97 KB
	Ocean_cp	OCE	512x512	POSIX mutex	4 KB
	Radiosity	RAD	room [65]	POSIX mutex	163 KB
	Raytrace	RAY	car [65]	POSIX mutex	8 KB
	Volrend	VOL	head [65]	POSIX mutex	6 KB
	Water-Ns	WAT	3375 mol	POSIX mutex, cas	65 KB
Galois	BFS	BFS	USA [22]	Spinlock, ldmin	52 MB
	CC	CC	USA [22]	Spinlock, ldmin	182 MB
	Cluster	CLU	NY [22]	Spinlock, stadd	15 MB
	GMETIS	GME	FLA [22]	Spinlock, cas	144 MB
	KCORE	KCOR	USA [22]	Spinlock, ldadd	91 MB
	Page Rank	PR	FLA [22]	Spinlock, cas	4 MB
	SPT	SPT	USAW [22]	Spinlock, cas	95 MB
	SSSP	SSSP	USA [22]	Spinlock, stmin	1 MB
GAP	BC	BC	Kronecker [45]	OpenMP	4 MB
	TC	TC	Kronecker [45]	OpenMP	10 KB
	Fluidanimate	FLU	simlarge [14]	POSIX mutex, cas	8 MB
	Histogram	HIST	IMG [17, 51]	stadd	2 MB
	Radix Sort	RSOR	2 MB vector	POSIX barrier, stadd	512 KB
	SPMV	SPMV	JP & rma10 [19]	stadd	3 MB

consumption. We perform this estimation using a process technology node of 22nm, a supply voltage of 0.8V, and the default clock gating scheme.

Workloads: We use an extensive set of well-known parallel applications to cover all types of synchronization primitives and direct atomic updates (e.g., ldadd or stadd):

- **Splash-3** [56] is a classic parallel scientific benchmark suite. It mainly uses POSIX primitives for synchronization.
- **Galois** [39] is an optimized graph analytics framework that contains classic graph analysis algorithms. It has its own spinlock implementation and uses direct atomic updates.
- **GAP benchmark suite** [12] is a popular graph algorithm suite written in OpenMP.
- **Fluidanimate** is a benchmark from the PARSEC benchmark suite [14] that makes use of a fine-grained synchronization.
- **Histogram** is based on the OpenCV [16] color histogram program (version 2.4.11).
- **SMPV**[67] is a sparse matrix-vector multiplication kernel where matrices use the compressed sparse column format.
- **Parallel 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 [60].

Table 3 details for each evaluated benchmark: the acronym, the input, the synchronization primitives and direct AMOs used, as well as the memory footprint used by AMOs.

Next, we characterize the usage of AMOs on the evaluated applications. Figure 6 shows the number of committed AMOs per kilo-instruction (APKI) for each workload. Applications with a higher APKI present more opportunities for DynAMO to improve

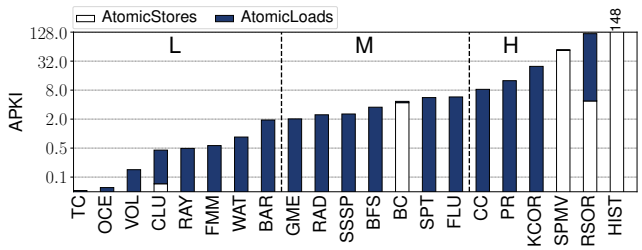

Figure 6: AMOs per kilo-instruction (APKI) for each workload. We split workloads into three sets: Low (L), Medium (M), and High (H) based on their APKI. The ratio of AtomicLoads and AtomicStores is represented as stacked bars.

performance. Hence, to highlight the performance improvement achieved by DynAMO, we define three sets of workloads depending on their APKI: Low, Medium, and High AMO intensity. The Low set contains applications that range from 0 to 2 APKI. The Medium set contains those between 2 and 8 APKI, and the High set those with more than 8 APKI.

6.2 Evaluation of Static AMO Policies

Figure 7 shows the relative speed-up of the different static AMO policies (Section 4) normalized with respect to *All Near*. The benefits of applying each of the static AMO policies are application dependent. We can group static AMO policies into two groups by their behavior. The first group is composed of *All Near*, *Present Near*, and *Shared Far* policies. In most cases, these three policies favor *near* AMOs, bringing the cache block into the L1D. *All Near* performs well in most cases because most applications present a reuse access pattern. Overall, *Present Near* performs better than *All Near* and *Shared Far* because it achieves high speed-ups in SPMV, Radix Sort, and Histogram: 1.62×, 1.26× and 2.29×, respectively. On average, *Present Near* is the best static policy and achieves speed-ups of 1.05× across all applications, 1.09× for Medium and High applications, and 1.19× for High APKI applications. *Shared Far* achieves lower performance than *All Near* because it predicts *far* AMOs for one of the most frequent cache states (shared clean), which degrades the performance of applications with reuse patterns. *Shared Far* has, on average, slowdowns of 8.2% for all applications, 9, 9% for Medium and High applications sets, and 11% for High APKI benchmarks.

The second group is composed of *Dirty Near* and *Unique Near* policies, which in most cases, issue *far* AMOs rather than *near* AMOs. Both policies apply the same decisions on all cache states except for SD. This cache state is very infrequent compared to UD, SC, and I states, so their difference in performance is minimal. *Shared Far*, *Dirty Near*, and *Unique Near* suffer slowdowns in applications with reuse access patterns. However, both policies outperform *All Near* and *Present Near* for Volrend, SPMV, Radix Sort, and Histogram benchmarks. On average, *Dirty Near* has slowdowns of 2.5% for all applications, 2.7% on Medium and High applications, and a speed-up of 1.06× for High APKI benchmarks. Meanwhile, Unique Near has slightly better averages with a slowdown of 1.2% for all applications, 0.6% for Medium and High applications, and a speed-up of 1.10× for high APKI applications.

Static AMO policies that issue a *far* AMO when the cache block is in Shared Clean state increment the execution time for many

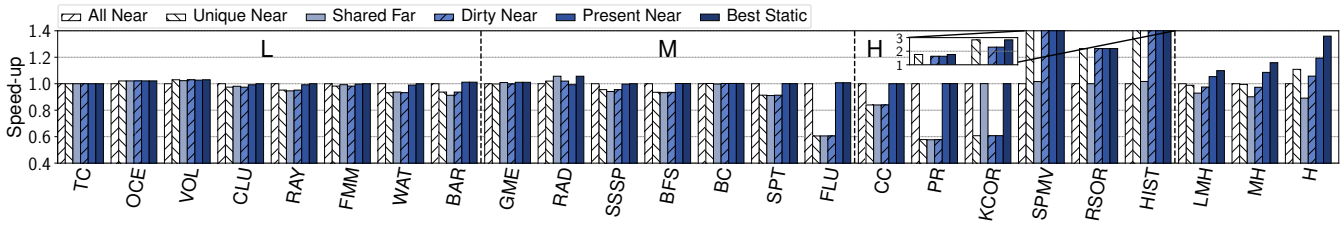

Figure 7: Execution speed-up of static AMO policies normalized with respect to *All Near*. Right most bars are the geomean of LMH, MH, and H application sets.

benchmarks, e.g., Raytrace, Water-Ns, Barnes, BFS, Fluidanimate, CC, PR, and Kcore. These benchmarks have significant reuse patterns and frequently read the variable before updating it with an AMO. Even though several benchmarks suffer from lock contention, *far* AMOs perform poorly because Pthread Mutex implementation does not perform well with *far* AMOs, as explained in Section 3.2.3.

In contrast, for Radiosity, static policies that perform *far* AMOs in SC state have a speed-up of 1.06×. Radiosity has a shared task queue accessed by all threads to enqueue or dequeue tasks. A single high-contended lock protects this queue. The lock is read before acquisition to avoid performing unnecessary AMOs. Therefore, lock and unlock operations are done at the LLC for *Shared Far*, *Dirty Near*, and *Unique Near*. When accessed through *near* AMOs, the lock ping-pongs from one core L1D cache to another as exclusivity requests are ordered by the directory. Hence, when the lock is accessed through *far* AMOs, all lock and unlock operations are forwarded to the LLC, removing the ping-pong effect.

SMPV, Radix Sort, and Histogram work well when employing *far* AMOs. The three benchmarks have a large AMO footprint. A few addresses have reuse patterns, while the rest are only accessed once. Thus, they exhibit a mixed access pattern in which a small working set is highly reused and fits in the private caches, and a more extensive working set is not reused. Thus, if the largest working set is accessed through *near* AMOs, it evicts those memory positions that are highly reused.

Finally, the *Best Static* bar represents the performance we can achieve by selecting the best static AMO policy for each workload. Programmers could achieve this performance by profiling the application and configuring the system to work with a specific static AMO policy. This method achieves an average 1.10× speed-up for all applications (LMH), 1.16× for Medium and High (MH) applications sets, and 1.35× for High (H) APKI applications. However, this approach is limited because it requires profiling using the target platform. In addition, some applications behave differently depending on the input data used. Moreover, static policies apply the same decision to all addresses without discerning the behaviors of different working sets. Therefore, achieving the best performance for each benchmark would require a hardware mechanism that dynamically selects the best placement to execute a given AMO.

6.3 DynAMO Evaluation

Figure 8 shows the speed-ups obtained with both DynAMO implementations. DynAMO-Metric is the design in which predictions are driven by cache block statistics. DynAMO-Reuse is the design that tracks the reuse pattern of each cache block and uses this information to select the best AMO execution placement. Furthermore, DynAMO-Reuse uses previous AMO executions to predict the first placement for a newly referenced address. We have two flavours of the DynAMO-Reuse predictor depending of which Static AMO policy they apply to blocks with zero reuse confidence. DynAMO-Reuse-UN applies *Unique Near* policy and DynAMO-Reuse-PN applies *Present Near* policy. In addition, we plot the *Best Static* bar as a comparison point. All results are normalized with respect to *All Near*. Although DynAMO-Metric achieves modest speed-ups on Volrend, FMM, and Gmetis, and, on average, performs equally well as the *All Near* baseline.

DynAMO-Reuse-UN achieves speed-ups of 1.06× for all parallel applications (LMH), 1.11× for MH intensity applications, and 1.25× speed-up for H intensity APKI applications. While DynAMO-Reuse-PN achieves 1.09×, 1.14× and 1.31×, respectively. Therefore, both DynAMO-Reuse predictors perform better than *Present Near*, which was the best individual static policy, and capture most of the performance obtained by the *Best Static* bar. While *Best Static* is a good upper bound comparison point. However, it is not an implementable baseline because it requires knowing in advance the best policy for any given workload-input pair.

Both DynAMO-Reuse predictors capture the reuse patterns of those variables that have locality and perform the AMO *near*. Thus, all two achieve the same performance as *All Near*, which is the best policy for these cases. However, since the DynAMO-Reuse-UN predictor is prone to predict more *far* AMOs, we observe a degradation in performance for Cluster, Raytrace, FMM, SPT, CC and PR. The DynAMO-Reuse-PN predictor is more conservative in these cases and predicts more *near* AMOs, achieving a higher performance. Therefore, the DynAMO-Reuse-PN predictor always performs equal or better than the baseline (*All Near*).

There are several applications where variables have no locality and their allocation in the L1D cache creates a thrashing behavior that harms the performance: Gmetis, SPMV, Radix Sort and Histogram. In these cases, both predictors detect which cache blocks are not reused and perform the AMOs at the HN. In particular, Barnes, Gmetis and Radix Sort have multiple phases and various variables accessed by AMOs with different locality patterns. Both predictors capture the dynamic behaviour of such applications delivering a performance unachievable to the best static AMO policies. However, for SPMV and Histogram both predictors do not match the performance of the best static policy (*Unique Near*). In these two applications, some cache blocks have a re-reference period short enough to be predicted as *near* AMOs, fetching the cache block into the L1D.

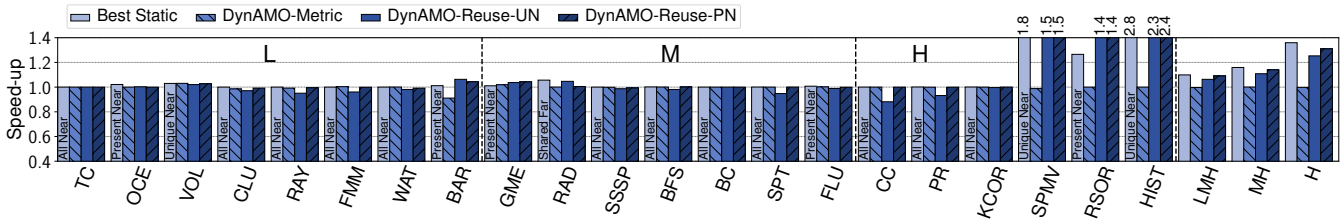

Figure 8: Execution speed-up of DynAMO predictors normalized to *All Near*. We add the *best static* bar with the best-performing static policy. Right most bars are the geomean of LMH, MH, and H application sets.

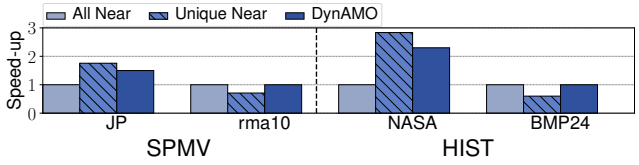

Figure 9: Execution speed-up of *Unique Near* and DynAMO-Reuse-PN predictor normalized to *All Near* for each input.

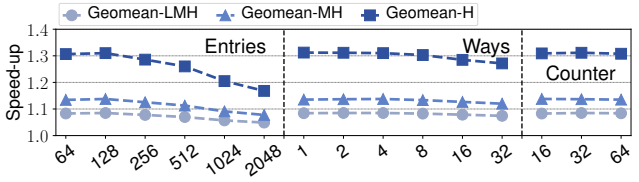

Figure 10: Execution speed-up of DynAMO-Reuse-PN predictor normalized to *All Near*. Left: Different entry counts with fixed 4-way associativity. Center: Different associativity values for a 128-entry structure. Left: different counter size for a 128-entry and 4-way structure.

Barnes and Radiosity are the only benchmarks where the DynAMO-Reuse-UN predictor outperforms DynAMO-Reuse-PN with a speed-up of 1.06× and 1.04×, respectively. In both benchmarks, the DynAMO-Reuse-UN predictor captures the ping-pong effect of some locks and performs the lock and unlock operations at the HN.

DynAMO-Reuse-PN provides the best overall performance gains over the baseline (*All Near* policy) because it accurately selects the most appropriate placement to execute the AMOs. Furthermore, DynAMO-Reuse-PN nearly reaches the upper bound performance of the Best Static policy without profiling and programmer hints.

6.4 Input Sensitive Benchmarks

In Section 6.3, we have seen that choosing the best static AMO policy achieves the highest performance. However, the performance of static AMO policies is tightly related to the access pattern of the application and these maybe dependent of the input data-set. To illustrate the input sensitivity of benchmarks, we have tested two different data-set inputs for SPMV and HIST (see Figure 9). In the experiments, we use *All Near* as our baseline, the best static policy obtained from Figure 7 (*Unique Near* for both applications), and the DynAMO-Reuse-PN. *Unique Near* outperforms *All Near* for JP input in SPMV and NASA input in HIST, but performs 30% slower than *All Near* for rma10 input in SPMV and 40% slower for BMP24 input in HIST. Meanwhile, DynAMO-Reuse-PN captures the different access patterns and adapts better to each input.

6.5 Energy Efficiency

We have computed and analyzed the dynamic energy consumed by *All Near*, *Unique Near*, and DynAMO-Reuse-PN. Dynamic energy reductions correlate with performance improvements, translating into reductions of 4, 6, and 12% for Low, Medium, and High APKI application sets, respectively. The dynamic energy consumed by the NoC mainly stems from the number of sent messages and remains relatively constant across policies for benchmarks in the Low and Medium APKI sets. However, for High APKI benchmarks, the dynamic energy spent in the NoC for DynAMO is noticeably higher than *All Near* for SPMV and HIST, as the core executes most AMOs remotely, increasing NoC traffic. Nevertheless, SPMV and HIST achieve overall energy reductions of up to 40% due to shorter execution times.

6.6 Impact of AMT Sizing

Figure 10 presents the performance of DynAMO-Reuse-PN for different configurations when changing the number of entries, ways, and the size of the reuse counter, with respect to the baseline (*All Near*). We evaluate different number of entries fixing the ways to 4 and the size of the reuse counter to 32 (left), we evaluate different ways fixing the number of entries to 128 and the reuse counter to 32 (center), and we evaluate the size of the reuse counter fixing the size of the predictor to 128 entries and 4 ways (right). We find that a modest 128-entry 4-way AMT with a reuse counter size of 32 (5 bits) is the best configuration for the predictor because most applications have a small AMO footprint that experiences L1D reuse. The performance of the predictor degrades for high APKI applications when the sizing or the associativity of the AMT table grows because the lifespan of entries is overextended. Thus, reducing the accuracy of predictions. By having a small number of entries, we reduce the lifespan of entries, removing outdated entries from the predictor.

6.7 Hardware Cost

Each AMT entry requires 49 bits for the physical address TAG, 5 bits for the reuse confidence counter and the reuse bit. Totalling 55 bits per entry, but we consider 64 bits. For a 128-entry AMT, the predictor needs a modest 1KB of storage per core. In addition, we have used CACTI 6.5 to estimate the area consumed by the predictor using a 22nm process and assuming it has 2 read and 1 write ports. The total area estimation is $0.0196mm^2$. As a reference, the simulated 64KB L1D cache has an area estimation of $0.3020mm^2$, i.e., 15× larger.

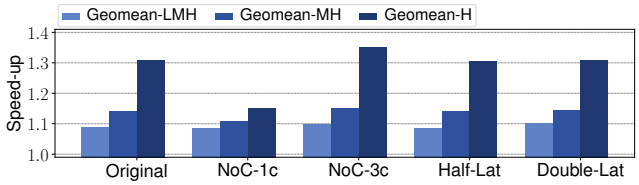

Figure 11: Execution speed-up of DynAMO-Reuse-PN predictor normalized to *All Near* on each different system: original (2 cycle hop latency), 1 and 3 cycle NoC hop latency, and halving and doubling the HBM latency.

6.8 System Design Space Exploration

This section quantifies the performance of the DynAMO-Reuse-PN predictor in systems with different architectures. Figure 11 presents the geomean of each application set normalized to *All Near* on each system. We have plotted the original system for reference. The first two experiments reconfigure the original NoC hop cost, which is one cycle for routing and another for link traversal. We test an NoC with no routing cost (NoC-1c), and one with a routing cost of 2 (NoC-3c). With one cycle hop cost the predictor reduces its speed-up due to a lower penalty of cache block ping-ponging. However, for the 3 cycle hop cost the DynAMO predictor increases the performance gains for high APKI benchmarks. The last two experiments test a system with half (Half-Lat) and double (Double-Lat) HBM latency cost. DynAMO performance is unaffected by main memory latency.

7 RELATED WORK

Academic literature has profusely explored hardware and software techniques to reduce the cost of synchronization instructions and atomic updates to shared data.

Most proposals focus on using dedicated hardware (accelerators, networks, new instructions) to speed-up synchronization primitives [1, 6, 25, 41, 47, 55, 61, 62, 69]. However, these proposals incur large area overheads. Many proposals also focus on speeding-up Barriers [2, 4, 13, 37, 52, 57, 59]. Fujitsu's A64FX processors [27] include specific instructions for barriers. Other techniques try to reduce the producer-to-consumer latency of AMOs or other write operations by forwarding data [3, 6, 24, 50, 68]. Recent research works have turned towards exploiting the capabilities of RMW updates instead of using locks [23, 67]. However, these approaches require support for Floating Operations in the cache hierarchy.

Academia and industry have proposed a plethora of alternatives to *near* and *far* AMOs, introducing new synchronization instructions [3, 6, 47, 55, 61, 62]. However, most ISAs are reticent to include instructions. The encoding space is a scarce resource in fixed-width ISA, and increasing the length of instructions increases the complexity of instruction fetch and decode. Moreover, these new instructions require changes in the software stack.

New paradigms with richer semantics may ease the programmability and provide better performance. Transactional Memory (TM) promises good parallel performance and easy-to-write parallel code [32]. Multi-Address atomic operations (MAD) [29] aim to enhance fine-grained synchronization performance. Task scheduling optimizes task execution through hardware support [40].

Table 4: Characteristics of synchronization alternatives.

Solution	Transparent	Performance	Cost
Far AMO	✓	×	✓
Custom Instructions [47, 61]	×	✓	✓
Accelerators [6, 41, 55]	✓	✓	×
Custom Networks [1, 25]	✓	✓	×
Parallel Reductions [23, 67]	×	✓	×
Core to Core[40, 61]	×	✓	✓
DynAMO	✓	✓	✓

Table 4 summarizes state-of-the-art proposals and qualitatively compares them in terms of (i) transparency, i.e., modifications at the application or software stack level required, (ii) performance, i.e., whether they achieve competitive performance improvements, and (iii) cost, i.e., whether implementation costs are large or not. Compared to prior work, DynAMO is the only proposal that is transparent to the programming interface, has a small cost associated with it, and achieves competitive performance by improving the best individual static AMO policy.

8 CONCLUSIONS

The open AMBA 5 CHI specification is being adopted by multiple hardware vendors for multi-core SoC designs. In this paper, we analyze and evaluate systems that support the execution of *near* and *far* AMOs as specified by AMBA 5 CHI. We show that currently available static policies (*All near* and *Unique Near*) fail to deliver the best performance over a wide range of workloads.

Therefore, we propose and evaluate three additional static polices and find that: (i) performance of static policies is highly application dependent; and (ii) the best static policy, which we term *Present Near*, is not one of the implemented policies by hardware vendors that support both *near* and *far* AMOs.

We then propose DynAMO, a simple predictor that dynamically decides the best location to execute the AMOs (*near* or *far*). DynAMO identifies the different locality patterns to take informed decisions, improving AMO latencies and increasing overall throughput. Our best DynAMO implementation is able to outperform the best static policy (*Present Near*), and it nearly matches per-workload *Best Static* without requiring any profiling or programmer input. DynAMO provides geometric mean speed-ups of 1.09× across all workloads and 1.31× on AMO-intensive applications with respect to a baseline that executes all AMOs *near*.

ACKNOWLEDGMENTS

This research was supported by the Spanish Ministry of Science and Innovation (MCIN) through contracts [PID2019-107255GB-C21], [TED2021-132634A-I00], and [PID2019-105660RB-C21]; the Generalitat of Catalunya through contract [2021-SGR-00763]; the Government of Aragon [T5820R]; the Arm-BSC Center of Excellence, and the European Processor Initiative (EPI) which is part of the European Union's Horizon 2020 research and innovation program under grant agreement No. 826647. V. Soria-Pardos has been supported through an FPU fellowship [FPU20-02132]; A. Armejach is a Serra Hunter Fellow and has been partially supported by the Grant [IJCI-2017-33945] funded by MCIN/AEI/10.13039/501100011033; M. Moretó through a Ramón y Cajal fellowship [RYC-2016-21104].

REFERENCES

[1] Sergi Abadal, Albert Cabellos-Aparicio, Eduard Alarcon, and Josep Torrellas. 2016. WiSync: An Architecture for Fast Synchronization through On-Chip Wireless Communication. *SIGARCH Comput. Archit. News* 44, 2 (mar 2016), 3–17.

[2] Jose L. Abellán, Juan Fernández, and Manuel E. Acacio. 2010. A G-Line-Based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs. In *2010 39th International Conference on Parallel Processing*. 267–276.

[3] Jose L. Abellán, Juan Fernández, and Manuel E. Acacio. 2011. GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *2011 IEEE International Parallel Distributed Processing Symposium*. 893–905.

[4] José L. Abellán, Juan Fernández, and Manuel E. Acacio. 2012. Efficient Hardware Barrier Synchronization in Many-Core CMPs. *IEEE Transactions on Parallel and Distributed Systems* 23, 8 (2012), 1453–1466.

[5] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. 1995. The MIT Alewife Machine: Architecture and Performance. *SIGARCH Comput. Archit. News* 23, 2 (may 1995), 2–13.

[6] B. E. S. Akgul and V. J. Mooney III. 2002. The System-on-a-Chip Lock Cache. *Design Automation for Embedded Systems* 7, 1 (2002), 139–174.

[7] Amazon Web Services. 2021. New – Amazon EC2 C7g Instances, Powered by AWS Graviton3 Processors. https://aws.amazon.com/blogs/aws/new-amazon-ec2-c7g-instances-powered-by-aws-graviton3-processors/. [Online; accessed 30-July-2022].

[8] Arm holdings. 2020. Arm Neoverse N2 Core Technical Reference Manual. https://developer.arm.com/documentation/102099/0001/The-Neoverse-N2--core. [Online; accessed 2-December-2021].

[9] Arm Holdings. 2021. AMBA 5 CHI Architecture Specification. https://developer.arm.com/architectures/system-architectures/amba/amba-5. [Online; accessed 30-July-2022].

[10] Ashkan Asgharzadeh, Juan M. Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. 2022. Free Atomics: Hardware Atomic Operations without Fences. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 14–26.

[11] Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A communication characterisation of Splash-2 and Parsec. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 86–97.

[12] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite.

[13] Carl J. Beckmann and Constantine D. Polychronopoulos. 1990. Fast Barrier Synchronization Hardware. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (New York, New York, USA) *(Supercomputing '90)*. IEEE Computer Society Press, Washington, DC, USA, 180–189.

[14] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81.

[15] Gavin Bonshor. 2021. AMD Releases Milan-X CPUs With 3D V-Cache: EPYC 7003 Up to 64 Cores and 768 MB L3 Cache. https://www.anandtech.com/show/17323/amd-releases-milan-x-cpus-with-3d-vcache-epyc-7003. [Online; accessed 30-July-2022].

[16] Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computervision with the OpenCV library*. O'Reilly.

[17] John Burkardt. 2021. PNG Files Florida State University. https://people.sc.fsu.edu/~jburkardt/data/png/bmp_24.png. [Online; accessed 30-July-2022].

[18] The C++ Standards Committee. 2023. std::Atomic Library. https://en.cppreference.com/w/cpp/header/atomic. [Online; accessed 12-April-2023].

[19] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages.

[20] A. de Dios, B. Sahelices, P. Ibáñez, V. Viñals, and J. M. Llabería. 2006. Speeding-Up Synchronizations in DSM Multiprocessors. In *Euro-Par 2006 Parallel Processing*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 473–484.

[21] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.

[22] DIMACS. 2006. The Ninth DIMACS challange on shortest paths. http://www.dis.uniroma1.it/challenge9/.. [Online; accessed 30-July-2022].

[23] V. Dimić, M. Moretó, M. Casas, J. Ciesko, and M. Valero. 2020. RICH: Implementing Reductions in the Cache Hierarchy. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 13 pages.

[24] Zhen Fang, Lixin Zhang, John B. Carter, Ali Ibrahim, and Michael A. Parker. 2007. Active Memory Operations. In *Proceedings of the 21st Annual International Conference on Supercomputing* (Seattle, Washington) *(ICS '07)*. Association for Computing Machinery, New York, NY, USA, 232–241. https://doi.org/10.1145/1274971.1275004

[25] Antonio Franques, Apostolos Kokolis, Sergi Abadal, Vimuth Fernando, Sasa Misailovic, and Josep Torrellas. 2021. WiDir: A Wireless-Enabled Directory Cache Coherence Protocol. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 304–317.

[26] Andrei Frumusanu. 2021. The Ampere Altra Max Review: Pushing it to 128 Cores per Socket. https://www.anandtech.com/show/16979/the-ampere-altra-max-review-pushing-it-to-128-cores-per-socket. [Online; accessed 30-July-2022].

[27] Fujitsu. 2021. Fujitsu A64FX Datasheet. https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf. [Online; accessed 30-July-2022].

[28] gem5 and Arm Holdings. 2021. gem5 CHI Protocol. https://www.gem5.org/documentation/general_docs/ruby/CHI/. [Online; accessed 30-July-2022].

[29] E. J. Gómez-Hernández, J. M. Cebrian, R. Titos-Gil, S. Kaxiras, and A. Ros. 2021. Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 337–349.

[30] James R. Goodman, Mary K. Vernon, and Philip J. Woest. 1989. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 64–75.

[31] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. 1998. The NYU Ultracomputer—Designing a MIMD, Shared-Memory Parallel Machine. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)* (Barcelona, Spain) *(ISCA '98)*. Association for Computing Machinery, New York, NY, USA, 239–254. https://doi.org/10.1145/285930.285983

[32] M. Herlihy and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 289–300.

[33] H. Hoffmann, D. Wentzlaff, and A. Agarwal. 2010. Remote Store Programming. In *High Performance Embedded Architectures and Compilers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–17.

[34] Arm Holdings. 2022. Armv8 Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/ha/?lang=en. [Online; accessed 30-July-2022].

[35] Arm Holdings. 2022. Do near or far atomics give the best performance on Neoverse systems? https://developer.arm.com/documentation/ka004706/latest/. [Online; accessed 30-July-2022].

[36] HPC Wire. 2020. AWS Graviton2-Powered EC2 Instances Now Available. https://www.hpcwire.com/2020/06/12/aws-graviton2-powered-ec2-instances/. [Online; accessed 2-December-2021].

[37] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. 1998. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain) *(ISCA '98)*. IEEE Computer Society, USA, 306–317.

[38] R.E. Kessler and J.L. Schwarzmeier. 1993. Cray T3D: a new dimension for Cray Research. In *Digest of Papers. Compcon Spring*. 176–182.

[39] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 65–76.

[40] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. Association for Computing Machinery, New York, NY, USA, 162–173.

[41] Andreas Kurth, Samuel Riedel, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2020. ATUNs: Modular and Scalable Support for Atomic Operations in a Shared Memory Multiprocessor. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[42] J. Laudon and D. Lenoski. 1997. The SGI Origin: A CcNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 241–251.

[43] H. Q. Le, J. A. Van Norstrand, B. W. Thompto, J. E. Moreira, D. Q. Nguyen, D. Hrusecky, M. J. Genden, and M. Kroener. 2018. IBM POWER9 processor core. *IBM Journal of Research and Development* 62, 4/5 (2018), 2:1–2:12.

[44] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12 (2001), 1352–1361.

[45] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. 2005. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *Knowledge Discovery in Databases: PKDD 2005*, Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 133–145.

[46] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 469–480.

[47] C. Liang and M. Prvulovic. 2015. MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 414–426.

[48] J. Lowe-Power, A. Mutaal Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. Rodrigues Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR]

[49] G.E. Moore. 1998. Cramming More Components Onto Integrated Circuits. *Proc. IEEE* 86, 1 (1998), 82–85.

[50] M. Musleh and V. S. Pai. 2015. Automatic Sharing Classification and Timely Push for Cache-Coherent Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages.

[51] NASA. 2021. NASA Video and Image Library. https://images.nasa.gov/. [Online; accessed 30-July-2022].

[52] Jungju Oh, Milos Prvulovic, and Alenka Zajic. 2011. TLSync: Support for multiple fast barriers using on-chip transmission lines. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 105–115.

[53] Myeong-Jae Park, Ho Sung Cho, Tae-Sik Yun, Sangjin Byeon, Young Jun Koo, Sangsic Yoon, Dong Uk Lee, Seokwoo Choi, Jihwan Park, Jinhyung Lee, Kyungjun Cho, Junil Moon, Byung-Kuk Yoon, Young-Jun Park, Sang-muk Oh, Chang Kwon Lee, Tae-Kyun Kim, Seong-Hee Lee, Hyun-Woo Kim, Yucheon Ju, Seung-Kyun Lim, Seung Geun Baek, Kyo Yun Lee, Sang Hun Lee, Woo Sung We, Seungchan Kim, Yongseok Choi, Seong-Hak Lee, Seung Min Yang, Gunho Lee, In-Keun Kim, Younghyun Jeon, Jae-Hyung Park, Jong Chan Yun, Chanhee Park, Sun-Yeol Kim, Sungjin Kim, Dong-Yeol Lee, Su-Hyun Oh, Taejin Hwang, Junghyun Shin, Yunho Lee, Hyunsik Kim, Jaeseung Lee, Youngdo Hur, Sangkwon Lee, Jieun Jang, Junhyun Chun, and Joohwan Cho. 2022. A 192-Gb 12-High 896-GB/s HBM3 DRAM with a TSV Auto-Calibration Scheme and Machine-Learning-Based Layout Optimization. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 65. 444–446. https://doi.org/10.1109/ISSCC42614.2022.9731562

[54] GNU Project. 2023. GCC __atomic Builtins. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html. [Online; accessed 12-April-2023].

[55] John T. Robinson. 1985. A Fast General-Purpose Hardware Synchronization Mechanism. *SIGMOD Rec.* 14, 4 (may 1985), 122–130.

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

[57] Jack Sampson, Ruben Gonzalez, Jean-francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder. 2006. Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 235–246.

[58] S. L. Scott. 1996. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 26–36.

[59] Shisheng Shang and Kai Hwang. 1995. Distributed hardwired barrier synchronization for scalable multiprocessor clusters. *IEEE Transactions on Parallel and Distributed Systems* 6, 6 (1995), 591–605.

[60] Andrew Sohn and Yuetsu Kodama. 1998. Load Balanced Parallel Radix Sort. In *Proceedings of the 12th International Conference on Supercomputing* (Melbourne, Australia) *(ICS '98)*. Association for Computing Machinery, New York, NY, USA, 305–312. https://doi.org/10.1145/277830.277903

[61] X. Tang, J. Zhai, X. Qian, and W. Chen. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 765–778.

[62] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. 2010. Architectural Support for Fair Reader-Writer Locking. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 275–286.

[63] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro* 27, 5 (2007), 15–31.

[64] C. M. Wittenbrink, Emmett K., and A. Prabhu. 2011. Fermi GF100 GPU Architecture. *IEEE Micro* 31, 2 (2011), 50–59.

[65] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 24–36.

[66] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2015. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 577–589. https://doi.org/10.1109/HPCA.2015.7056064

[67] G. Zhang, W. Horn, and D. Sanchez. 2015. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 13–25.

[68] L. Zhang, Z. Fang, and J.B. Carter. 2004. Highly efficient synchronization based on active memory operations. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 58–.

[69] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. 2007. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. *SIGARCH Comput. Archit. News* 35, 2 (jun 2007), 35–45.