SYNAPSE: Decoupling Protocol, Topology, and Policy for Automated Synthesis of Coherence Controllers

Nicolò Carpentieri
Technical University of Munich
Munich, Germany

Anatole Lefort
Technical University of Munich
Munich, Germany

David Schall
Technical University of Munich
Munich, Germany

Julian Pritzi
Technical University of Munich
Munich, Germany

Pramod Bhatotia
Technical University of Munich
Munich, Germany

***Abstract*—Designing cache coherence protocols (CCPs) is a highly complex task: modern heterogeneous systems impose distinct constraints on protocols, topologies, and caching policies. This challenge is only intensified as coherent interconnects push cache coherence beyond chip boundaries toward rack-scale deployments.**

We propose SYNAPSE, a methodology for synthesizing cache coherence controllers grounded in a key observation: any controller's state machine is fully determined by three orthogonal constituents: *protocol specification*, *system topology*, and *caching policies*. Central to our approach is an abstract controller model that decouples coherence logic from architectural constraints via two protocol-agnostic building blocks, a caching client and manager, enabling automated derivation of controllers for arbitrarily complex configurations. SYNAPSE realizes this through a fully automated synthesis pipeline with a declarative DSL, validates the correctness of the synthesized FSMs for liveness and safety properties, and generates gem5-based implementations for rapid design-space exploration.

In our evaluation, SYNAPSE successfully synthesizes diverse protocol variants across SMP, NUMA, multi-host CXL, and heterogeneous chiplet-based SoC scenarios, matching handwritten implementations within 1.0% performance on average. Targeting diverse architectures SYNAPSE surfaces key protocol-policy interactions, offering a systematic framework for designing coherence controllers in increasingly complex environments.

I. INTRODUCTION

Cache coherence protocols (CCPs) are a central component of modern hardware systems, serving as the common substrate that coordinates communication between CPU cores and system memory [8], [11], [44], [55], [63]. They provide a crucial abstraction for software developers by ensuring that all cores maintain a consistent view of data. Consistency is managed transparently and efficiently in hardware, relieving developers from the burden of manual consistency management.

Since the introduction of CCPs alongside the first multi-core processors, hardware architectures have become increasingly intricate along three dimensions that directly shape coherence design. First, *protocol diversity:* contemporary chips now incorporate a variety of core types, from energy-efficient to high-performance cores [9], [41], [42], and are augmented with diverse accelerators and coprocessors like GPUs [2], [7], [43], [64] and SmartNICs [48], [65], each governed by its own proprietary coherence protocol, ranging from Single-Writer-Multiple-Reader (SWMR)-based approaches to Release Consistency protocols. Second, *topology diversity*: systems have expanded beyond a single die through architectures such as chiplets [51], [54], NUMA clusters [50], and coherent interconnects such as CXL [22] and NVLink [66] that span entire server racks. Third, *policy diversity*: caching policies, governing how data is allocated, replaced, and written back, must be independently tailored to each architecture, as a policy optimal for one system may actively harm correctness or performance in another [13], [63], [74]. Together, these three dimensions define the CCP design space, and navigating it requires careful co-design of protocol, topology, and policy.

Designing CCPs across these three axes is notoriously difficult. At their core, CCPs function as distributed state machines communicating over a network, where ensuring freedom from data inconsistencies, deadlocks, and race conditions already requires extensive verification effort [68]–[70]. Yet correctness alone is insufficient: CCPs sit at the core of the system-on-chip (SoC) connecting all components, so even minor inefficiencies become system-wide bottlenecks — MSI and MESI (**M**odified, **E**xclusive, **S**hared, **I**nvalid) offer identical SWMR guarantees, yet MSI's higher coherence traffic renders it impractical in real systems [63]. Additionally, the increasing diversity of modern architectures places an even greater burden on CCP developers [30], [52], [53], [69]. The coherence requirements of a mobile SoC with a strict power budget, a mix of energy-efficient and performance-oriented cores, and various accelerators differ dramatically from those of a server chip with hundreds of high-performance cores [10], [37], [75].

The consequences for hardware architects are severe: the intersection of these three axes creates an enormous, continuously expanding design space, where traditional manual adaptation is becoming a fundamental bottleneck. Comprehensive exploration is now virtually impossible without a systematic way to navigate this complexity.

In this work, we ask: *how can coherence controllers be automatically synthesized for arbitrarily complex system configurations, spanning diverse protocols, topologies, and caching policies, while being correct?*

We present SYNAPSE, a new methodology for automated synthesis of coherence controllers for arbitrarily complex cache-coherent systems. SYNAPSE decouples any cache coherence controller's FSM into three main constituents: (1) the *protocol specification*, defining the stable and transient states of the coherence protocol; (2) the *system topology*, defining the controller's position in the memory hierarchy (e.g., L1, L2, or directory level); and (3) the *caching policies*, defining the actions a controller performs in response to incoming coherence events. By decoupling these three constituents, any cache controller for any

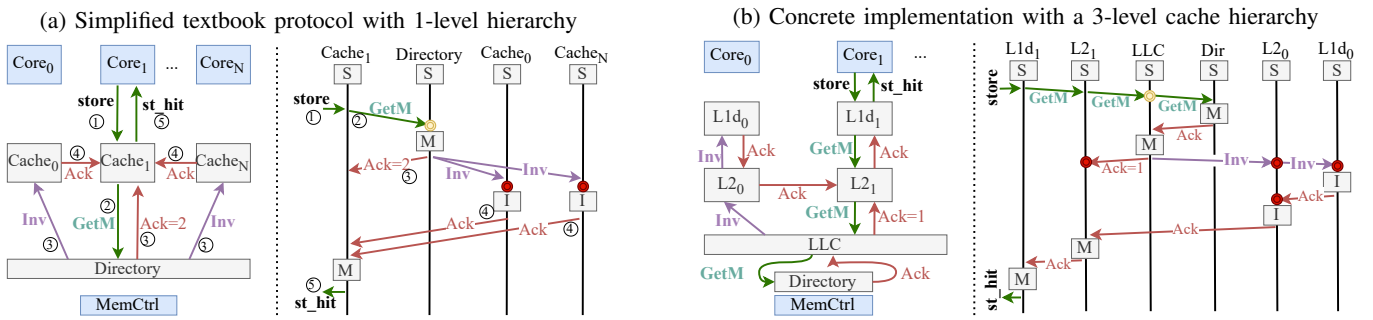


Fig. 1: Coherence transaction flow for store operation from $Core_1$. Left parts shows system architecture. Right parts the matching protocol message flows and controller state transitions. Red dots mark race points and the yellow point the serialization point.

architecture can be derived from this common set of ingredients.

To achieve this decomposition, we propose a **novel abstract cache controller model** and a **generalized synthesis algorithm**. By introducing two fundamental building blocks—the caching client and the caching manager—and defining their behavior via abstract states and requests, we establish a semantic mapping across heterogeneous protocols. The algorithm then derives the FSM of any target controller by fusing transition sequences from the manager and client FSMs according to the specific constraints of the caching policy. In this process, the protocol, policies, and topology act as a functional recipe that instantiates these building blocks into a concrete controller for any target system. This independence allows developers to swap a complete CCP without modifying the topology, or tailor to a specific controller instance through policies alone, without touching the protocol.

To realize this methodology, we develop a **synthesis pipeline** with three stages: (1) it takes a DSL specification and a system topology as input, (2) generates their FSMs via the synthesis algorithm, and (3) produces a complete gem5-based simulation setup with all inter-controller communication established. Correctness is guaranteed by an integrated validation stage that checks deadlock freedom (*liveness*) via Murφ model checker [40] and adherence to the target memory consistency model (*safety*) via litmus tests [3].

To demonstrate SYNAPSE's practicality, we evaluate four scenarios: a Symmetric Multiprocessing (SMP) setup to evaluate various multi-level cache hierarchies; a NUMA layout to explore off-core protocols and inclusion policies; a multi-host CXL fabric to study different interconnect protocols over high-latency fabrics; and a heterogeneous chiplet-based SoCs system to assess the impact of diverse write policies on performance. In our evaluation, SYNAPSE correctly synthesizes controllers for all scenarios. Targeted litmus tests verify weak memory consistency, while the synthesized models perform within 1.0% of handwritten gem5 implementations across 30 applications from the PARSEC [1], Phoenix [72], and Splash-4 [31] suites.

In summary, the contributions of this paper are as follows:

- We propose a decomposition of cache coherence controller into three orthogonal constituents—protocol, topology, and policies—and show that this decoupling enables specifying controller behavior independently at a convenient abstraction level.
- We present SYNAPSE, a methodology grounded in a novel abstract cache controller model and a generalized synthesis algorithm. By fusing protocol-agnostic building blocks via a semantic mapping, it enables the automated derivation of complex controller FSMs for any coherence protocol and system topology.
- We develop a synthesis pipeline that realizes SYNAPSE by taking a declarative CCP specification and a system topology as input and automatically generating correct cache-coherence controllers for arbitrarily complex cache-coherent systems.
- We evaluate SYNAPSE on four representative architecture scenarios—SMP, NUMA, multi-host CXL, and heterogeneous chiplets—across multiple protocols and benchmark suites, demonstrating performance parity with handwritten implementations, and the utility of automated design-space exploration.

II. Background and Motivation

A. Cache Coherence Protocols (CCPs)

In a nutshell, CCPs are state machines that manage read and write permissions for individual cache lines. They define the transitions and actions required to satisfy requests from processing cores and ensure that data shared by multiple cores remains consistent. The MESI protocol and its variants are widely adopted for their intuitive programming model—enforcing SWMR invariants—and favorable performance characteristics [8], [36], [44]. In this work, we focus on directory-based protocols for their superior scalability compared to snooping-based approaches.

While programmers benefit from hardware-managed data consistency, designing and implementing CCPs is notoriously difficult due to their complexity and central role in a SoC. Fig. 1a illustrates a typical store operation in a simple 1-level cache hierarchy with MESI. Before writing new data to a cache line ①, the CCP must ensure no other cache holds a copy. The cache controller, therefore, sends a *GetM* request to the directory ②, which maintains a global view of sharers. The directory replies with the number of peer caches holding copies (two in this example, *Ack=2*) and issues invalidation messages to them ③. After invalidation, the peer caches acknowledge the request ④. Once all acknowledgments are received, the initiating cache upgrades the line's permission from S (shared) to M (modified) and completes the write ⑤.

As multiple cores may simultaneously request conflicting permissions, designers must carefully handle race conditions to maintain correctness. In the example, race points at cache 0 and cache N (red dots) require the caches to process invalidations before other forwarded requests from the directory to preserve

the ordering decided by the directory (yellow linearization point). As system complexity grows, these challenges multiply: Fig. 1b shows that the same write in a realistic 3-level hierarchy requires significantly more messages, doubles the number of race points, and makes manual implementations increasingly prone to subtle, non-deterministic bugs.

Beyond correctness, CCPs must also be highly efficient since they sit at the core of the SoC, connecting all components across the entire chip. A functionally correct but inefficient protocol may create performance bottlenecks that render it impractical. For example, the E state in MESI provides no additional functional benefit compared to MSI—both protocols offer identical SWMR guarantees. However, the E state allows caches to silently upgrade from read to write without directory communication, reducing coherence traffic, improving performance, and lowering energy consumption [46]. This optimization comes at the cost of substantial implementation complexity, adding tens of transient states and hundreds of transitions. Consequently, MSI is rarely used in practice despite its simpler and less error-prone design [8], [11], [44].

The MESI example illustrates a performance-correctness trade-off within the SWMR model. However, for massively parallel accelerators such as GPUs, the SWMR model itself becomes the bottleneck: the invalidate-on-write invariant does not scale to the thousands of threads modifying data concurrently [63]. These devices instead employ Release Consistency (RC) protocols, where coherence is managed explicitly via *release* and *acquire* operations rather than implicitly on every memory access. On a store-release, all dirty cache lines are written back and made visible to other cores; on a load-acquire, the cache self-invalidates and fetches a fresh copy from lower memory levels, which require only two stable states: Valid and Invalid.

Takeaway. Designing CCPs is highly complex; they must be both **correct and performant**—either alone is insufficient.

B. Cache Coherence in Modern Architectures

The challenges in designing CCPs are amplified in modern architectures [35], which have evolved from uniform multicore architectures to highly diverse and complex systems. Fig. 2 illustrates the four typical scenarios where CCPs are deployed today. The conventional case is Symmetric Multiprocessing (SMP) (Scenario S1), where a single CCP connects a set of homogeneous cores. Non-Uniform Memory Access (NUMA) (Scenario S2) extends this by combining multiple SMP chips, each with their own local memory, into a larger *logical* server CPU. Multi-host systems (Scenario S3), such as those based on CXL or NVLink [22], go further by allowing multiple SMP or NUMA nodes to share a common memory pool cache-coherently [38]. Finally, Scenario S4 depicts a heterogeneous chiplet-based SoCs system in which CPUs are integrated with various accelerators, such as GPUs, NPUs, or DSPs, with the CCP serving as the common substrate for seamless integration. Coherence may be enforced through proprietary protocols tightly coupled to the SoC fabric (e.g., Intel UPI, Arm CHI) or through open standards such as CXL that extend coherence across nodes to combine distinct accelerator and CPU nodes [22], [37].

While the cache-coherent layer provides a convenient abstraction that makes all these diverse architectures easy to use, the CCP becomes a critical design bottleneck: getting it right is essential to the overall system's correctness, performance, scalability, and

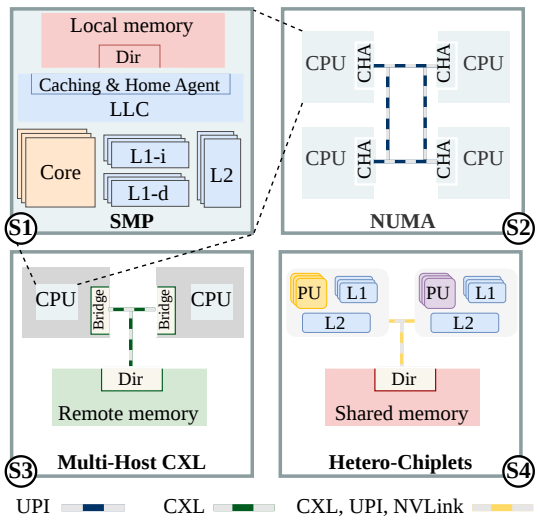


Fig. 2: Four typical cache coherence scenarios in modern architectures.

energy efficiency, yet the increasing architectural diversity makes its design significantly more complex. Heterogeneous accelerators like GPUs exhibit completely different memory access patterns and memory consistency models compared to CPUs, necessitating distinct coherence strategies. Furthermore, chiplet-based designs and advanced off-chip coherent interconnects like CXL introduce substantial variance in access latencies—a last-level cache miss may require anywhere from 80ns to 400ns depending on whether it is served from local DRAM on the same chiplet, a neighboring chiplet, or remote DRAM on another server node [59]. Developers thus face challenges similar to those encountered during the UMA-to-NUMA transition [14], [33], where CCPs required significant reworking to hide longer remote memory access latencies [17], [39], but at a greater scale and with greater complexity. Using a single protocol across all device types is unrealistic and will either create performance bottlenecks or result in power-hungry and inefficient implementations.

These challenges are recognized by recent industrial interconnect standards, which provide mechanisms to configure coherence traffic for diverse device types. For example, ARM's CHI [58] protocol and CXL [22] offer various features such as network telemetry messages, pre-fetching requests, bulk-data transfer options, and other configurable parameters that can be tailored for different compute and memory units. However, manually identifying the optimal configuration for these rapidly evolving and increasingly complex architectures is time-consuming and error-prone, making comprehensive design space exploration practically infeasible.

Takeaway. The diversity of modern hardware architectures creates an enormous design space for CCP optimization. Navigating this space requires automated approaches to enable rapid prototyping and comprehensive design space exploration.

C. The Case for Architecture-Aware CCP Synthesis

Designing optimal CCPs for increasingly complex and diverse architectures is a challenging and time-consuming task. Our work therefore seeks to answer the core question:

Research question: *How can we develop a generic*

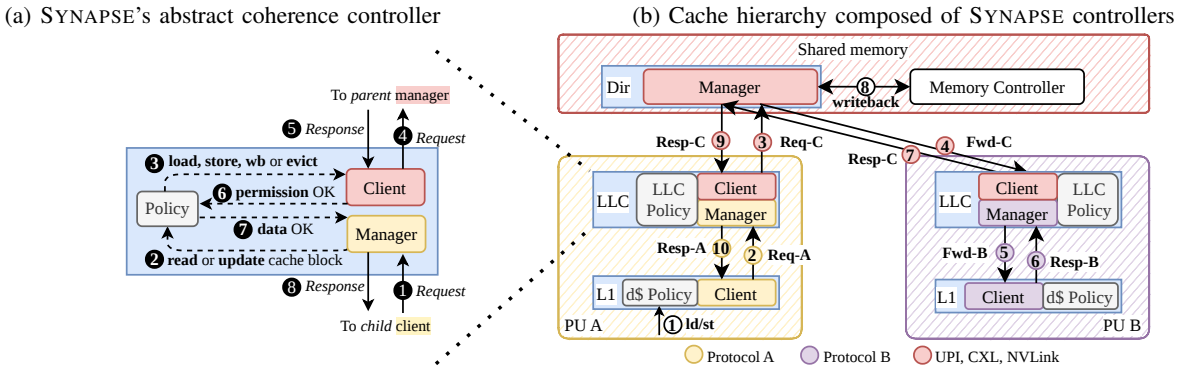


Fig. 3: Overview of SYNAPSE. Cache hierarchies consist of abstract coherence controllers, which are specialized and generically synthesized into any concrete controller state machine within the hierarchy. The abstraction decouples protocol-specific *client* & *manager* to maintain coherence up and down the hierarchy, and a protocol-agnostic *caching policy* that governs data movements.

methodology to automatically synthesize coherence controllers for arbitrarily complex cache-coherence configurations?

Such a methodology should enable rapid prototyping and large-scale design-space exploration across a wide range of architectures. To this end, it must fulfill four key requirements:

Req. 1: Balance generality and specificity. The methodology must be generic enough to capture the diversity of modern architectures, by accommodating different CCP families (e.g., SWMR and RC protocols) and the four common coherence scenarios described in Sec. II-B. At the same time, it must be specific and provide fine-grain control in the synthesis to tailor high-performance controllers for a given architecture, by supporting interchangeable caching policies.

Req. 2: Full automation. To avoid time-consuming and error-prone manual designs, the methodology must automatically synthesize all components of the coherent network without manual intervention.

Req. 3: Correctness validation. While automation removes most manual errors, integrating formal verification is essential to ensure correctness in corner cases that may be missed during design space exploration, catching subtle liveness and safety issues early.

Req. 4: Performance evaluation. The methodology must be integrated with a performance evaluation framework. This is paramount for comprehensive design space exploration, as it allows developers to validate the practical effectiveness of synthesized protocols. A CCP may be functionally correct, but without a performance model, its real-world utility cannot be accurately assessed.

D. Research Gap

While substantial work exists in both CCP synthesis and performance evaluation, no existing technique satisfies all four requirements simultaneously. Performance evaluation frameworks like gem5 [57] provide rich simulation models and infrastructure for detailed performance analysis, including a dedicated DSL (SLICC) for cache coherence modeling. However, SLICC operates at a low abstraction level, requiring developers to define complete state machines, transient states, messages, and transitions for all network components. For instance, implementing a MESI protocol for a two-level cache hierarchy requires over 2,500 lines of code to define directory, L1, and L2 cache controllers. Moreover, architectural changes such as adding a third cache level necessitate creating completely new variants and rewriting the controller models.

Similarly, TileLink [76] is an open, chip-scale interconnect standard that provides coherent shared-memory access across heterogeneous SoC components. But like SLICC, its coherence specification operates at a low abstraction level, requiring developers to manually define complete protocol state machines.

Conversely, automated synthesis tools like ProtoGen [68], HieraGen [70], and HeteroGen [69] enable users to specify cache coherence protocols on a high level in the form of atomic operations and stable states, which a compiler takes to synthesize concurrent state machines for coherence controllers, including all transient states. Unfortunately, these tools *only* target formal verification, emitting models for model checkers like Murφ to validate safety and liveness properties. Hereby, they make various simplifications that preclude performance evaluation: they synthesize only L1 data cache and directory controllers, and lack in architectural fidelity. For instance, they do not support multi-level hierarchies, instruction caches, or different write or inclusion policies. While vCXLGen [52] extends this line of work to support synthesis of performance models, it focuses solely on CXL-based multi-host coherence, omitting the other scenarios and sharing the same limitations in architectural fidelity.

Takeaway. Prior work has addressed protocol design, verification, and performance as independent problems. Existing tools are either manual and architecturally detailed or automated but limited to CCP verification or specific coherence scenarios. The key missing element is a generic methodology that spans all modern architectural scenarios, while simultaneously fulfilling the requirements of full automation, correctness validation, and performance integration.

III. SYNAPSE OVERVIEW

We introduce SYNAPSE, a generic and systematic methodology to synthesize state machines of complex and practical coherent cache hierarchies. Fig. 3 shows the idea behind SYNAPSE. It treats coherence protocol, caching policy, and topology as independent abstractions, thereby enabling modular cache and memory system designs across a range of modern hardware-coherent architectures.

Abstract coherence controller (§ IV). At the center of the SYNAPSE methodology is the *abstract* coherence controller. It unifies 3 sub-components: *coherence client*, *manager*, and *caching policy*. The *coherence manager* receives requests from controllers at the previous level in the hierarchy and keeps track of cache line state between them, like a directory. The *coherence client* sends requests up in the hierarchy to retrieve cache line data or permissions, like a cache.

Fig. 3a illustrates its generic logic. While the *manager* orchestrates coherence between child clients, the *caching policy* intercepts data accesses (❷-❼) and determines the new state of cached data by propagating the right *client* coherence transactions (❸-❻) to its parent. Hence, it enforces caching policies while maintaining coherence across hierarchies.

Controller synthesis (§ V). SYNAPSE practically realizes this abstraction via its generic synthesis algorithm, which generates the concrete finite state machines (FSMs) of any coherence controller in a hierarchy. In essence, the algorithm can synthesize any coherence controller (e.g., L1-d/i, L2, or LLC) with the same client/manager specifications, simply by exchanging the controller's policy.

In Fig. 3b, SYNAPSE synthesizes specialized LLC slices that enable coherent shared memory through the interconnect protocol [52], simply by exchanging the LLC's client from the host's protocol to the interconnect's. To achieve this, the algorithm first establishes a semantic mapping between the individual FSMs of the 3 sub-components in SYNAPSE controllers by bringing them to a common *protocol-agnostic* abstraction level, to then systematically fuse the coherence transactions of *manager* and *client* as constrained by the chosen *policies*.

Synthesis pipeline (§ VI). To demonstrate SYNAPSE's potential, we develop an end-to-end pipeline that flexibly generates simulation models of complete cache hierarchies from independent specifications of topologies, protocols, and policies. It parses the 3 inputs, initializes an IR of their individual FSMs, and connects them to form a topology of abstract controllers like in Fig. 3b. The synthesis algorithm then generates the concrete FSMs for all controllers in the specified topology, by labeling the client and manager FSMs with protocol-agnostic semantics and merging them with their chosen policy. The main back end generates coherence models for *performance* simulations, but we also develop a second *validation* back end to generate model-checking system models. It ensures the *correctness* of the generated controller FSMs, i.e., that they are deadlock-free and introduce no memory consistency violation.

Automatically synthesized controllers (§ VII). Using the synthesis pipeline, SYNAPSE automatically generates controllers for the four scenarios described in Sec. II-B: SMP (S1), NUMA (S2), Multi-Host CXL (S3), and Hetero-Chiplets (S4), spanning from single-node shared-memory to disaggregated and heterogeneous architectures.

IV. GENERALIZED CACHE COHERENCE

In this section, we present our system model (summarized in Fig. 3b) and define the abstract coherence controller, which enables the decoupling of topologies, coherence protocols, and caching policies.

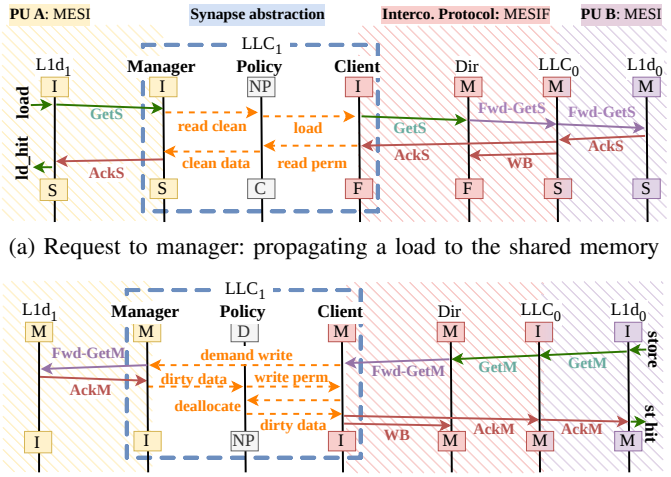


Fig. 4: Conceptual internal operations of SYNAPSE controllers to propagate coherence transactions across domains.

A. System Model

Caching architectures. The four sample systems depicted in Fig. 2 are all hierarchical cache systems. These hierarchies are composed of varied cache controllers (L1-i/d, L2, LLC, Bridge, CHA, Dir), arranged to form specific topologies. Fig. 3b presents our system model of the above cache architectures. Each controller tracks the coherence state of cache blocks held by its direct children (if any), as well as blocks in its own cache (if any). Every controller independently allocates, updates, or replaces its data blocks with respect to its *caching policy*.

Heterogeneous cache hierarchies. Multi-cluster systems such as $S2$, $S3$, $S4$ in Fig. 2 may resolve coherence through a stack of heterogeneous protocols. They leverage multiple parallel *intra*-cluster protocols and a common *interconnect* protocol. As depicted in Fig. 3b, we model them as parallel homogeneous sub-hierarchies, interconnected through specialized heterogeneous controllers (CHA, Bridge). In practice these are analog to LLC slices, with extra logic to maintain coherence with parent and peers over the interconnect protocol.

B. Abstract Coherence Controllers

Although cache hierarchies are seemingly built from a common set of controllers, their design remains non-modular: each individual controller adheres to a coherence protocol tailored jointly to the topology and the caching policy selected by the platform's architects.

The controller abstraction. To enable a modular design of cache hierarchies, we devise a single *generic* and *abstract* coherence controller that is freely composable into any topology. It is *abstract* in that it can be instantiated for any coherence protocol, and *generic* in that it can incarnate any concrete controller within the hierarchy.

Building blocks. As seen in Fig. 3b, each controller mediates coherence between its children and parent (if any). In essence, they incarnate a *directory* and *cache* simultaneously. From this observation, we realize our controller abstraction by combining 3 sub-components:

- **Coherence manager:** tracks and orchestrates cache line access between its child controllers (client), i.e., a *sub-directory*.
- **Coherence client:** requests cache line access or data from a parent controller (manager), i.e., an *intermediate cache*.
- **Caching policy:** defines when the controller allocates, updates, flushes or replaces its cache blocks.

Note that not all 3 sub-components need to be instantiated, as shown for leaf caches (L1) and the root directory (Dir) in Fig. 3b.

Propagation principle. Abstract controllers coordinate coherence transactions between their *manager* and *client*, while enforcing their *caching policy*. In detail, from Fig. 3b intermediate controllers process requests from their children (2) or parent (4), and may propagate them in turn as forwarded requests (3, 5). Requests need to propagate in the hierarchy when the state of at least one sub-component can not fulfill the request. This propagation happens *synchronously*.

Example. We illustrate the operating principle with concrete examples in Fig. 4. (a) *Request to manager:* When a request is observed by a manager (`GetS`), it is propagated up the hierarchy by the client (`GetS`) if it is a *data miss* to the policy or a *permission miss* to the *client*. In Fig. 4a, the policy is *inclusive* as it transitions from a *non-present* (*NP*) cache block to *clean* (*C*). (b) *Request to client:* When a request is observed by a client (`Fwd-GetM`), it is propagated down the hierarchy by the manager (`Fwd-GetM`) if it is a *data downgrade* to the policy or a *permission downgrade* to the *client*. In Fig. 4b, because the client must transition to *Invalid* (I) and yield its data and permissions, it reclaims both from the manager then deallocates its cache block (NP).

V. The Synthesis Algorithm

In this section, we present our generalized method to construct coherence controllers of arbitrary protocols and caching policies.

Intuition. We synthesize any coherence controller by fusing the finite state machines (FSMs) of a manager, policy, and client. FSMs of the coherence manager and client define all valid protocol transactions, while the policy defines controller instances (e.g., L1-i/d, LLC). The end goal is to produce a *unified* FSM with complete transition sequences that propagate across hierarchies, as seen in Fig. 4. In brief, our algorithm identifies and composes transition sequences from the manager and client FSMs, in a way specific to a controller's policy.

Overview. To understand our algorithm (Alg. 1), let's first decompose the complete transition sequences from the previous examples:

• *Request from child client.* In Fig. 4a, the `GetS` request must propagate up because of a *data* and *permission miss*. We identify it from the manager FSM, as the transaction `I+GetS` returns new *read permissions* with *clean*, *coherent data*. To actually propagate it, we intuitively *ask* the policy which client state can fulfill this request, (line 3 Alg. 1) with `policy.match-rule()`. In the client FSM, it finds the transaction `I+GetS` to reach this final client state, and synchronously starts it with `client.process()` (line 6). In that example, the caching policy is inclusive, so the LLC must allocate a new cache block (line 4) before, as the policy may need to first trigger replacement (eviction of another block).

Algorithm 1: Synthesis of coherence controllers

```
1  ∀(manager-state, client-state) ∈ manager.States × client.States
2  when receiving Request FROM child client:
       /* Policies steer the final cache state */
3      client-final-state = policy.match-rule(client-state, Request);
       /* Policy: run pre-rule */
4      if client-final-state<Data> > P then cache.allocate-block();
5      policy.start-rule(client-state, Request);
       /* Execute
          client transaction(s) until the Request
          is fulfilled and final state is reached */
6      client.process(client-state, Request, client-final-state);
       /* Policy: run post-rule */
7      policy.end-rule(client-state, Request);
8      if client-final-state<Data> == NP then cache.deallocate-block();
       /* Execute manager transaction */
9      manager.process(manager-state, Request);
10 when receiving Request FROM parent manager:
       /* Policy not needed for backward requests,
          client steers the final cache state */
11     client-final-state = client.simulate(state, Request);
12     if client-final-state < client-state then
           /* Downgrade
              child caches first by re-submitting
              the request to the manager */
13         manager.process(manager-state, Request, client-finale-state);
14     end
15     if client-final-state<Data> == NP then cache.deallocate-block();
       /* Execute client transaction */
16     client.process(state, Request);
```

Lastly, as the client returns with $\xrightarrow{AckS}$(F) since a sharer cache still exists, (line 9) the manager may only execute its transaction (I) $\xrightarrow{GetS}$ (S) as it cannot return read-exclusive permissions (`E`).

• *Request from parent manager.* In Fig. 4b, the `Fwd-GetM` requests must propagate down as it downgrades data and permissions of the client. We detect in the client FSM that its protocol imposes the transition (M) $\xrightarrow{FwdM}$ (I), with `client.simulate()` (line 11), and that it yields its *write permission* and *dirty*, *coherent data*. In the manager FSM, we find that the transaction `M+FwdM` produces the same effects, and we synchronously start it with `manager.process()` (line 13). Only after it returned, the client runs its original transaction to service the `Fwd-GetM` request (line 16). Note that policies do not customize data movements, as forwarded requests already impose clients' data state.

Challenges. From this operational intuition, we make two remarks: (1) Synapse treats coherence transactions as *black boxes* and does not modify their internal behavior during composition; (2) The algorithm, however, requires equivalent semantics between the states and requests of the manager, policy, and client to identify which requests propagate and how. Since the client and manager FSMs come from concrete protocol specifications while the policy remains protocol-agnostic, the key challenge is establishing a semantic mapping between them.

A. Abstract Protocol State Machine

To establish this semantic mapping across all input FSMs, we define a generic characterization of controller states and requests, that allows us to bring all FSMs to a common *protocol-agnostic* abstraction level.

Universal semantics. We build our abstract model on key properties universally found in coherence protocols: *operation permission*, *data state*, and *coherence state*, that we use to *label*

the states and requests of concrete state machines. We define them as follows:

- **Permission**: $Op \in \{None,\ Read,\ Write,\ Read\text{-}Write\}$
- **Data state**: $Data \in \{Not\ Present,\ Clean,\ Dirty\}$
- **Coherence**: $Coh \in \{co,\ nco\}$

Abstract states and messages. We define the *states* of abstract controllers and abstract *messages* as a tuple of the previous properties:

- **Abstract state**: $S\langle Op,\ Data,\ Coh\rangle$
- **Abstract message**: $M\langle Op,\ Data,\ Coh\rangle$

These 3 properties are universal concepts of protocols, enabling us to lift concrete coherence manager and client to this representation. For instance, the M state of $MESI$ is labeled with $S\langle$*Rd/Wr*, *dirty*, *co*$\rangle$; or the V state in the RCC protocol is labeled with $S\langle$*Rd/Wr*, *dirty*, *nco*$\rangle$.

These labels also create a semantically rich characterization for protocol messages (cache requests and core events/accesses alike), which we leverage to translate request semantics across heterogeneous protocols. As a concrete example, a MESI *dirty-replacement* request (`PutM`) and an RCC *release* request (`PutR`) both are labeled $\langle Wr,\ dirty,\ co\rangle$, thus the algorithm propagates them similarly.

Abstract caching policies. We define policies as a set of *rules* that together specify controllers' abstract state changes on messages:

- **Rule:** $initialState + Message \rightarrow finalState$

For instance, a *load miss* of an inclusive L1 is specified as:

$$S\langle _,\ NP,\ _\rangle\ +\ M\langle Rd,\ none,\ _\rangle \rightarrow S\langle Rd,\ P,\ _\rangle$$

Informally, this example denotes that when receiving a *demand-read* with a *not present* (NP) cache block, the policy must transition the controller to a final state with *read permissions* and *present* (P) data (either clean or dirty). If unspecified with _, a property matches any values.

Crucially, this example also captures how our universal semantics allows policies to easily customize the propagation logic of controllers, by enabling them to find semantically matching transitions in the client FSM when invoking `policy.match-rule()` (line 3 Alg. 1).

A set of such *rules* form a caching policy, which simultaneously specifies: (a) the concrete operations that a controller can perform, and (b) the data management rules associated with each operation. For instance, an L1-i policy will not specify any rules containing *Wr* operations, and a hierarchical controller policy (L2, LLC) will implement write-back or write-through, and inclusive or exclusive behaviors through dedicated rules.

B. Labeling Concrete FSMs

To construct our abstract state machines, we statically analyze each input protocol in isolation. We generate all communication traces between client and manager, and collect all distinct FSM transactions that form a complete transition sequence from a stable state and core access (load, store) to final states where the cache obtains its requested permissions. We then traverse those communication traces to label the concrete FSM states and messages with our abstract model properties $\langle$*Op, Data, Coh*$\rangle$. For that, we start from the `read` and `write` statements present in the FSM to detect states where cache blocks are read or updated by the client and manager, and then we propagate the labels to other messages and states as we traverse the communication traces. Our algorithm then finds semantically matching transitions when it propagates requests by simply comparing the $\langle$*Op, Data, Coh*$\rangle$ labels, intuitively repeating the semantically equivalent access from the initial requestor.

VI. The Synapse Synthesis Pipeline

In this section, we present Synapse's synthesis pipeline, an end-to-end compiler to generate concrete controller implementations of entire hierarchies from protocol, topology, and policy specifications.

Workflow. The compiler pipeline is organized in the classic three stages: *Front*, *Middle*, and *Back end*, as shown in Fig. 5. From the three independent input specifications, brought together as the Synapse-DSL, it synthesizes the controller state machines and formally verifies the correctness of their intermediate representation (IR), and then generates coherence controller models through a dedicated back end for gem5's SLICC code, enabling direct performance evaluation and design space exploration in simulations.

We covered the synthesis algorithm in the previous section; next, we detail its integration with the front and back ends.

A. Synapse DSL

Topologies. We specify cache hierarchies and hardware parameters of the simulated system with a simple declarative YAML-like syntax. Topologies are a tree of *clusters*, which comprise an *interconnect* (star, mesh, point-to-point) with its link latency, flit size, and link bandwidth; and an enumeration of *nodes* (either *core*, *memory*, *cache controller* or *sub-cluster*). Each node includes hardware parameters relevant to the simulated system (e.g., core type, cache capacity, mem. type, capacity, or latency). In addition, cache controller nodes hold references to specifications of a coherence manager, client and policy.

Coherence protocols. We rely on existing machine-readable stable state protocol (SSP) specifications [68] of concrete 1-level CCP, which concisely specify protocols (e.g., 650 SLOC for CXL.mem). They specify state machines of *cache* and *directory*, which we simply map to our coherence *client* and *manager*, respectively.

Policies. We repurpose the SSP specification format for our policies, by using a subset of its syntax and our own symbol conventions for our model of abstract states and messages. For ease of use, we implement a library of policies that contains all base controllers (L1d/i, L2, LLC, Dir), as well as common *write* (write-back, write-through) and *inclusion* policy (inclusive, non-inclusive, exclusive).

B. Controller Synthesis

The DSL parser initializes the component IRs, representing each controller FSM as a directed graph. Transition objects also contain an abstract syntax tree (AST) of the controller action specified in the SSP. The middle end then processes these component IRs in three steps to produce concrete controller FSMs for the full hierarchy.

Labeling. We run the static analysis from §V-B on each protocol FSM in isolation, traversing communication traces between

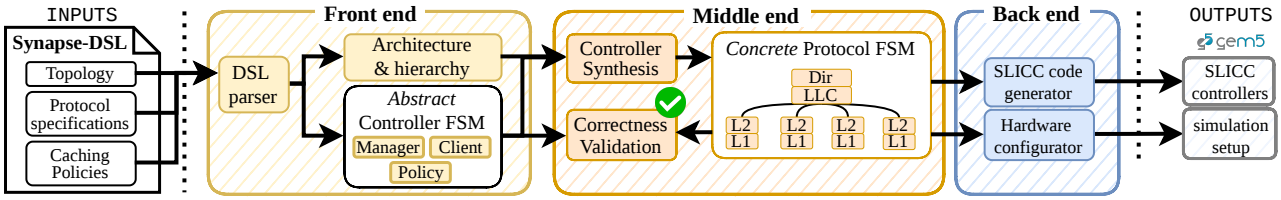


Fig. 5: The SYNAPSE pipeline. It takes as input coherence protocol, caching policy, and topology specifications, to generate all SLICC controllers in the hierarchy, enabling performance evaluation of CCPs in gem5 simulations.

```
1 Rule( S(_,NP,_), M(Rd,None,_), S(Rd,P,_) )
```

Fig. 6: DSL excerpt from `LLC-WriteBack-Inclusive` specs.

```
1  Architecture cache {
2    Stable{I, S, F, E, M}
3    //Transaction: Start, Access, Final
4    Process(I, load, State){
5      msg = Request(GetS, ID, directory.ID); //Src, Dest
6      req.send(msg);
7      await {
8        when GetF_Ack:
9          cl=GetF_Ack.cl; load; State=F; break;
10       when GetM_Ack:
11         cl=GetM_Ack.cl; load; State=E; break;
12     }
13   }
```

Fig. 7: DSL excerpt from `MESIF` specs.

cache and directory to label every *state* and *transition* of the *client* and *manager* IR with an abstract $\langle Op, Data, Coh \rangle$ tuple.

Synthesis. Alg. 1 runs for each controller instance in the topology, fusing client and manager transition sequences as constrained by the controller's policy rules into a concrete FSM IR.

Topology resolution. We group concrete controller IRs into clusters following the communication structure of the hierarchy, and designate one *manager* and multiple *client* controllers in each cluster. As a controller may be a manager in one cluster and a client in another, the list of clusters captures the hierarchy's tiers. We pass it to the back end for virtual channel wiring and simulation setup.

Example. We illustrate the pipeline through the synthesis of the LLC transition in Fig. 4a. In this example, the LLC receives a demand-read (`GetS`) request from an L1d, with its manager and client components in state `I` and a missing cache block.

The static analysis first labels the states and messages of the parsed FSMs, labeling `I` with $S\langle None, NP, co \rangle$ and `GetS` with $M\langle Rd, None, _ \rangle$. These match the inclusive LLC load miss rule, shown in Fig. 6, thus the policy assigns $S\langle Rd, P, _ \rangle$ as the abstract client final state when running `policy.match-rule()` (line 3 Alg. 1). The call to `client.process()` (line 6) then finds the transaction `I+load` with equivalent labels in the client FSM, shown in Fig. 7, and executes it until it reaches state `F` or `E`. The synthesis algorithm keeps both end paths in the final FSM, as labels from both end states – `F`: $S\langle Rd, clean, co \rangle$, and `E`: $S\langle Rd/Wr, clean, co \rangle$ – are compatible with the abstract end state of the policy. If no transaction with equivalent labels exists, the policy and protocol are incompatible and the synthesis algorithm raises an exception. This is the case, for instance, when pairing an exclusive policy with a protocol lacking a forwarding mechanism.

C. SLICC Code Generator

The gem5 back ends output SLICC code for each cache controller in the topology, along with the simulation setup code that configures the system and hardware as in the topology specifications. Lowering our IR of the controller FSMs to the SLICC dialect is straightforward, as SLICC implements FSMs of coherence controllers in an event-driven language. Thus, we only traverse our controllers' FSMs IR, group messages by virtual channel, and generate protocol message handlers for each coherence state. Interestingly, thanks to policies, our FSM IR already models data management in cache controllers, so we can easily map it to dynamic allocations of the *cache block entry* in SLICC. Similarly, SLICC requires states to list their permissions, for gem5 introspection, which we deduce from our abstract FSM permissions labels (None, Rd, Wr, Rd/Wr) that we used during synthesis.

Back end extensibility. To date, Synapse features only a gem5 back end, as its primary goal is rapid architectural design-space exploration. However, since the synthesis algorithm operates at the specification level, the front end, middle end and the DSL are back-end agnostic. Future work could therefore extend Synapse with an additional back end, for instance, to target hardware synthesis.

D. Correctness Validation

We validate the generated FSMs at the IR level by using explicit state model checking, following the methodology established by previous works on CCPs [52], [53]. A separate SYNAPSE back end automatically generates multiple system models for the Murφ model checker [40] that represent the specified system configuration. We exhaustively explore these models to verify *liveness* and *safety* properties.

Liveness validation. We verify that the generated FSMs are *deadlock-free*, i.e., the system models can always transition to a different state of the modeled system. We perform *unrestricted* exploration of the system models and ensure that all reachable states can transition to a different state [52]. For complex system models, we avoid a state space explosion by generating multiple system models variants: each with only 1 cluster fully modeled at a time, and with 3 cores at most.

Safety validation. We verify that the generated FSMs do not violate the memory consistency model (MCM) of the simulated cores, i.e., the system models produce disallowed memory reorderings. For that, we use litmus tests: small multi-threaded programs whose outcomes characterize MCM-allowed or -disallowed re-orderings of memory operations. Like in [53],

Feature \ Scenario	S1	S2	S3	S4
CPU				
Cores config.	2 GHz, x86, 8-wide OoO, 192 ROB			
Cores #	[8-30]	[8-30]	18	64
L1-D	64 KiB	64 KiB	256 KiB	64 KiB
L1-I	64 KiB	64 KiB	256 KiB	64 KiB
L2	1 MiB	-	-	-
LLC	8 MiB	2 MiB	8 MiB	2 MiB
Interconnect				
SMP *(S1)*	p2p, 32B/flit, 1c router, 5 ns link			
NUMA *(S2)*	mesh, 80B/flit, 1c router, 10 ns link			
Multi-Host CXL *(S3)*	star, 256B/flit, 1c router, 70 ns link			
Hetero-Chiplets *(S4)*	star, 256B/flit, 1c router, 70 ns link			
Memory				
Local (*S1-2*)	DDR5, 4400 MHz, 8 GiB, 1-ch, 10 ns			
Rem/Shr (*S3-4*)	DDR5, 4400 MHz, 8 GiB, 8-ch, 10 ns			

TABLE I: Simulated system parameters.

we generate Murφ system models whose exploration is *restricted* by the instructions of a litmus test. That is, threads of the litmus test are mapped to L1 controllers, which issue memory operations corresponding to litmus test instructions. We generate separate Murφ models per possible thread-to-L1 assignment. The model checker then explores *all potential memory re-orderings and thread interleavings*, ensuring no MCM violations, i.e., no forbidden test outcome, is reached. We obtain litmus tests from the herd7 [3] tool, for sequential (SC) and relaxed consistency (RC) threads, including common checks like *IRIW, MP, 2+2W, CoRR1, CoRR2, LB, R, RWC, S, SB, WRC, WRW+2W, and WWC*. By default, SYNAPSE integrates with SC and RC, but users can also provide appropriate litmus tests for their MCM as SYNAPSE generates litmus test models from any custom test set.

VII. AUTOMATICALLY SYNTHESIZED COHERENCE CONTROLLERS

Based on the synthesis pipeline described in Sec. VI, SYNAPSE automatically generates concrete gem5 simulation setups for the four scenarios S1–S4 presented in Sec. II-B and shown in Figure 2. We use gem5 [57] version 23.0, leveraging its detailed memory subsystem, Ruby [56], which provides accurate modeling of cache hierarchies and coherence protocols, enabling precise performance evaluation of the synthesized controllers. All experiments are performed in gem5's system call emulation (SE) mode.

S1: SMP. SYNAPSE synthesizes the S1 controllers for a single-node SMP with multiple cores, all sharing a unified last-level cache and a single local memory managed by a centralized directory controller. Each core has a private 64 KiB L1-I and 64 KiB L1-D cache, a private 1 MiB L2, and a shared 8 MiB LLC. The main memory is modeled as a single-channel DDR5. The on-chip network uses gem5's Garnet model [15] with a point-to-point topology; link latency is set to achieve a 100 ns end-to-end memory access latency, consistent with measured SMP baselines [27], [34], [36].

S2: NUMA. For S2, SYNAPSE automatically generates 4 NUMA nodes, each with the synthesized intra-cluster configuration of S1. Each node manages its own local memory partition, with remote accesses resolved via a directory coherence protocol extended across all nodes. Inter-node communication is modeled in Garnet as a point-to-point mesh with 80 B/flit links and 10 ns link latency, configured to match measured remote memory access latencies of real NUMA systems [45], [61], [62], yielding a remote access latency under 130 ns.

S3: Multi-Host CXL. This synthesized configuration comprises two compute nodes, each with the synthesized SMP intra-cluster configuration, connected to a third *remote* storage node, emulating a disaggregated CXL memory system. Nodes are connected via a star interconnect with 256 B/flit links and 70 ns link latency, producing an end-to-end remote memory access latency of 400 ns, consistent with reported CXL measurements [59]. While PCIe-level CXL simulators exist [79], we use Garnet to model the off-chip network across all experiments, enabling a fair apples-to-apples comparison between CCPs and isolating their effects from PCIe-related differences.

S4: Hetero-Chiplets. SYNAPSE synthesizes the S4 controllers targeting the multi-Processing Unit (PU) setup from [81]: a system with 8 heterogeneous, chiplet-based SoCs, each with 8 cores (64 PUs total), connected via a coherent star fabric modeling either Intel UPI or CXL, using the remote access latencies from the NUMA and Multi-host CXL scenarios, respectively. Each PU represents an accelerator tile within its chiplet, equipped with a private 64 KiB L1-I and L1-D cache and accesses a shared 2 MiB LLC. All chiplets access a globally shared memory pool, managed by a centralized directory, over the coherent star fabric.

Each accelerator is modeled using gem5's SimpleTiming CPU model. To accurately reflect the accelerator's memory behavior, we must enforce a relaxed memory model rather than the strict consistency model CPUs default to. To achieve this, we use a custom *Sequencer*, the interface between the CPU and gem5's Ruby memory subsystem. This modified *Sequencer* implements two key operational changes. First, we remove the single-outstanding-request constraint per cache block (which is typically necessary in default CPUs to meet SWMR constraints). Second, to simulate the non-blocking store buffers typical of accelerators [77], the *Sequencer* returns immediate, asynchronous completions for memory operations that do not require immediate visibility (e.g., non-temporal and release stores). This decouples CPU execution from memory latency, while correctness is preserved by the *Sequencer*, draining all outstanding operations at explicit synchronization barriers (e.g., Acquire operations) that enforce global visibility.

Tab. I lists a summary of the simulated system parameters for each scenario.

VIII. EVALUATION

We evaluate SYNAPSE in two parts. In the first part, we validate SYNAPSE's ability to synthesize CCP controllers across a wide range of system configurations (*generality*) that are *correct* via litmus tests, and that match the performance of hand-tuned protocols already available in gem5 (*performance*) using multi-threaded benchmarks on the SMP scenario (S1). In the second part, we study three use cases to demonstrate how SYNAPSE facilitates design space exploration across realistic scenarios: (i) analyzing the effect of different off-core coherence protocols in the NUMA scenario (S2), (ii) exploring the effect of varying bridge coherence protocols in the multi-host CXL scenario (S3), and (iii) studying the impact of diverse allocation policies in the heterogeneous chiplet-based SoC scenario (S4).

	Case Studies	Controllers	On-Core	Off-Core	Write-Policy	Inclusion-Policy
S1	**SMP**	L1-i, L1-d, L2, LLC, Dir	MESI, MOESI	—	WB	Inclusive
S2	**NUMA**	+ Home-Agent	MESI	MESI, MOESI, MESIF	WB	Inclusive, Exclusive
S3	**Multi-Host CXL**	+ Bridge	MESI	MESI, CXL.mem, MOESI, MESIF	WB	Inclusive
S4	**Hetero-Chiplets**	L1-i, L1-d, LLC, Dir	MESI-RC*	MESI, CXL.mem	WB, WT	Inclusive

TABLE II: Case studies of the evaluation, with a summary of the generated controllers, protocols, and policies.

A. Generality

SYNAPSE's generic architecture enables the synthesis of a large number of design combinations. Based on the 4-use-case study presented in Sec. VII, we synthesize designs for each use case with different combinations of on-core and off-core protocols, and write-back policies.

Starting with SMP *(S1)*, we generate cache controllers for a multi-level cache hierarchy under two SWMR protocols, MESI, and MOESI. These designs demonstrate SYNAPSE's capability to synthesize homogeneous protocols that are widely adopted by CPU vendors such as Intel, AMD, and Arm [8], [11], [44].

NUMA *(S2)* extends a single SMP node to 4-sockets, introducing a tiered memory hierarchy in which remote accesses incur significantly higher latencies than local ones, typically 1.3× more [16], [25], [29]. Managing this gap requires dedicated off-core coherence protocols as well as LLC inclusion policies that minimize unnecessary remote traffic. To this end, we use SYNAPSE to synthesize three off-core protocol variants: MESI, MOESI, and MESIF, evaluated under both inclusive and exclusive LLC policies, and paired with a MESI on-core protocol.

Multi-Host CXL *(S3)* targets a disaggregated system in which multiple compute nodes share a common memory pool. Unlike NUMA, this scenario operates at substantially higher remote memory access latencies, reaching up to 400 ns [61]. SYNAPSE synthesizes MESI, MOESI, MESIF, and CXL off-core variants, allowing us to evaluate the effect of varying bridge coherence protocols across this higher-latency interconnect fabric.

Hetero-Chiplets *(S4)* models a chiplet-based SoCs where massively parallel workloads cannot afford the invalidate-on-write overhead of SWMR protocols. By synthesizing MESI-RC (a Release Consistency variant of MESI where coherence is triggered explicitly via acquire/release rather than on every memory access) as the on-core protocol paired with MESI and CXL off-core variants and by generating both write-back and write-through variants, SYNAPSE generates both write-back and write-through variants for this scenario.

Tab. II summarizes the design space evaluated with SYNAPSE, mapping the synthesized controllers, on-core and off-core protocols, and caching policies across all four architectural scenarios. To quantify the abstraction SYNAPSE provides, Tab. III reports, for each combination, the size of the DSL inputs, the number of states and transitions of the generated FSMs, and the resulting SLICC code size. Overall, these results show that Synapse synthesizes directory-based [1] controllers spanning every level of the hierarchy and a broad, heterogeneous design space, from compact protocol and policy specifications alone.

[1]Support for snooping-based protocols in Synapse is straightforward: The SSP already supports broadcast messages, so the input protocol logic can be adapted for snooping.

B. Correctness

To validate the SLICC back end, we verify that SYNAPSE-synthesized CCPs correctly enforce the host's local MCM by executing litmus tests in the gem5 simulation environment across the diverse combinations of topologies and protocols presented in §VIII-A. We exclude the Hetero-Chiplets protocols from this assertion, as the custom sequencer introduces a different memory consistency model that is neither captured by our existing litmus tests nor easily addressed without defining new ones, which we consider outside the scope of this paper.

For the correctness tests, we configure the system with ARM cores instead of x86, as ARM's weaker memory model permits more reordering to be observable without appropriate fences, making litmus tests more effective for testing cache coherence designs.

We execute five common litmus tests generated using the herd7 tool [3]: *2_2W*, *MP*, *R*, *S*, *SB*. Each test is executed 100,000 times, with threads distributed equally across the compute nodes.

Results. Across all tests, no forbidden outcomes were observed, confirming that the synthesized protocols maintain consistency by enforcing appropriate orderings.

Cross validation. To ensure that the litmus test will indeed detect forbidden outcomes and SYNAPSE does not introduce stronger memory ordering guarantees than the MCM defines, we intentionally removed all synchronization fences from the litmus tests. As expected, these modified tests produced at least one forbidden outcome in each case, serving as a control that our tests don't pass unconditionally.

C. Performance

We evaluate SYNAPSE-generated controllers for the SMP scenario (S1) against all hand-crafted CCP implementations available in gem5 for this configuration: MESI two- and three-level protocols (referred respectively as MESI-2L and MESI-3L) and the `MOESI_CMP` two-level protocol (referred to as MOESI-2L)[2], assessing whether automated synthesis incurs any execution overhead compared to manually-written protocols.

Workloads. We used 30 parallel applications drawn from established benchmark suites: 11 applications from PARSEC [1], 7 applications from Phoenix [72], 12 applications from Splash [31]. All benchmarks are executed with small input sizes to maintain tractable simulation times while preserving fundamental communication patterns.

Results. Figure 8 reports normalized execution time across all workloads for the three protocol pairs, normalized to the handwritten gem5 implementations. On average, the protocols synthesized by SYNAPSE remain within 1.0% of the handwritten protocols across all configurations: MESI-2L shows a mean difference of $+0.5\%$, MESI-3L of -1.1%, and MOESI-2L

[2]Other coherence protocol implementations included in gem5 target outdated systems or lack real-world relevance (eg., MI, MSI).

Protocols		Input size (LOC)			States/Transitions				Synth. code (LOC)
On-core	Off-core	Protoc.	Policy*	Topology*	L1	L2	LLC	Dir	
MESI	MESI	410	40±15	80±10	13/41	**	56/327	6/45	7K
MOESI	MOESI	513	40±15	80±10	17/47	**	128/699	9/52	15K
MESI	MOESI	923	40±15	80±10	12/36	45/188	83/468	9/52	19K
MESI	MESIF	1067	40±15	80±10	12/36	45/188	92/542	9/49	14K
MESI	CXL.mem	1060	40±15	80±10	12/36	45/188	64/358	12/74	14K

*Policy and topology input size varies with the specification used across the use-cases. **for 2-level on-core hierarchy

TABLE III: List of input protocol combinations, with input sizes, generated states and transitions, and resulting SLICC code size.

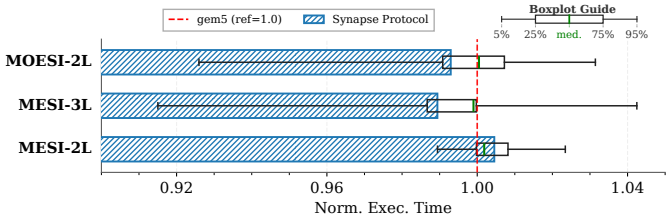


Fig. 8: Mean normalized execution time of Synapse-synthesized vs. gem5 manual (library) implementations across 30 workloads.

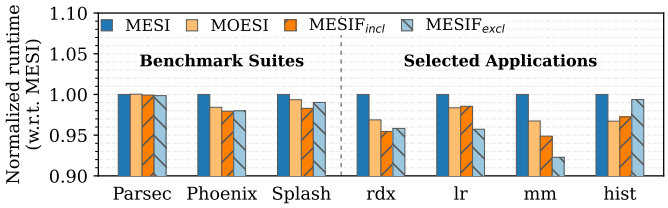


Fig. 9: Normalized execution time of 4 off-core protocol configurations in the NUMA scenario.

of -0.7%. The per-benchmark variance is relatively contained across the evaluated configurations. Specifically, considering the 5th and 95th percentiles, 90% of the benchmarks report execution times ranging between -1.1% and $+2.3\%$ for MESI-2L, between -8.5% and $+4.2\%$ for MESI-3L, and between -7.4% and $+3.1\%$ for MOESI-2L.

Analysis. We perform a root-cause analysis to understand the performance differences among outliers and find that they are due to subtle differences in the implementation of *unblock* messages. In gem5 handwritten SLICC implementations changing, the access permissions of a cache line (e.g., upgrading to write or downgrading to read access) requires the requesting node to send an explicit unblock message to the directory, adding a network round-trip per transition. Synapse eliminates this step entirely in MESI; in MOESI, we introduce unblock messages only when the directory holds the line in O state. This approach significantly reduces network traffic, and unlike gem5, it allows the LLC and directory to immediately process subsequent requests to the same cache line, rather than stalling them until the unblock is received. The overall overhead observed in Synapse is attributable to minor differences in response channel organization between Synapse and gem5's handwritten implementation.

Takeaway. The results of the first part of the evaluation demonstrate that Synapse synthesizes *correct* CCP controllers across a wide range of system configurations and protocol variants, as confirmed by litmus tests. Performance comparisons further show that the synthesized controllers closely match hand-tuned gem5 implementations, with an average difference within 1.0%. In the second part, we demonstrate Synapse's practicality through three use cases targeting design space exploration across increasingly complex architectures.

D. Use-case #1: Off-Core Protocols in NUMA Scenario

In the first use case, we use Synapse to synthesize and evaluate four off-core protocol variants in the NUMA scenario (S2): MESI, MOESI, and two versions of MESIF, all paired with a fixed MESI on-core protocol. For MESIF, we explore the impact of different caching policies by synthesizing both inclusive and exclusive LLC variants. We compare these configurations to understand how the choice of protocol, together with LLC inclusion policies, affects memory bandwidth, interconnect overhead, and execution time across diverse multi-threaded workloads.

Workloads. We used the same set of multi-threaded applications as in the SMP performance evaluation (§ VIII-C).

Results. Figure 9 reports normalized execution time for MESIF$_{excl}$, MESIF$_{incl}$, and MOESI, all normalized to the MESI baseline. On average across the three protocol extensions, introducing the `F` and `O` states yields a 0.9% reduction in execution time compared to the MESI baseline. MESIF$_{excl}$ outperforms the baseline by an average of 0.8%, with per-benchmark variations ranging from a 7.7% reduction to a 3.3% increase. MESIF$_{incl}$ achieves an average speedup of 1.2%, ranging from 5.9% faster to 0.4% slower. Similarly, MOESI achieves an average speedup of 0.6% over the baseline, with execution times ranging from 3.6% faster to 3.2% slower.

Analysis. Compared to MESI, the three alternative protocols reduce memory read bandwidth by enabling cache lines to be served from peer caches rather than main memory. On average, MESIF$_{excl}$ reduces memory read bandwidth by 17.4%, MESIF$_{incl}$ by 15.5%, and MOESI by 13.1%. Despite not achieving the highest bandwidth reduction, MESIF$_{incl}$ benefits from the lowest interconnect overhead, generating only 2.0% more network traffic versus 5.1% for MESIF$_{excl}$. While MESIF$_{excl}$ maximizes effective cache capacity by pushing the forwarder into the private caches, this design adds two extra network hops to the critical path because shared read requests must be routed from the LLC to a peer cache. Conversely, MESIF$_{incl}$ keeps the forwarder directly in the LLC, avoiding these extra hops and the full memory fetches of MOESI to shorten the overall shared read latency. Radix (`rdx`) exemplifies this trend: MESIF$_{incl}$ achieves a 4.5% speedup versus 4.2% for MESIF$_{excl}$ and 3.1% for MOESI, driven by 5.2% lower miss latency than MESIF$_{excl}$ and 2.8% lower than MOESI.

In read-dominated benchmarks with large working sets, however, MESIF$_{excl}$ outperforms MESIF$_{incl}$ by preventing L1 lines from duplicating in the LLC, expanding effective cache capacity and reducing evictions. In linear regression (`lr`), this yields $1.9\times$ higher LLC load hit rates (13.0% vs. 6.8%), translating to a 4.3% speedup against 1.5% for MESIF$_{incl}$. In matrix multiply

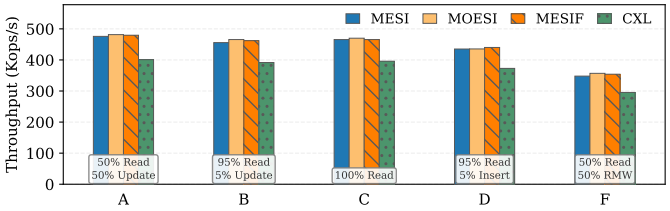


Fig. 10: Throughput (Kops/s) of four inter-node bridge protocol configurations in the multi-host CXL scenario across five YCSB workloads.

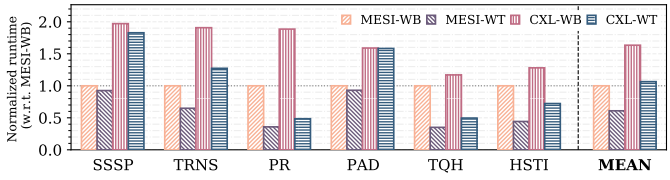


Fig. 11: Performance comparison between write-back and write-through policies for MESI and CXL off-core protocols in the hetero-chiplets scenario.

(mm), the effect is similarly pronounced: 1.3× higher LLC hit rates (19.4% vs. 14.8%) and a 7.7% vs. 5.1% speedup.

In write-sharing dominated benchmarks such as histogram (hist), MOESI outperforms MESIF$_{incl}$ (+1.2%) and MESIF$_{excl}$ (+2.7%). MESIF's F state requires a two-step process for dirty-line sharing: writeback to the directory followed by forwarding to the requestor. The writeback holds the directory in a transient state until it receives it, serializing subsequent requests to the same line. On the other hand, MOESI's O state eliminates this bottleneck via direct cache-to-cache transfers of dirty data, reducing directory stalls in hist by 48% and 35% against MESIF$_{excl}$ and MESIF$_{incl}$ respectively.

Insight. Larger cache capacities in modern systems make MESI protocols increasingly attractive, as the cache-space and traffic savings historically offered by O and F states become less critical. Our results, however, show that these states remain beneficial in NUMA topologies, where MESIF and MOESI outperform the MESI baseline, with MESIF's additional F state being most beneficial in this setting.

E. Use-case #2: CCP Effects in Distributed KVS

In the second use case, we evaluate the effect of varying bridge coherence protocols on latency and throughput under disaggregated memory conditions. For this, we use SYNAPSE to synthesize controllers for four off-core protocol variants (MESI, MOESI, MESIF, and CXL) in the multi-host CXL scenario (S3). All protocols use an inclusive policy and a fixed MESI on-core protocol.

Workloads. We build an in-memory distributed key-value store (KVS) using a concurrent hash-map with a dedicated allocator, where shared data objects reside on remote disaggregated memory while thread-private data remain local. We drive the KVS with YCSB [23] workloads to model realistic data-serving access patterns.

Results. Figure 10 compares the four protocols across the five YCSB workloads under 16 threads, measuring absolute throughput (Kops/s). Across all workloads, MOESI and MESIF closely track MESI, with average throughput gains of 1.4% (0.1%–2.6%) and 1.0% (0.0%–1.8%), respectively. In contrast, CXL shows a consistent throughput reduction, averaging 14.8% (14.1%–15.8%) relative to MESI.

Analysis. Figure 10 reveals a clear trend: CXL consistently underperforms all other protocol variants in throughput across all workloads. The reason is that YCSB workloads are characterized by read-dominated access patterns with periodic writes, producing a read-before-write profile in which cache lines accumulate sharers across clusters before an ownership upgrade is issued. This pattern maximizes cross-cluster invalidation traffic, where CXL's overhead is most pronounced. For write requests, dirty-owner invalidation requires 6 remote message delays versus 3 in the other variants. For read requests, CXL still blocks the directory awaiting memory writeback, whereas MOESI, and MESIF allow the prior owner/forwarder to supply data directly via Fwd_GetS. Both cases compound into a convoy effect on contended lines, causing CXL to incur 37% more cycles per LLC miss than MESI. Among the non-CXL variants, MOESI marginally outperforms MESI and MESIF as during dirty-owner invalidation, the O state supplies dirty data directly to the requestor on ownership downgrade, avoiding the writeback to memory that both MESI and MESIF must complete before releasing ownership.

Insight. CXL consistently underperforms the other protocols because of its less efficient invalidation mechanism. For a write request to a line shared across clusters, the directory first invalidates all sharers, then reads the line back from memory to supply it to the requester. Other protocols avoid this memory access by forwarding the line directly from the owner or forwarder.

F. Use-case #3: Impact of Caching Policies

In the final use case, we assess how caching policies affect performance under massively parallel workloads. Using SYNAPSE and the heterogeneous chiplet SoC scenario (S4), we compare four configurations combining MESI-RC on-core with either MESI or CXL off-core protocols, each with a write-back and write-through policy.

Workloads. We evaluate multi-PU workloads from the CHAI [32] and Pannotia [18] benchmark suites, which are widely used in previous works on multi-PU cache coherence protocols [4] [5] [73].

Results. Figure 11 reports execution time across all workloads normalized to MESI-WB. MESI-WT consistently outperforms MESI-WB, reducing execution time by 1.08–2.9× (average 1.9×). CXL-WT similarly outperforms CXL-WB by 1.08–3.8× (average 1.8×). Both CXL configurations underperform their MESI counterparts, with CXL-WB incurring up to 1.97× overhead relative to MESI-WB.

Analysis. Two patterns emerge from these results. First, the write-through policy outperforms write-back due to a different handling of non-temporal stores (writes annotated as having no reuse). Under WB, every store must first acquire exclusive ownership, even for streaming data with no reuse. Under WT, non-temporal stores bypass L1 and write directly to the LLC, serving as the coherence point in this design, without acquiring exclusive ownership. This design reduces both memory traffic and cache pollution. Memory

reads decrease by 1.5–14× across benchmarks. L1 evictions drop by up to 3× (PR), and are eliminated entirely in TQH, HSTI, and PAD, freeing cache capacity for data with actual reuse.

Second, both CXL configurations consistently underperform their MESI counterparts. This gap stems directly from the architectural difference between the two scenarios: the CXL interconnect introduces higher remote memory access latency, amplifying its cost regardless of the write policy.

Insight. For massively parallel workloads with streaming, no-reuse data, the caching policy is a more effective design lever than the off-core protocols: selecting write-through over write-back yields larger gains than switching the underlying coherence protocol (CXL write-through outperforms MESI write-back in PR, TQH, and HSTI benchmarks, despite the lower UPI interconnect latencies), making policy selection a first-order decision for accelerator-centric systems.

Takeaway. Together, the three use cases demonstrate that SYNAPSE enables practical design space exploration across diverse architectures, surfacing non-obvious trade-offs between protocol variants, inclusion policies, and interconnect characteristics—insights that can help hardware architects in making informed design decisions.

IX. Related Work

Cache coherent abstractions and DSLs. As detailed in Sec. II-D, SLICC and TileLink [76], the de facto DSLs for coherence modeling in gem5 and chip-scale SoC design respectively, both operate at a low abstraction level, requiring developers to manually define complete protocol state machines. While the ProtoGen [68] DSL raises the abstraction level by accepting atomic SSP, it remains limited to protocol semantics and lacks support for architectural configuration.

Orthogonally, Hemiola [19] and Dave et al. [24] introduce DSLs targeting hardware synthesis: Hemiola via a Coq-embedded [71] rule-based specification synthesized through Kami [20], Dave et al. via Bluespec for RTL generation from concurrent specifications. However, Hemiola targets specific architectures, while Bluespec requires specified protocols to already be concurrent, with no support for automatic concurrency generation.

Automated coherence protocol design. While we have already discussed ProtoGen [68], HieraGen [70], HeteroGen [69], and vCXLGen [52] at length in Sec. II-D, alternative approaches have explored program synthesis techniques for protocol construction with a focus only on verification. Examples include VerC3 [26] and TRANSIT [78], which treat protocol design as a program-synthesis problem from partial specifications. VerC3 mitigates state-space explosion through heuristic pruning, while TRANSIT relies on user-provided fixes to counterexamples. However, both approaches face scalability limitations when applied to complex, realistic protocol designs. Verifiability can also be achieved through manual design [49], [60], [82] which structure protocols hierarchically to facilitate formal verification, but without automating the design process itself.

In contrast, MapleBoard [80] and SYNTHIA [47] provide an end-to-end performance evaluation of synthesized protocols. However, they restrict the design space to predictable, snooping bus-based coherence protocols for real-time multi-processor systems-on-chip (MPSoCs), which limits exploration of directory-based protocols commonly used in modern large-scale systems.

Heterogeneous cache coherence. Combining heterogeneous coherence protocols across CPUs, GPUs, and accelerators requires bridging fundamentally different consistency semantics, a challenge addressed by diverse industry standards [12], [21], [22], [28] through configurable coherence mechanisms. Prior work has tackled protocol bridging manually [6], [49], [67], each tailored to specific protocol combinations or interface assumptions. However, these approaches assume a static architecture and lack generality: merged directory solutions [6] prevent dynamic host attachment, while custom interface approaches [67] remain confined to SWMR semantics.

X. Conclusion

This work presents SYNAPSE, a generic and systematic methodology and compilation framework for the automated synthesis of cache coherence controllers grounded in a key observation: any controller FSM is fully determined by three orthogonal constituents: protocol specification, system topology, and caching policies. Decoupling these constituents enables SYNAPSE to synthesize correct, architecture-aware controllers for arbitrarily complex systems via a declarative DSL, a synthesis pipeline, and a gem5 back end, with correctness validated through Murφ model checking and memory consistency litmus tests. Our evaluation across four representative modern architectures confirms that synthesized controllers match hand-tuned gem5 implementations within 1.0% on average, while three use cases demonstrate SYNAPSE's practicality in exposing non-obvious protocol and policy trade-offs across NUMA, multi-host CXL, and heterogeneous chiplet scenarios.

Acknowledgment

We thank the anonymous reviewers and members of the Systems Research Group at the TU Munich for their feedback. This work was supported in part by the DFG Priority Program "Disruptive Memory Technologies" (SPP 2377), an ERC Starting Grant (ID: 101077577), the Chips Joint Undertaking (JU), European Union(EU) HORIZON-JU-IA (No. 101140087), the Intel Trustworthy Data Center of the Future (TDCoF), Google Research Grants.

Artifact Appendix

A. Abstract

This artifact provides the synthesized cache-coherence controllers of Synapse, the two gem5-based environments that run them, and the scripts to reproduce our evaluation (Figures 8–11 and the ARM litmus tests). We include detailed instructions for using the models and for building the simulators, benchmarks, and their dependencies from scratch.

Because the evaluation relies on two distinct simulators, the artifact is split into two self-contained parts. Scenarios S1–S3 (Figures 8–10 and the litmus tests) target gem5 v23, whereas scenario S4 (Figure 11, heterogeneous accelerator chiplets) targets gem5 v20. Each part ships with a `Dockerfile` that provisions the corresponding environment.

Building all workloads and all 24 model variants (14 x86 and 10 ARM) takes roughly 4–6 hours on a 32-core server. To avoid

this, we also provide two prebuilt Docker images, one per environment, with every binary already compiled and ready to run.

B. Artifact check-list (meta-information)

- **Compilation:** GCC/G++ 11.4.0, SCons 4.0+, Python 3.10+, gem5 v23.1.0.0 dependencies (S1–S3); gem5 v20.1.0.0 dependencies (S4), and a RISC-V cross toolchain, provided in the S4 Docker image (S4).
- **Data set:** PARSEC 3.0, SPLASH-4, Phoenix-2.0, YCSB-C, Rodinia and Chai. Litmus tests: *2_2W, MP, R, S, SB* and their fenced (`dmb.sy`, `dmb.sys`) variants, generated with Herd7 [3].
- **Run-time environment:** Ubuntu 22.04 LTS (native or Docker container).
- **Hardware:** Any general-purpose x86_64 CPU.
- **Metrics:** Execution time (Figures 8, 9, 11), throughput (Figure 10), and litmus-test outcomes.
- **Experiments:** Use the provided per-figure scripts to evaluate the synthesized controllers against the handwritten baselines and across the coherence-protocol combinations of each scenario.
- **How much disk space is required? (approx):** ∼40 GiB. Prebuilt images: ∼7 GiB (S1–S3) and ∼3 GiB (S4); one gem5 build directory: ∼5 GiB; benchmarks (incl. PARSEC inputs): ∼4 GiB; experiment outputs: ∼15 GiB.
- **How much time is needed to prepare workflow? (approx):** ∼4–6 hours to compile all 24 gem5 variants; ∼30 minutes to compile the workloads. None if the prebuilt images are used.
- **How much time is needed to complete experiments? (approx):** ∼1 day for the full suite (Figures 8, 9, 10, 11, and the litmus tests) on a 32-core server; a functional check takes under a minute.
- **Publicly available:** Yes.
- **Code licenses:** MIT.
- **Archived (DOI):** https://doi.org/10.5281/zenodo.21539116

C. Description

1) How to access: The source code is publicly available on GitHub (https://github.com/TUM-DSE/Synapse) and archived on Zenodo (https://doi.org/10.5281/zenodo.21539116).

2) Hardware dependencies: The artifact can be evaluated on any general-purpose x86_64 CPU with at least 40 GiB of free disk space to build and run both gem5 environments. Each gem5 simulation uses about 12 GiB of RAM (median 9.6 GiB over the runs behind the paper's figures; worst case ∼53 GiB); the run scripts cap concurrency according to the available memory.

We recommend a server with at least 32 cores to speed up compilation and to parallelize the simulations.

3) Software dependencies: We recommend running all compilation and experiments inside the provided Docker containers, in which case the only software dependency is a working Docker environment. Alternatively, the S1–S3 part can be evaluated on a native Ubuntu 22.04 installation (native, container, or VM); refer to the artifact `README` for the native setup instructions. The S4 part requires a RISC-V cross toolchain and is only supported through its Docker image.

4) Data sets: The artifact includes the source code and instructions to build all workloads from source. We also provide the compiled binaries as two prebuilt Docker images on Docker Hub (https://hub.docker.com/r/carpeniem/synapse-artifact-s1_2_3 and https://hub.docker.com/r/carpeniem/synapse-artifact-s4). To build from source, use the `Build` scripts as noted in the `README`.

D. Installation

The artifact repository contains all necessary components, including the two gem5 simulators, the synthesized coherence controllers, the benchmark suites, and the experiment scripts. For each component we provide detailed build instructions in the corresponding `README` section.

To build gem5 with all protocol variants for both the x86 (performance figures) and ARM (litmus tests) architectures, follow the `Build gem5` section of the `README`. To build the accelerator-chiplet environment for Figure 11, follow the Figure 11 instructions.

To build the benchmarks, follow the `Build Benchmarks` section of the `README`.

Each figure is produced by a single command that generates its experiment configuration, runs the simulations, extracts the statistics, and plots the result; detailed instructions are in the `Run Experiments` section.

E. Experiment workflow

A quick functional check (`just functional-test`, under a minute) runs the synthesized controllers on a deterministic workload and compares the result against native execution, exercising the whole toolchain before any full experiment.

The four figures are then reproduced independently with `just fig8`, `fig9`, `fig10`, `fig11`, and the correctness evaluation with `just litmus`.

F. Evaluation and expected results

Once the experiments complete, each run script writes the corresponding figure into the `data` directory of its experiment tree.

The figures should match those in the paper: the synthesized controllers track the handwritten gem5 baselines within about 1% (Figure 8), and the protocol and policy studies of Figures 9, 10, and 11 reproduce the reported trends. The litmus-test report confirms that every forbidden outcome is correctly prevented by the synthesized controllers.

G. Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

References

[1] "The princeton application repository for sharedmemory computers (parsec)." [Online]. Available: http://parsec.cs.princeton.edu/.

[2] I. Advanced Micro Devices, Jan. 2011. [Online]. Available: https://ir.amd.com/news-events/press-releases/detail/168/amd-fusion-apu-era-begins

[3] J. Alglave and L. Maranget, "herd7 consistency model simulator," http://diy.inria.fr/www/, 2022.

[4] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy release consistency for gpus," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[5] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A flexible interface for efficient heterogeneous coherence," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, M. Annavaram, T. M. Pinkston, and B. Falsafi, Eds. IEEE Computer Society, 2018, pp. 261–274. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00031

[6] ——, "Spandex: a flexible interface for efficient heterogeneous coherence," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 261–274. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00031

[7] AMD. [Online]. Available: https://www.amd.com/en/newsroom/press-releases/2025-5-20-amd-introduces-new-radeon-graphics-cards-and-ryzen.html

[8] AMD, "Amd64 architecture programmer's manual volume 2: System programming," 2012. [Online]. Available: https://web.archive.org/web/20170619232736/http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf#page=217

[9] ARM. [Online]. Available: https://developer.arm.com/documentation/den0013/0400/big-LITTLE

[10] ——, 2015. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/joined-up-thinking-arm-s-interconnect-portfolio

[11] ARM, "Mesi and moesi protocols," 2024. [Online]. Available: https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Cache-coherency/MESI-and-MOESI-protocols

[12] ARM Limited, "AMBA 5 CHI architecture specification," ARM Limited, Technical Specification ARM IHI 0050E.a, 2020, version E.a. [Online]. Available: https://developer.arm.com/documentation/ihi0050/latest/

[13] L. Backes and D. A. Jiménez, "The impact of cache inclusion policies on cache management techniques," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 428–438. [Online]. Available: https://doi.org/10.1145/3357526.3357547

[14] M. Bailleu, D. Stavrakakis, R. Rocha, S. Chakraborty, D. Garg, and P. Bhatotia, "Toast: A heterogeneous memory management system," in *2024 33rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024, pp. 53–65.

[15] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[16] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 01 2008, pp. 43–57.

[17] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó, and M. Casas, "Reducing cache coherence traffic with a numa-aware runtime approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1174–1187, 2018.

[18] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," 09 2013, pp. 185–195.

[19] J. Choi, A. Chlipala, and Arvind, "Hemiola: A dsl and verification tools to guide design and proof of hierarchical cache-coherence protocols," in *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 317–339. [Online]. Available: https://doi.org/10.1007/978-3-031-13188-2_16

[20] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3110268

[21] C. Consortium, "Ccix standard." [Online]. Available: https://www.ccixconsortium.com/

[22] C. E. L. Consortium, "Compute express link (cxl) specification 3.2," Compute Express Link Consortium, Tech. Rep., 2024. [Online]. Available: https://computeexpresslink.org/cxl-specification/

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[24] N. Dave, M. Ng, and Arvind, "Automatic synthesis of cache-coherence protocol processors using bluespec," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, 2005, pp. 25–34.

[25] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 33–48. [Online]. Available: https://doi.org/10.1145/2517349.2522714

[26] M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan, "Verc3: A library for explicit state synthesis of concurrent systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1381–1386.

[27] P. Esmaili-Dokht, F. Sgherzi, V. S. Girelli, I. Boixaderas, M. Carmin, A. Monemi, A. Armejach, E. Mercadal, G. Llort, P. Radojković, M. Moreto, J. Giménez, X. Martorell, E. Ayguadé, J. Labarta, E. Confalonieri, R. Dubey, and J. Adlard, "A mess of memory system benchmarking, simulation and application profiling," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 136–152.

[28] H. Foundation, "Heterogeneous system architecture (hsa) foundation," 2024. [Online]. Available: https://hsafoundation.com/

[29] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern numa systems," *Commun. ACM*, vol. 58, no. 12, p. 59–66, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2814328

[30] A. Goens, S. Chakraborty, S. Sarkar, S. Agarwal, N. Oswald, and V. Nagarajan, "Compound memory models," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591267

[31] 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

[32] J. Gómez-Luna, I. E. Hajj, L.-W. Chang, V. Garcia-Flores, S. G. D. Gonzalo, T. B. Jablin, A. J. Peña, and W. mei W. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 43–54, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:8297740

[33] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov, "Bandwidth-aware page placement in numa," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 546–556.

[34] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 413–422.

[35] B. A. Hechtman and D. J. Sorin, "Evaluating cache coherent shared virtual memory for heterogeneous multicore chips," 2013. [Online]. Available: https://arxiv.org/abs/1310.7792

[36] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[37] M. Hill and V. Janapa Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 317–330.

[38] M. Hille, N. Asmussen, H. Härtig, and P. Bhatotia, "A heterogeneous microkernel os for rack-scale systems," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 50–58. [Online]. Available: https://doi.org/10.1145/3409963.3410487

[39] C.-C. Huang, R. Kumar, M. Elver, B. Grot, and V. Nagarajan, "C3d: Mitigating the numa bottleneck via coherent dram caches," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[40] E. humanum est, "Hieragen/murphi · errare-humanum-est/hieragen, https://github.com/errare-humanum-est/hieragen/tree/master/murphi," 2025. [Online]. Available: https://github.com/Errare-humanum-est/HieraGen/tree/master/Murphi

[41] P. Hübner, A. Hu, I. Peng, and S. Markidis, "Apple vs. oranges: Evaluating the apple silicon m-series socs for hpc performance and efficiency," in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025, p. 45–54. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11106171/references

[42] Intel. [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html

[43] Intel. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-2/intel-xe-gpu-architecture.html

[44] Intel, "Intel® 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1," Intel, Tech. Rep., 2024.

[45] Intel, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[46] D. P. Kaur and V. Sulochana, "Design and implementation of cache coherence protocol for high-speed multiprocessor system," in *2018 2nd IEEE International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES)*, 2018, pp. 1097–1102.

[47] A. M. Kaushik and H. Patel, "Automated synthesis of predictable and high-performance cache coherence protocols," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 816–821.

[48] E. F. Kfoury, S. Choueiri, A. Mazloum, A. AlSabeh, J. Gomez, and J. Crichigno, "A comprehensive survey on smartnics: Architectures, development models, applications, and research directions," *IEEE Access*, vol. 12, p. 107297–107336, 2024.

[49] E. Ladan-Mozes and C. E. Leiserson, "A consistency architecture for hierarchical shared caches," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 11–22. [Online]. Available: https://doi.org/10.1145/1378533.1378536

[50] C. Lameter, "Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors." *Queue*, vol. 11, no. 7, pp. 40–51, 2013.

[51] J. H. Lau, "Recent advances and trends in chiplet design and heterogeneous integration packaging," *Journal of Electronic Packaging*, vol. 146, no. 010801, 2023. [Online]. Available: https://doi.org/10.1115/1.4062529

[52] A. Lefort, J. Pritzi, N. Carpentieri, D. Schall, S. Dittrich, S. Chakraborty, N. Oswald, and P. Bhatotia, "vcxlgen: Automated synthesis and verification of cxl bridges for heterogeneous architectures," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 2177–2196. [Online]. Available: https://doi.org/10.1145/3779212.3790245

[53] A. Lefort, D. Schall, N. Carpentieri, J. Pritzi, S. Chakraborty, N. Oswald, and P. Bhatotia, "C^3: CXL Coherence Controllers for Heterogeneous Architectures," in *Proceedings of the 32nd IEEE International Symposium on High-Performance Computer Architecture (HPCA-32)*, 2026.

[54] T. Li, J. Hou, J. Yan, R. Liu, H. Yang, and Z. Sun, "Chiplet heterogeneous integration technology—status and challenges," *Electronics*, vol. 9, no. 4, p. 670, Apr. 2020.

[55] X. Lian, X. Ning, M. Xie, and F. Yu, "Cache coherence protocols in shared-memory multiprocessors," in *Proceedings of the 2015 International Conference on Computational Science and Engineering*. Atlantis Press, 2015/07, pp. 286–289. [Online]. Available: https://doi.org/10.2991/iccse-15.2015.52

[56] J. Lowe-Power, "Introduction to ruby," 2025. [Online]. Available: https://www.gem5.org/documentation/learning_gem5/part3/MSIintro/

[57] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrill'on, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, 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, T. M"uck, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. 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, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and 'Eder F. Zulian, "The gem5 simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[58] A. Ltd., "AMBA® 5 CHI Architecture Specification," Arm Ltd., Cambridge, United Kingdom, Specification IHI 0050 G, Mar. 2024, non-confidential public release — Ninth edition. [Online]. Available: https://developer.arm.com/documentation/ihi0050/latest

[59] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[60] O. Matthews and D. J. Sorin, "Architecting hierarchical coherence protocols for push-button parametric verification," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 477–489.

[61] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 261–270.

[62] D. Molka, R. Schöne, D. Hackenberg, and M. S. Müller, "Memory performance and spec openmp scalability on quad-socket x86 64 systems," in *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 170–181.

[63] V. Nagarajan, D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Feb. 2020.

[64] NVIDIA. [Online]. Available: https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/

[65] NVIDIA, "Dpu." [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[66] NVIDIA, "Nvlink." [Online]. Available: https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf

[67] L. E. Olson, M. D. Hill, and D. A. Wood, "Crossing guard: Mediating host-accelerator coherence interactions," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 163–176. [Online]. Available: https://doi.org/10.1145/3037697.3037715

[68] N. Oswald, V. Nagarajan, and D. J. Sorin, "Protogen: Automatically generating directory cache coherence protocols from atomic specifications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 247–260.

[69] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr, "Heterogen: Automatic synthesis of heterogeneous cache coherence protocols," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 756–771.

[70] N. Oswald, V. Nagarajan, and Sorin, Daniel J., "Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 888–899.

[71] C. Paulin-Mohring, "Introduction to the coq proof-assistant for practical software verification," in *Workshop on Learning from Authoritative Security Experiment Results*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:17057477

[72] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," USA, p. 13–24, 2007. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346181

[73] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems," 02 2020, pp. 582–595.

[74] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 241–252. [Online]. Available: https://doi.org/10.1145/2370816.2370853

[75] S. Shukla and M. Chaudhuri, "Pool directory: Efficient coherence tracking with dynamic directory allocation in many-core systems," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015, pp. 557–564.

[76] SiFive, Inc., "Sifive tilelink specification," 2018. [Online]. Available: https://sifive.cdn.prismic.io/sifive%2F57f93ecf-2c42-46f7-9818-bcdd7d39400a_tilelink-spec-1.7.1.pdf

[77] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 578–590.

[78] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: specifying protocols with concolic snippets," *SIGPLAN Not.*, vol. 48, no. 6, p. 287–296, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462174

[79] Y. Wang, L. Wu, W. Hong, Y. Ou, Z. Wang, S. Gao, J. Zhang, S. Ma, D. Dong, X. Qi, M. Lai, and N. Xiao, "Cxl-dmsim: A full-system cxl disaggregated memory simulator with comprehensive silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1–1, 2025. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2025.3607145

[80] Z. Wu, A. M. Kaushik, P. Tegegn, and H. Patel, "A hardware platform for exploring predictable cache coherence protocols for real-time multicores," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 92–104.

[81] Y. Yu, N. Oswald, and A. Khandelwal, "Cord: Low-latency, bandwidth-efficient and scalable release consistency via directory ordering," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1311–1326. [Online]. Available: https://doi.org/10.1145/3695053.3731074

[82] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 471–482.