A. M. Kaushik and H. Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols," in proceedings of IEEE Design Automation and Test in Europe (DATE), Mar. 2021, pp. 1–6. doi: 10.23919/DATE51398.2021.9474238.

Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols

Anirudh Mohan Kaushik and Hiren Patel
Electrical and Computer Engineering Department
University of Waterloo, Waterloo, Canada
{anirudh.m.kaushik, hiren.patel}@uwaterloo.ca

***Abstract*—We present SYNTHIA, an open and automated tool for synthesizing predictable and high-performance snooping bus-based cache coherence protocols for multi-core processors in multi-processor system-on-chips (MPSoCs) deployed in real-time systems. SYNTHIA automates the complex analysis associated with designing predictable and high-performance cache coherence protocols, and constructs new states (transient states) and corresponding transitions that achieve predictability and performance. We use SYNTHIA to construct complete protocol implementations from simple specifications of common protocols (MSI, MESI, and MOESI protocols). We validated the correctness, predictability, and performance guarantees of the generated protocol implementations from SYNTHIA using manually implemented versions, and a micro-architectural simulator.**

I. INTRODUCTION

Hardware cache coherence protocols for multi-core processors in MPSoCs for real-time and safety-critical systems have recently become an attractive solution for predictably managing shared data communication between the multiple cores [1]–[3]. Recent efforts showed that it was possible for predictable cache coherence to offer up to 4× average-case performance improvement over traditional alternatives used in multi-core platforms [1], [2] while also providing worst-case latency bounds essential for schedulability in real-time systems.

A hardware cache coherence protocol has a set of rules that ensures memory operations from cores operate on up-to-date versions of the requested data. The coherence protocol is a state machine with *coherence states,* and *transitions* between coherence states. Designing cache coherence protocols that deliver high-performance and that are correct is known to be challenging [4]. This is because the design process requires manually analyzing all possible interleavings of memory operations from different cores to the same shared data, and then constructing protocols that allow for these interleavings with little to no stalling of the memory operations. Designing one that also guarantees worst-case latency bounds (often called predictability) further exacerbates the challenge. This is because ensuring predictability while considering the many scenarios of interleaving memory operations across different cores requires intricate analyses of the hardware architecture and the protocol [1]. Missing one scenario can compromise predictability or limit the achievable performance.

The increase in complexity in designing predictable and high-performance cache coherence protocols comes in the form of additional states and transitions to the protocol [1], [5]. For example, the Modified-Shared-Invalid (MSI) protocol with no additional support for predictability or high-performance has 3 states and 12 transitions. A predictable and high-performance variant of the same protocol, however, has 15 states and 58 transitions [1]; a 5× increase in protocol size (number of states and transitions). A protocol designer is more prone to miss some states and transitions due to this dramatic increase in protocol complexity to achieve predictability and high-performance, which in turn compromises on correctness.

To improve productivity and simplify the construction of *correct, predictable, and high-performance* cache coherence protocols, we propose SYNTHIA, a tool that automates the coherence protocol construction. SYNTHIA takes as input a simple specification of a protocol that is devoid of states and transitions to achieve predictability or high-performance. This allows a protocol designer to focus on how a memory operation proceeds correctly *without* worrying about interleaving memory operations on the same data and carrying out the memory operation in a predictable manner. SYNTHIA *refines* this simple input specification and produces a predictable protocol implementation that achieves predictability and high-performance.

Our novel contributions in this work are listed next.

- We present an approach to automatically construct predictable and high-performance snooping bus-based cache coherence protocols.
- We implement our approach in a tool called SYNTHIA. SYNTHIA carefully analyzes scenarios that require access to the shared bus including those that allow simultaneous interleaving memory operations on the same data. This analysis results in the construction of new states and transitions that achieve predictability and high-performance.
- We evaluate SYNTHIA by generating predictable and high-performance protocol implementations for several common protocols [5]. On average, the complexity of the generated protocols have an increase of 4.9× the number of states and transitions. We thoroughly validate their correctness and ensure they are efficient. SYNTHIA is available at https://github.com/caesr-uwaterloo/Synthia.

II. BACKGROUND AND RELATED WORKS

A. Hardware cache coherence

Coherence states. Coherence states encode information about access permissions (read/write) on the cache line, and the state of the cache line data contents. There are two types of coherence states: (1) *stable* states (s-states) and (2) *transient*

A. M. Kaushik and H. Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols," in proceedings of IEEE Design Automation and Test in Europe (DATE), Mar. 2021, pp. 1–6. doi: 10.23919/DATE51398.2021.9474238.

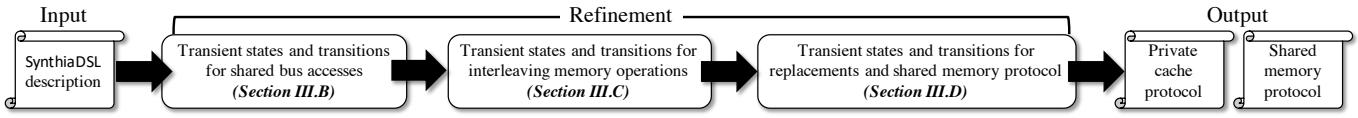

Fig. 1: High level overview of SYNTHIA.

states (t-states) [5]. Memory operations on a cache line begin and end on s-states, and a cache line goes through different t-states at various stages of a memory operation. Examples of a core's memory operations on a cache line include read/write memory requests and data responses or coherence message communication due to own requests or in response to other core's memory requests. There are three types of t-states, and each t-state is suffixed with the following: (1) **_AD**, (2) **_D**, and (3) **_A**. **_AD** t-states denote that a core has issued a memory request to a cache line, and is waiting for both the request to be *ordered* on the snooping bus and the requested cache line data contents. **_D** t-states denote that a core has observed its memory request on the snooping bus, and is waiting for the cache line data contents. **_A** t-states denote that a core has the cache line data, and is waiting for its memory operation to be ordered on the snooping bus.

Transitions. A cache line in a core's private cache *transitions* from one state (s-state or t-state) to another state based on the core's memory operations on the cache line or on observing memory operations from other cores to the same cache line. When a transition is taken, the cache controller executes actions to ensure data correctness. For example, exchanging coherence messages between cores and the shared memory [5].

Snooping bus-based protocols. In snooping bus-based protocols, cores observe memory operations from other cores by snooping a shared bus. Snooping bus-based protocols are typically deployed in multi-core platforms with fewer cores (less than 16 cores), and are representative of those used in real-time domains [5], [6]. Data communication using a snooping bus-based cache coherence protocol begins when a core generates a memory request to a cache line. If the data for the requested cache line is available in the core's private cache, the request is a cache hit. Otherwise, the core's cache controller *issues* a coherence message based on the type of memory request (read or write). The cache controller then *broadcasts* the coherence message on the snooping bus. The coherence message is *ordered* on the bus when all cores and the shared memory observe the broadcasted coherence message.

B. Predictable hardware cache coherence

Predictable hardware cache coherence protocols ensure that there is a worst-case latency bound on memory accesses across all cores [1]–[3]. These protocols are deployed on a multi-core model that uses a shared snooping bus to communicate coherence messages between cores and the shared memory, and a shared data bus between cores and the shared memory. The shared snooping bus is a non-atomic split transaction bus [5]. The shared snooping bus and data bus deploy a *predictable arbitration policy* to predictably manage simultaneous accesses from cores. Examples of predictable arbitration policies include time division multiplexing (TDM) and round-robin (RR). These predictable arbitration policies divide access time to the shared bus into fixed time slots, and allocates these time slots to cores. A core is granted *exclusive* access to the bus at the start of its allocated slot. A core can only access the bus in its allocated slot; a pending bus access from a core that arrives immediately after the start of its allocated slot must wait for the start of its next allocated slot [1]. The memory hierarchy of the multi-core consists of one level of split private data and instruction write-back caches, and a shared last level cache memory. The private caches store a subset of data present in the shared memory. A core can communicate data in its private cache with other cores through point-to-point interconnects.

Prior works on designing predictable cache coherence protocols such as [1], [2] modified existing conventional cache coherence protocols to satisfy predictability. These works first exhaustively analyzed different scenarios that can result in unpredictable scenarios. New t-states and transitions were constructed to address these unpredictable scenarios while maintaining data correctness and most of the performance benefits in the conventional protocols. Depending on the conventional protocol complexity, the analysis and the number of t-states and transitions to be constructed for predictability can be high making it an error prone process. SYNTHIA relieves this complexity burden by *automating* the analysis.

C. Related works

Oswald et al. [4] recently presented ProtoGen, an automated tool that constructed high-performance directory-based cache coherence protocols. While SYNTHIA takes inspiration from ProtoGen, it differs from it in two ways. First, SYNTHIA generates snooping bus-based coherence protocols, which have different designs and construction mechanisms compared to directory based protocols [5]. This is because of differences in coherence message communication (broadcast vs unicast) and ordering mechanisms (bus vs directory) [5]. Second, SYNTHIA constructs *predictable high-performance* coherence protocols whereas ProtoGen constructed coherence protocols that only optimize performance. As a result, protocols generated with SYNTHIA can be used in real-time multi-cores.

Alternate protocol synthesis tools such as Transit [7] and VerC3 [8] relied on program synthesis that use a combination of designer provided guidance and model checking to complete partial descriptions of an input protocol specification. A key feature of these tools was frequent designer intervention to add information to the input specification for correct protocol construction [4]. On the other hand, SYNTHIA only requires a designer to provide a simple input specification, and generates the corresponding correct, predictable, and high-performance protocol implementation without further designer intervention.

III. SYNTHIA IMPLEMENTATION

Figure 1 presents an overview of SYNTHIA. SYNTHIA takes as input a protocol specification written in SYNTHIADSL (Sec-

A. M. Kaushik and H. Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols," in proceedings of IEEE Design Automation and Test in Europe (DATE), Mar. 2021, pp. 1–6. doi: 10.23919/DATE51398.2021.9474238.

tion III-A). The specification consists of s-states and transitions between s-states at the private cache level. Note that SYNTHIA assumes the input specification is correct, and does not perform any verification for correctness on the input. The input is refined by creating new t-states and corresponding transitions, and results in a predictable and high-performance protocol implementation. This refinement identifies two main scenarios to construct t-states: (1) transitions that must wait for some communication on the shared bus such as broadcast coherence messages or data communication with shared memory (Section III-B), and (2) transitions that change due to interleaving memory operations on the same cache line (Section III-C). In these sections, we focus more on the construction of t-states as there are several subtleties to consider. We explain the construction of transitions due to t-states using examples.

A. Protocol specification in SYNTHIADSL

The input information about s-states and transitions is defined in a domain specific language, SYNTHIADSL. There are two components in the input: (1) coherence state encoding of s-states and (2) transitions between s-states. Figure 2 shows the MSI protocol specification in SYNTHIADSL.

Coherence state encoding. Each s-state of a cache line specified in the input is a 3-tuple of the form (ap, ds, da) where ap is the access permission, ds is the data state of the cache line, and da is the data authority of the cache line. The access permission conveys the type of memory operation (read/write) permitted on the cache line by a core. A core that does not have a cache line in its private cache has $invalid$ access permissions. The data state of a cache line conveys whether a core has modified the data contents of the cache line. A $dirty$ data state means that a core may have modified the data contents, and $clean$ data state means that the core has not modified the data contents. The data authority of a cache line conveys whether a core can communicate the cache line data contents to another core that requests for the same cache line via the point-to-point data interconnects. An $active$ data authority means that a core can send the cache line data contents in its private cache to the requesting core and $passive$ data authority means the core does not respond with data to another core's request. Lines 1-3 show the coherence state encoding for the `M`, `S`, and `I` states.

A key benefit of this state encoding is that protocols with states different from those found in the common MSI, MESI, and MOESI protocols can also be modeled in SYNTHIADSL. For example, there is no state with encoding $(read, active, clean)$ in the common protocols. A core that has a cache line in such a state can respond with unmodified data to other cores' memory requests to the same cache line. Hence, SYNTHIA can construct protocols from different protocol specifications including the common ones.

Transitions. $(\texttt{src}, \texttt{ev}) \rightarrow \texttt{dst}$ is a transition where `src` is the source s-state, `dst` is the destination s-state, and ev is the event that triggers the transition. OwnRead and OwnWrite events denote a core's own read and write requests *issued* to a cache line, and OtherRead and OtherWrite events denote other cores' read and write requests to a cache line *ordered* on the bus. Replacement denotes a cache line replacement.

```
1  M : (write, dirty, active)
2  S : (read, clean, passive)
3  I : (invalid, clean, passive)
4  (I, OwnRead)      -> S
5  (I, OtherRead)    -> I
6  (I, OwnWrite)     -> M
7  (I, OtherWrite)   -> I
8  (S, Replacement)  -> I
9  (S, OwnRead)      -> S
10 (S, OtherRead)    -> S
11 (S, OwnWrite)     -> M
12 (S, OtherWrite)   -> I
13 (M, OwnRead)      -> M
14 (M, OtherRead)    -> S
15 (M, OwnWrite)     -> M
16 (M, OtherWrite)   -> I
17 (M, Replacement)  -> I
```

Fig. 2: MSI protocol specification in SYNTHIADSL.

Lines 4-17 define the state transitions in the MSI protocol. For example, consider $(\texttt{I}, \text{OwnRead}) \rightarrow \texttt{S}$. This means that a core performs a read operation on a cache line that it does not have in its private cache (`I`). On receiving the requested cache line data, the core transitions the cache line to `S` state. We use OwnWR (OtherWR) to denote a transition triggered on either OwnRead or OwnWrite (OtherRead or OtherWrite).

Note that the input in SYNTHIADSL specification does not require transitions that are triggered when an own memory operation is *ordered* on the bus, and when the core *receives* the requested data. Furthermore, there are no *actions* that a core's cache controller should execute as a consequence of the transition. Examples of actions include sending data to another core (SD), and write-back data to shared memory (WD). During protocol construction, SYNTHIA automatically adds such transitions triggered when the memory operation is ordered (Ordered) and on receiving the requested data (RD), and the appropriate actions based on the data state and data authority of the states involved in the transition.

B. t-states for communication on the shared bus

Key idea. Recall that the shared bus for predictable cache coherence protocols uses a predictable arbitration policy that allocates each core a fixed time slot to exclusively access the bus [1]. This means that a core must wait for its allocated time slot to communicate on the shared bus. These protocols use t-states to denote that a core has pending shared bus communication and is waiting for its allocated time slot [1]. Hence, SYNTHIA analyzes each transition in the input specification, and identifies whether a transition must communicate coherence messages or data or both on the shared bus.

Mechanism. Algorithm 1 shows our implementation that discovers when t-states need to be added. The input to Algorithm 1 is a transition t. The procedures SOURCE and DESTINATION return the source and destination s-states of t. This algorithm exploits two key insights. First, for transitions triggered on own memory operations (line 3), t-states are required only when (a) `src` has $invalid$ access permissions and $\texttt{src} \neq \texttt{dst}$ (lines 6-7) or (b) `src` has $clean$ data state and the operation is OwnWrite (lines 8-9). Second, for transitions triggered on other memory operations, t-states are required depending on the overall state of the cache line before and after the memory operation *across all cores* (lines 10-16). Using Figure 3 as an illustrative example, we explain this implementation.

Consider insight (1). If `src` has $invalid$ access permissions (ISINVALIDAP returns true), then `src` does not have the cache line data contents to complete the own memory operation (lines

A. M. Kaushik and H. Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols," in proceedings of IEEE Design Automation and Test in Europe (DATE), Mar. 2021, pp. 1–6. doi: 10.23919/DATE51398.2021.9474238.

Algorithm 1 t-states for shared bus communication

```
1:  procedure IsTSNeededBusComm(t)
2:      src = t.source(), dst = t.destination(), ev = t.event()
3:      if IsOwn(ev) then
4:          if src == dst then
5:              return false
6:          else if IsInvalidAP(src) then
7:              return true
8:          else if IsCleanDS(src) ∧ ev == OwnWrite then
9:              return true
10:     else if IsDirtyDS(src) ∨ IsActiveDA(src) then
11:         tList = ownTransitions(ev)
12:         for all ot ∈ tList do
13:             if IsCumulativeChange(ot, t) ≠ 0 then
14:                 return true
15:     return false
```

6-7). Hence, such transitions require t-states that wait for both the broadcast of coherence message regarding the memory operation to be ordered on the bus and the requested data contents. In Figure 3, (I, OwnRead) → S has t-states IS_AD and IS_D where IS_AD waits for the coherence message broadcast to be ordered and IS_D waits for the requested data.

The single-writer-multiple-reader (SWMR) is a key invariant that coherence protocols must satisfy for data correctness [5]. This invariant mandates that at any instance of time either multiple cores have read-only copies of the same cache line (*clean* data state) in their private caches or only one core has a write-only copy of the cache line (*dirty* data state) in its private cache. Hence, for (src, OwnWrite) → dst where src has *clean* data state (lines 8-9), the core must at least broadcast a coherence message so that other cores can invalidate their cache line copies to satisfy the SWMR invariant. As a result, at least one t-state is required that waits for the coherence message broadcast to be ordered on the shared bus. In Figure 3, SM_A is a t-state that waits for the broadcast of OwnWrite.

Consider insight (2). Unlike the previous case, determining whether (src, Other) → dst requires t-states by solely looking at the properties of src and dst can introduce *unnecessary* t-states. Unnecessary t-states introduces unnecessary bus communication, which in turn causes unnecessary delays to the memory operation. As an example, consider the transitions (M, OtherRead) → S and (M, OtherWrite) → I. Although both I and S have same data authority and data state, (M, OtherWrite) → I does not require t-states whereas (M, OtherRead) → S requires at least one t-state. This is because (M, OtherRead) → S performs a write-back of the updated data contents, which must wait for the allocated time slot to communicate data to the shared bus. Hence, at least one t-state is required to indicate the pending write-back operation.

We find that taking into account the *cumulative* coherence states of a cache line across *all* cores can identify whether (src, Other) → dst must access the shared bus, and hence, requires t-states. For example, consider a two-core system c_0 and c_1 where c_0 has cache line X in M state and c_1 does not have X (I state). Consider that c_1 issues an OwnWrite. c_0 moves to I and c_1 moves to M after c_1's OwnWrite based on the transitions described in Figure 2. Notice that only one core has X in M state before and after c_1's memory operation. Hence, the *cumulative* data state and data authority of X across all cores

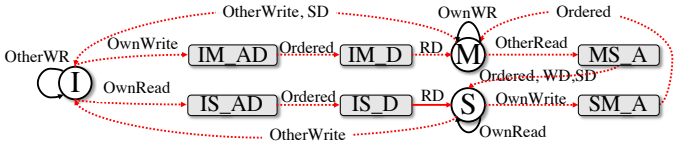

Fig. 3: MSI protocol refinement for communication on the shared bus.

remains the *same* before and after c_1's memory operation. As a result, there is no need for c_0 to communicate the updated data contents of X to the shared memory, and inform the shared memory about the change in its data authority of X. Alternatively, consider that c_1 issues an OwnRead. c_0 and c_1 transition to S after c_1's OwnRead. In this scenario, the cumulative data state and data authority of X across all cores *changes* after c_1's OwnRead. Before the memory operation, c_0 has X with *dirty* data state and *active* data authority, and after the memory operation, c_0 and c_1 have X with *clean* data state and *passive* data authority. In this case, c_0 must communicate the change in data authority (from *active* to *passive*) and data state (*dirty* to *clean*) in X to maintain data correctness. In the MSI protocol, the communication of both data state and data authority changes are realized by c_0 doing a write-back. As a result, this scenario requires t-states.

In Algorithm 1, SYNTHIA only considers transitions triggered on other memory operations where src has either *dirty* data state or *active* data authority. ownTransitions(*ev*) returns a list of transitions triggered on own memory operation based on *ev*. For example, if *ev* is OtherWrite, then ownTransitions(*ev*) returns valid transitions triggered on OwnWrite. For each returned transition from line 11, SYNTHIA computes the change in cumulative data state and cumulative data authority between destination and source states in *t* and *ot*. IsCumulativeChange first computes a value based on the data state and data authority of the destination states in *ot* and *t* and a value based on the source states in *ot* and *t*, and then returns the difference between the computed values. A non-zero difference means that a core must respond with some operation that requires shared bus access, and hence, requires at least one t-state; otherwise no t-states are required. In Figure 3, consider (M, OtherRead) → S. Line 11 returns *ot* =(I, OwnRead) → S. The source states in *ot* and *t* are M and I and the destination states in *ot* and *t* are both S. Line 13 returns true as the cumulative changes in data authority and data state are not zero, which results in constructing MS_A.

C. *t-states for interleaving memory operations*

Key idea. In the protocol so far, there is no information regarding what a core must do when it has a pending operation on a cache line *and* observes interleaving memory operations from other cores on the same cache line. For example, consider a core that has a cache line in IM_D that is waiting to receive the requested data to complete its pending OwnWrite. Notice that there are no transitions in Figure 3 that determine what this core should do on observing OtherWrite or OtherRead on the same cache line. This scenario can occur as it may take several cycles for the core to receive the requested data during which multiple cores can perform operations on the same cache

A. M. Kaushik and H. Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols," in proceedings of IEEE Design Automation and Test in Europe (DATE), Mar. 2021, pp. 1–6. doi: 10.23919/DATE51398.2021.9474238.

Algorithm 2 t-states for interleaving memory operations

```
1:  procedure isTSNeededInterleavingMemOps(t, ts)
2:      src = t.source(), dst = t.destination(), ev = t.event()
3:      for all oev ∈ {OtherRead, OtherWrite} do
4:          if isPreOrdered(ts) then
5:              newDst = getDst(src, oev)
6:              if newDst ≠ dst then
7:                  if isOther(ev) then
8:                      return isTSNeededBusComm(t)
9:                  return true
10:         if isPostOrdered(ts) then
11:             newDst = getDst(dst, oev)
12:             if newDst ≠ dst then
13:                 return true
14:     return false
```

line. One solution is to *stall* any state changes to a cache line in a t-state until it transitions to its destination s-state. This solution trades simple protocol design for reduced performance as it introduces stalls. On the other hand, minimizing stalling while still maintaining predictability requires careful analysis of state changes due to interleaving other memory operations on a cache line in a t-state. In this step, we perform such analysis to construct t-states and transitions that capture the correct order of state changes due to interleaving memory operations. SYNTHIA relies on Algorithm 1 to achieve this minimal stalling while still maintaining predictability.

Mechanism. For this analysis, we first classify t-states into two categories based on the *relative ordering* of other memory operations observed by a cache line on the shared bus: *pre-ordered* and *post-ordered* t-states. Algorithm 2 constructs t-states based on this classification. The algorithm takes as input a t-state (ts) and the transition on which this t-state lies on (t). For both categories, t-states are not required if there is no state change due to interleaving other memory operations.

Pre-ordered transient states. A cache line is in a *pre-ordered* t-state if the core's pending memory operation is *not* yet ordered on the snooping bus. For example, **_AD** and **_A** states are pre-ordered t-states as they wait for the core's memory operation to be ordered on the snooping bus. A cache line in a pre-ordered t-state observes interleaving memory operations from other cores on the bus (if any) *before* it sees its own memory operation ordered on the bus. As a result, a cache line in a pre-ordered t-state reacts to other memory operations (if any) *in the same way as if the cache line is in the source state*. For example, IM_AD, which lies on (I, OwnWrite) → M, reacts to other memory operations in the same way as I.

Lines 4-9 in Algorithm 2 describe the conditions for constructing new t-states and transitions for a pre-ordered t-state. In line 5, SYNTHIA applies the other memory operation oev on the source s-state of the transition, and extracts the new destination s-state ($newDst$). A new t-state *may* be required to capture any state change ($newDst \neq dst$) depending on the transition type of t. If t is triggered on an own memory operation, then t-states are required in order to capture the state change, and appropriate transitions to ensure the own memory operation ultimately completes. For example, consider the pre-ordered t-state SM_A, which reacts to other memory operations in the same way as S. On an OtherWrite, a cache line in S invalidates its data contents and moves to I state. Hence, a

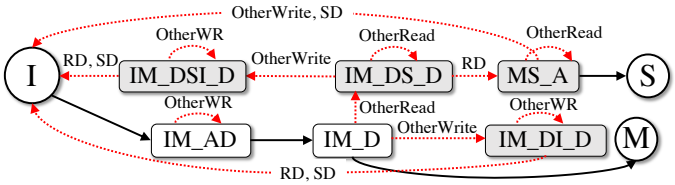

Fig. 4: MSI protocol refinement for interleaving memory operations.

cache line in SM_A must transition to a t-state that conveys that the cache line data contents are invalid and an OwnWrite operation is pending. For transitions triggered on other memory operations, SYNTHIA uses the result of Algorithm 1 to construct necessary t-states (lines 7-8). For example, consider MS_A, which lies on (M, OtherRead) → S. An OtherWrite on M transitions to $nextDst$ =I. While a new t-state can be constructed, applying Algorithm 1 shows that this is not required; isTSNeededBusComm on (M, OtherWrite) → I returns false as described in Section III-B.

Post-ordered transient states. A cache line is in a *post-ordered* t-state *after* the core's pending memory operation is ordered on the snooping bus. Hence, other memory operations (if any) are ordered *after* the core's pending memory operation. A cache line in a post-ordered t-state reacts to other memory operations on the cache line *in the same way as if the cache line is in the destination state*. **_D** states are post-ordered t-states. For example, IM_D on (I, OwnWrite) → M reacts to other memory operations in the same way as M.

Lines 10-13 in Algorithm 2 describe the conditions for post-ordered t-states. In contrast to pre-ordered t-states, SYNTHIA applies other memory operations on the destination s-state (line 11). Consider IM_D. An OtherRead on M transitions to S. Hence, SYNTHIA constructs a new t-state IM_DS_D as shown in Figure 4. A core that has a cache line in IM_DS_D completes the pending OwnWrite on receiving the requested data, and finally transitions to S.

SYNTHIA uses Algorithm 1 to decide whether a post-ordered t-state transitions to the destination s-state directly or through other t-states. For example, consider a core that has a cache line in IM_DS_D. On receiving data, the core must complete the pending write operation, *write-back* the updated data contents, send the data to the requesting core, and transition to the final destination s-state S. Since, there is an operation that requires shared bus access (write-back), IM_DS_D cannot directly transition to S, and must first transition to a t-state (MS_A) to indicate pending write-back as shown in Figure 4. In this case, IM_DS_D lies on M and S, and Algorithm 1 returns true for (M, OtherRead) → S (details in Section III-B).

D. Replacements and shared memory protocol

Replacements to cache lines with *dirty* data state or *active* data authority must write-back data to the shared memory or inform the shared memory regarding change in data authority respectively. Hence, such cache line replacements require t-states; otherwise, no t-states are required. The shared memory protocol consists of two s-states to denote the cache line data state, and a t-state that waits for a core to write-back data. Additional memory states are dependent on the input s-states.

A. M. Kaushik and H. Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols," in proceedings of IEEE Design Automation and Test in Europe (DATE), Mar. 2021, pp. 1–6. doi: 10.23919/DATE51398.2021.9474238.

TABLE I: Evaluation of SYNTHIA on different protocols. SYNTHIA took less than a few seconds to construct the protocols.

Protocol	Input		SYNTHIA output		Validation		Stalling transitions based on Algorithm 2		
	States	Transitions	States	Transitions	Correctness	Exhaustive testing	Disabled	Only pre-ordered	Only post-ordered
MSI	3	14	17	66	✓	✓	12 of 36	4 of 39	8 of 48
MSI-P	3	14	17	64	✓	✓	14 of 39	4 of 39	10 of 51
MESI	4	19	26	96	✓	✓	18 of 51	6 of 57	12 of 72
MESI-P	4	19	26	95	✓	✓	22 of 57	6 of 57	16 of 78
MOESI	5	24	27	103	✓	✓	18 of 57	6 of 60	12 of 78

IV. RESULTS

Evaluation of SYNTHIA. SYNTHIA successfully constructs non-stalling and predictable coherence protocols from s-states specifications of MSI, MESI, and MOESI protocols [5]. Table I shows the number of states and transitions in the input and output. All the states in MSI-P and MESI-P protocols have passive data authority. As a result, all data communication between cores in these protocols are through the shared memory. The predictable and high-performance protocol of MSI-P is the PMSI protocol [1]. A key takeaway is the significant increase in the number of states and transitions in order to achieve predictability and high-performance. For example, a predictable and high-performance MOESI implementation has more than 5× the number of states and transitions compared to the input specification. Hence, SYNTHIA relieves the design burden on a protocol designer by automating the analysis and protocol construction. We validated the protocols generated by SYNTHIA against manually implemented verified versions of the protocols [1]. We found that the states and transitions in the protocols generated by SYNTHIA matched the manually implemented versions. We also checked their correctness, predictability, and performance through exhaustive testing using the gem5 simulator [9].

We perform the following experiment to highlight how SYNTHIA improves the productivity of protocol designers. Suppose a protocol designer manually designed the protocols listed in Table I, and missed certain analyses that account for interleaving other memory operations to the same shared data (Algorithm 2). The missing analyses result in stalled transitions in the output protocol, which limit the performance of the cache coherence protocol. Consider a case where a protocol designer does not perform *any* of the analyses outlined in Algorithm 2. For the MSI protocol, we observed that 12 transitions out of total 36 transitions are stalling transitions, which constitutes more than 30% of the transitions (highlighted in Table I). Across all protocols, we observed more than 30% of the transitions in the output protocols are stalling transitions. If a designer accounts for only one type of t-states (post-ordered or pre-ordered), then 9%-16% of the transitions in the constructed protocols are stalling transitions. On the other hand, protocols generated by SYNTHIA have *no* stalling transitions due to interleaving memory operations from other cores.

Predictability and performance evaluation. We manually converted the states, transitions, and actions in the generated protocols into the SLICC syntax, and evaluated them using the gem5 micro-architectural simulator [9]. We used the synthetic workloads from [1] and verified the data correctness of the protocols. We modeled a 8-core multi-core platform where the shared bus deploys a TDM arbitration. Each core is allocated one TDM slot. Table II shows the maximum observed worst-case latency (WCL) experienced by a memory request under the predictable cache coherence protocols, and the average-performance speedup of the protocols compared to a cache bypassing technique. The cache bypassing technique disables private caching of shared data. From Table II, the observed WCL across all protocols are within their derived analytical WCL bound, and the generated cache coherence protocols outperform (as high as 3.94×) the cache bypassing technique while achieving predictability.

TABLE II: Predictability and performance evaluation.

Protocol	Predictability (WCL in cycles)		Performance
	Observed WCL	Analytical WCL	Speedup
MSI	3061	6450	3.44×
MESI	3214	6450	3.38×
MOESI	2019	6450	3.94×
MSI-P	6364	7250	1.72×
MESI-P	5964	7250	1.69×

V. CONCLUSION

We present SYNTHIA, an automated tool for constructing correct, predictable and high-performance cache coherence protocols. SYNTHIA automates the analyses that identifies scenarios involving interleaving memory operations from multiple cores to shared data and that require access to the shared bus. SYNTHIA refines the input protocol using the analysis by adding new states and transitions that achieve predictability and high-performance. We validated the correctness, predictability, and performance of the protocols generated by SYNTHIA, and confirmed that the states and transitions in the generated protocols matched manually implemented versions.

REFERENCES

[1] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *RTAS*, 2017.

[2] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "CARP: A Data Communication Mechanism for Multi-Core Mixed-Criticality Systems," in *RTSS*, 2019.

[3] N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *RTSS*, 2019.

[4] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications," in *ISCA*, 2018.

[5] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis lectures on computer architecture*, 2011.

[6] N. Sensfelder, J. Brunel, and C. Pagetti, "On How to Identify Cache Coherence: Case of the NXP QorIQ T4240," in *ECRTS*, 2020.

[7] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets," in *PLDI*, 2013.

[8] M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan, "Verc3: A library for explicit state synthesis of concurrent systems," in *DATE*, 2018.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, 2011.