Automatic Construction of Predictable and High-Performance Cache Coherence Protocols for Multicore Real-Time Systems

Anirudh Mohan Kaushik, *Member, IEEE*, and Hiren Patel, *Member, IEEE*

***Abstract*—Predictable hardware cache coherence is a viable shared data communication mechanism between cores for multicore real-time platforms. Prior works have established that predictable hardware cache coherence protocols offer significant performance advantages over alternative predictable data communication mechanisms while ensuring predictability. Unlike alternative predictable data communication mechanisms, designing predictable cache coherence protocols is nontrivial as it requires detailed understanding of the impact of different memory activity patterns to shared data for predictable and coherent data communication. Furthermore, designing predictable cache coherence protocols that deliver high average-case performance is even more challenging as it entails identifying opportunities such that a core's access to a data is not stalled in the presence of interleaving memory activity from other cores to the same data. To this end, we present SYNTHIA, an open and automated tool for synthesizing predictable and high-performance snooping bus-based cache coherence protocols for multicore platforms deployed in real-time systems. SYNTHIA automates the complex analysis associated with designing predictable and high-performance cache coherence protocols, and constructs the complete protocol implementation (coherence states and transitions) that achieve predictability and performance. We use SYNTHIA to construct complete protocol implementations from simple specifications of common protocols (modified-shared-invalid (MSI), MESI, and MOESI protocols) and a predictable variant of the MESIF cache coherence protocol, which was recently found to be deployed in an existing multicore platform designed for real-time platforms. We validated the correctness, predictability, and performance guarantees of the generated protocol implementations from SYNTHIA using manually implemented versions, and a micro-architectural simulator.**

***Index Terms*—Automated protocol synthesis, hardware cache coherence, predictability.**

I. INTRODUCTION

HARDWARE cache coherence protocols for multicore platforms used for real-time and safety-critical systems have recently become an attractive solution for predictably managing shared data communication between the multiple cores [1]–[5]. 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 multicore platforms [1], [2] while also providing worst-case latency (WCL) 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 [6]. 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 WCL 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], [7]. For example, the modified-shared-invalid (MSI) protocol with no additional support for predictability or high performance has three 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

Manuscript received 24 December 2020; revised 23 May 2021 and 31 August 2021; accepted 23 September 2021. Date of publication 26 October 2021; date of current version 20 September 2022. This article was recommended by Associate Editor M. Mutyam. *(Corresponding author: Anirudh Mohan Kaushik.)*

The authors are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L3G1, Canada (e-mail: amkaushi@uwaterloo.ca).

Digital Object Identifier 10.1109/TCAD.2021.3123056

protocol implementation that achieves predictability and high performance.

Our previous work described high-level details about SYNTHIA's mechanism and applied SYNTHIA to three popular coherence protocols (MSI, MESI, and MOESI coherence protocols). In this work, we elaborate SYNTHIA's mechanism that constructs the protocol implementations and describe the construction of a predictable variant of a currently implemented coherence protocol. We extend our previous work [8] in two ways. First, we expand on SYNTHIA's mechanism that constructs the predictable and high-performance protocol implementation (Section V). Our previous work [8] described the conditions under which new states and transitions were constructed, and did not describe the type of states and transitions constructed. Second, we use SYNTHIA to construct a predictable variant of the modified-exclusive-shared-invalid-forward (MESIF) protocol (Section VI), and evaluate the predictability and performance of predictable MESIF (PMESIF) protocol using the gem5 micro-architectural simulator [9]. Recent reverse-engineering efforts by Sensfelder *et al.* [10] found that the NXP QorIQ multicore platforms deployed the MESIF cache coherence protocol. We describe in detail the construction of the PMESIF protocol by SYNTHIA and highlight key predictability and performance features of the PMESIF protocol.

Our main contributions in this work are as follows.

1) We present an approach to automatically construct predictable and high-performance snooping bus-based cache coherence protocols.
2) We implement our approach in a tool called SYNTHIA. The input to SYNTHIA is a simple protocol specified in a domain-specific language SYNTHIADSL only using stable states. SYNTHIA uses the input protocol specification and 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. The key operation in SYNTHIA is careful analysis of scenarios that require access to the shared bus, and allowing simultaneous interleaving memory operations on the same data to proceed without stalling.
3) We evaluate SYNTHIA by generating predictable and high-performance protocol implementations for several common protocols, such as the MSI, MESI, MOESI, and MESIF coherence protocol [7], [10]. 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 using the gem5 micro-architectural simulator. SYNTHIA is available at https://github.com/caesr-uwaterloo/Synthia.

II. BACKGROUND

Hardware Cache Coherence: A cache coherence mechanism avoids data incoherence by deploying a set of rules to ensure that cores access and cache the correct version of data at all times. While a cache coherence mechanism can be realized in either software or hardware, this work focuses on hardware cache coherence mechanisms. This is because hardware

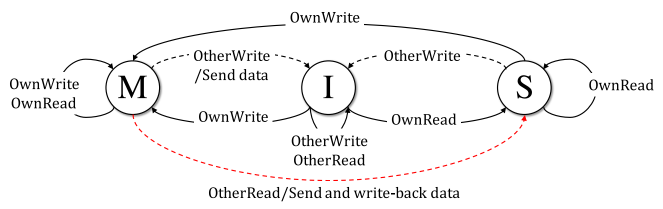

Fig. 1. MSI cache coherence protocol.

cache coherence mechanisms are staple in existing multicore platforms [11]. The main component in a hardware cache coherence mechanism is a *cache coherence protocol*, which is a state machine that deploys the set of rules. Typically, the coherence protocol maintains data coherence at cache line granularity, which is a fixed-size collection of data. The protocol state machine implements these rules with a set of *coherence states* that convey access permissions (read and write) and other information about the cache line data, and *coherence state transitions* between states are triggered based on the memory activity of cores on the shared data. On a state transition, the cache controller executes actions to ensure data correctness. For example, exchanging coherence messages between cores and the shared memory [7]. The cores' cache controllers implement the coherence protocol, and the coherence state information for the cache lines are maintained in the caches' tag arrays.

MSI Cache Coherence Protocol: Cache coherence protocols deployed in current multicore platforms consist of three fundamental stable states, which establish the MSI protocol: 1) *modified* (**M**); 2) *shared* (**S**); and 3) *invalid* (**I**) [7]. Fig. 1 shows the MSI protocol state machine at the private cache memory level. A cache line in **M** means that the current core has written to it and it did not propagate the updated data to the shared memory yet. Only one core can have a specific cache line in a modified state, and is referred to as the *owner*. A core that has a cache line in **M** and observes remote memory activity on the cache line must update the cache line copy in the shared memory. This operation is called *write-back* to shared memory. A cache line in **S** means that the core has a valid, yet unmodified version of that line. One or more cores can have versions of the same cache line in a shared state to allow for fast read accesses. Cores that have the same cache line in the shared state are referred to as *sharers*. This constraint of one owner for a cache line or multiple cores sharing a cache line is referred to as the single-writer multiple-reader (SWMR) invariant [7]. A cache line in **I** denotes the unavailability of that line in the cache or that its data is outdated and no longer valid. Transitions between different states are triggered on memory activity as shown in Fig. 1. For example, consider a core that wants to write to a cache line that it has in the invalid state. This cache line transitions from the invalid state to the modified state on the write request (OwnWrite). Other cores that may have the same cache line in the shared state, observe this write request as an OtherWrite memory activity and transition to the invalid state to preserve the SWMR invariant.

Snooping Bus-Based Cache Coherence: Cores' cache controllers and the shared memory change the state of their cache line copies based on the observed memory activity. In snooping bus-based mechanisms, cores *broadcast* their memory activity on a shared bus, and cores and the shared memory observe memory activity on a cache line by *snooping* this shared bus. Hence, the snooping bus is the *ordering point* for all memory activity. The snooping bus-based coherence mechanisms are typically used for multicore platforms with small core count (4–16 cores) due to bus scalability limits [7], which is the case in current real-time systems [10], [12].

Memory Activity Under Snooping Bus-Based Cache Coherence: The following example describes a core's memory activity to a cache line under a snooping bus-based cache coherence mechanism. We use the notation Z to denote a cache line with address Z. Consider the following scenario: a core c_0 *issues* a read to Z. The read to Z first checks c_0's private cache for the data contents of Z. On a private cache hit, the necessary data is supplied, and the read is marked complete. Private cache hits do not generate coherence activity. On a cache miss, the private cache controller of c_0 *generates* a coherence message. The cache controller then *broadcasts* the appropriate coherence message based on the request type. A coherence message is said to be *ordered* on the bus when all cores and the shared memory observe the corresponding memory request on the bus.

Stable and Transient Coherence States: There are two types of coherence states: 1) *stable* states (s-states) and 2) *transient* states (t-states) [7]. 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. 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 denotes 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 denotes that a core has observed its memory request on the snooping bus, and is waiting for the cache line data contents. **_A** t-states denotes that a core has the cache line data, and is waiting for its memory operation to be ordered on the snooping bus. As an example, consider a core that performs a read operation on a cache line that it does not have in its private cache under the MSI protocol. The starting stable state of the cache line is **I**. The core issues a coherence message and changes the coherence state of the cache line to t-state **IS_AD**, which denotes that a core has issued a coherence message for its read request and is waiting for the message to be ordered on the bus and the corresponding data response. When the coherence message is ordered on the snooping bus, the core changes the coherence state to t-state **IS_D**, which denotes that a core has observed its ordered coherence message, and is waiting for the data response. On receiving the data response, the cache line coherence state transitions to the stable coherence state **S**, and the core completes the read memory operation.

In addition to capturing pending memory state information, t-states also capture information regarding coherence state changes due to *interleaving* memory activity from other cores to the same cache line. Interleaving memory activity from other cores to the same cache line is possible due to the *nonatomic* implementation of the snooping bus [7]. Nonatomic snooping bus is preferred and implemented in existing multicore platforms due to their performance benefits [7]. There are two ways to deal with interleaving memory activity on the same cache line. In the first approach, a core that has a cache line in a t-state can *stall* any coherence state changes until it completes its pending memory operation. However, such a protocol design will have poor performance as it introduces stalls. The second approach is a nonstalling approach wherein t-states can capture the impact of the interleaving memory activity to a cache line on a core's pending memory operation on the same cache line. The following example illustrates this information capture. Consider c_0 has a cache line in t-state **IS_D**. c_0, all other cores, and the shared memory have observed c_0's read coherence message, and c_0 is waiting for the data response. While c_0 is waiting for the data response, another core c_1 broadcasts a coherence message based on a write request to the same cache line. Since c_1 is performing a write operation, c_0 must invalidate the cache line on receiving the data response to maintain the SWMR invariant. As a result, c_0 moves the cache line from t-state **IS_D** to t-state **IS_DI**. t-state **IS_DI** denotes that a core is waiting for the requested cache line data contents, and on receiving the cache line completes the read operation and moves to the **I** state. Note that this approach introduces additional t-states to eliminate stalling, and hence, offers better performance compared to the stalling-based approach.

III. Related Works

A. Predictable Hardware Cache Coherence

Predictable hardware cache coherence protocols ensure that there is a WCL bound on memory accesses across all cores [1]–[5]. These protocols are deployed on a multicore 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 nonatomic split transaction bus [7]. 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 multicore 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 [1], [2], [4] 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

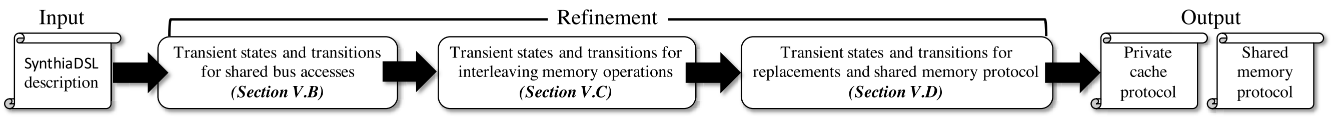
Input
Refinement
Output
SynthiaDSL description
Transient states and transitions for shared bus accesses (Section V.B)
Transient states and transitions for interleaving memory operations (Section V.C)
Transient states and transitions for replacements and shared memory protocol (Section V.D)
Private cache protocol
Shared memory protocol

Fig. 2. High level overview of SYNTHIA.

be high making it an error-prone process. SYNTHIA relieves this complexity burden by *automating* the analysis, and the construction of correct, predictable, and high-performance coherence protocols. A designer provides SYNTHIA a protocol specification using only s-states and SYNTHIA automatically constructs a correct, predictable, and high-performance coherence protocol.

B. Cache Coherence Protocol Synthesis

Oswald *et al.* [6] presented ProtoGen, an automated tool that constructed high-performance directory-based cache coherence protocols. The input to ProtoGen was an atomic specification of a directory cache coherence protocol specified using stable coherence states. ProtoGen refined the input atomic specification by adding new transient states and transitions using domain knowledge built into the tool. The output of ProtoGen was a nonstalling directory protocol implementation. 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 [7]. This is because of differences in coherence message communication (broadcast versus unicast) and ordering mechanisms (bus versus directory) [7]. This results in different protocol construction mechanisms. 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 multicores. Furthermore, SYNTHIA is better positioned than ProtoGen for multicore platforms with lower core counts where snooping-bus-based protocols offer better average-case performance than directory-based protocols [7].

Alternate protocol synthesis tools, such as Transit [13] and VerC3 [14] 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 a frequent designer intervention to add information to the input specification for correct protocol construction [6]. We take an alternative approach by embedding domain knowledge about predictable high-performance snooping bus-based protocols into SYNTHIA to automate the protocol construction. Furthermore, 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.

IV. SYSTEM MODEL

We consider a multicore model with N cores $C = \{c_0, c_1, \ldots, c_{N-1}\}$. Cores have a memory hierarchy consisting of private level one (L1) data and instruction caches, and a shared memory. The shared memory (LLC) contains all the data required by the tasks running on the cores, and the private caches hold a subset of the tasks' data. Real-time tasks are mapped for execution across cores, and communicate shared data with each other. We assume that all real-time tasks are safety-critical tasks and must meet their timing requirements.

The multicore model deploys a predictable snooping bus-based cache coherence protocol [1]. As a result, the multicore model consists of the hardware structures described in [1] that work in tandem with the cache coherence protocol to enable predictable data communication. Cores broadcast coherence messages for their memory operations on the shared snooping bus, which we refer to as the message bus. Cores can communicate data with each other via the shared memory or point-to-point interconnects between the cores' L1 caches. Data communication between cores and shared memory is through shared data bus interconnects. The shared bus interconnects are implemented as nonatomic split-transaction buses [7]. The shared request and data bus interconnects deploy a predictable arbitration policy to predictably manage simultaneous accesses from cores. While the output protocol specification from SYNTHIA is not tied to a particular arbitration policy, the evaluation in Section VII uses TDM predictable bus arbitration to empirically validate the predictability guarantees of the predictable cache coherence protocols generated by SYNTHIA.

V. SYNTHIA IMPLEMENTATION

Fig. 2 presents an overview of SYNTHIA. SYNTHIA takes as input a protocol specification written in SYNTHIADSL (Section V-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 V-B) and 2) transitions that change due to interleaving memory operations on the same cache line (Section V-C). We explain the construction of transitions due to t-states using examples. After refinement, SYNTHIA outputs the cache coherence protocol state machines at the private cache level and shared memory. Table I describes the routines used in the protocol construction algorithms.

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

TABLE I
DESCRIPTION OF ROUTINES USED IN SYNTHIA FOR PROTOCOL CONSTRUCTION

Routine	Description
NEWTRANSIENTSTATE(x, y, z)	Construct new transient state of the form xy_z and sets state encoding of xy_z to be equal to x if pre-ordered t-state or y if post-ordered t-state.
NEWTRANSITION(x, y, e)	Construct new transition from state x to state y triggered on event e.
t.SOURCE()	Return the source state for transition t.
t.DESTINATION()	Return the destination state for transition t.
t.EVENT()	Return the triggering event for transition t.
t.SETDESTINATION$(()d)$	Sets the destination state of transition t to d.
ISOWN(ev)	Returns true if ev is an own event (OwnReadM, OwnRead, OwnWrite), false otherwise.
ISINVALID(s)	Returns true if access permissions of s is $invalid$, false otherwise.
ISDIRTY(s)	Returns true if data state of s is $dirty$, false otherwise.
ISACTIVE(s)	Returns true if data authority of s is $active$, false otherwise.
GETDST(s, ev)	Returns the destination s-state on applying event ev on source s-state s.
OWNTRANSITIONS(ev)	Returns the set of transitions triggered on other event ev. For example if ev is OtherWrite, then routine returns transitions triggered on OwnWrite.
ISCUMULATIVECHANGE(t_1, t_2)	Returns true if the cumulative data state or data authority across the source states in t_1 and t_2 are different from the cumulative data state or data authority across the destination states in t_1 and t_2, false otherwise.
ISTSNEEDEDFORBUSCOMM(t)	Returns true if t-states are needed on a transition for shared bus communication, false otherwise.

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

Fig. 3. MSI Protocol Specification in SYNTHIADSL.

of s-states and 2) transitions between s-states. Fig. 3 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* ∈ {invalid, read, write, exread} is the access permission, *ds* ∈ {clean, dirty} is the data state of the cache line, and *da* ∈ {active, passive} 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. *read* denotes that a core can read the cache line data contents, and *write* denotes that a core can read and write the cache line data contents. Under the SWMR invariant [7], at any instance of time only one core can have a cache line with write access permissions or multiple cores can have the cache line with read access permissions. *exread* denotes that a core can read the cache line data contents, and is the only core (exclusive) that has the cache line. 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. For the MESI and MOESI protocols, the state encoding of `E` and `O` states are (*exread*, *dirty*, *active*) and (*read*, *dirty*, *active*), respectively. 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. In Section VI, we show the construction of the PMESIF coherence protocol; MESIF protocol is deployed in NXP QorIQ multicore processors [10].

Transitions: $(\mathtt{src}, \mathtt{ev}) \rightarrow \mathtt{dst}$ is a transition where `src` is the source s-state, `dst` is the destination s-state, and `ev` ∈ {OwnReadM, OwnRead, OwnWrite, OtherRead, OtherWrite, Replacement} is the event that triggers the transition. OwnReadM event denotes a core's own read request *issued* to a cache line, and the core receives the data response from the shared memory. OwnRead event denotes a core's own read request *issued* to a cache line, and the core receives the data response from its cache (cache hit) or from another core, respectively. OwnWrite event denotes a core's own write request issued to a cache line, and the core receives the data response from the shared memory or its own cache (cache hit) or another core. 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. Lines 4–17 in Fig. 3 define the state transitions in the MSI protocol. For example, consider $(\mathtt{I}, \text{OwnReadM}) \rightarrow \mathtt{S}$. 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 from the shared memory, 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 MSI protocol has the same source and destination states for OwnReadM and OwnRead events (**I** and **S** states). The MESIF protocol described in Section VI has different destination states for OwnReadM and OwnRead events.

Notice that the input SYNTHIADSL specification does not have: 1) transitions triggered when a core observes its own memory operation; 2) transitions triggered when a core receives the requested data; and 3) actions that a core executes as a consequence of a transition, such as sending data to another core (SD) and write-back data to shared memory (WD). These information is added by SYNTHIA during protocol construction. SYNTHIA automatically adds transitions triggered when a core's own memory operation is ordered on

Algorithm 1 Construction of t-states and Transitions Due to Shared Bus Communication

```
 1: procedure CONSTRUCTTSANDTRANSITIONSFORBUSCOMM($t$)
 2:     $src = t$.SOURCE(), $dst = t$.DESTINATION(), $ev = t$.EVENT()
 3:     if ISOWN($ev$) then
 4:         if ISINVALID($src$) $\lor$ (ISCLEAN($src$) $\land$ $ev$ == OwnWrite) then
 5:             $s_1$ = NEWTRANSIENTSTATE($src$, $dst$, _AD)
 6:             $s_2$ = NEWTRANSIENTSTATE($src$, $dst$, _D)
 7:             $t$.SETDESTINATION($s_1$)
 8:             $t_1$ = NEWTRANSITION($s_1$, $s_2$, Ordered)
 9:             $t_2$ = NEWTRANSITION($s_2$, $dst$, RD)
10:     else if ISDIRTY($src$) $\lor$ ISACTIVE($src$) then
11:         $tList$ = OWNTRANSITIONS($ev$)
12:         for all $ot \in tList$ do
13:             if ISCUMULATIVECHANGE($ot$, $t$) $\neq 0$ then
14:                 $s_1$ = NEWTRANSIENTSTATE($src$, $dst$, _A)
15:                 $t$.SETDESTINATION($s_1$)
16:                 $t_1$ = NEWTRANSITION($s_1$, $dst$, Ordered)
```

the snooping bus (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 and Transitions Due to Shared Bus Communication*

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 on the shared bus.

Mechanism: Algorithm 1 describes the construction of t-states and transitions for shared bus communication; this algorithm expands the ISTSNEEDEDBUSCOMM routine in [8] that only describes the conditions under which SYNTHIA constructs t-states and transitions. The input to Algorithm 1 is a transition t, and it is applied to each transition in the input SYNTHIADSL specification. This algorithm exploits two key insights. First, for transitions triggered on own memory operations (line 3), t-states are required only when :1) `src` has *invalid* access permissions and `src` $\neq$ `dst` (lines 6–11) or 2) `src` has *clean* data state and the operation is OwnWrite (lines 12–15). 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 13–16). New transitions are required for cases that introduce new t-states. Using Fig. 4 as an illustrative example, we explain this implementation.

Consider Insight (1): If `src` has *invalid* access permissions (ISINVALID returns true), then `src` does not have the cache line data contents to complete its own memory operation (lines 6–11). 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. Lines 7 and 8 construct **_AD** and **_D** t-states, and lines 9–11 constructs the transitions due to these new t-states. In Fig. 4, (`I`, OwnRead) → `S` has t-states `IS_AD` and `IS_D` where `IS_AD` waits for the coherence message

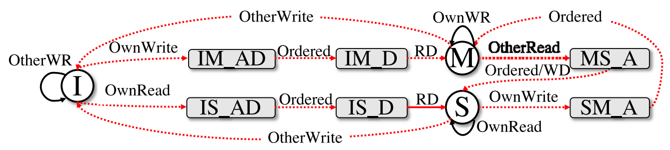

Fig. 4. MSI protocol refinement for communication on the shared bus. Constructed t-states and transitions are highlighted.

broadcast to be ordered and `IS_D` waits for the requested data. The original transition (`I`, OwnRead) → `S` is changed to (`I`, OwnRead) → `IS_AD`, (`IS_AD`, Ordered) → `IS_D`, and (`IS_D`, RD) → `S`. There are two possible formulations of transient states for transitions (`src`, OwnWrite) → `dst` where `src` has *clean* data state depending on whether upgrades are supported. When upgrades are supported, `src` has valid cache line data and hence, only requires the OwnWrite operation to be broadcasted and ordered on the snooping bus for `src` to transition to `dst`. This requires one **_A** transient state to denote waiting for OwnWrite to be ordered on the snooping bus. However, on an intervening write operation from another core, the cache line in **_A** transient state must invalidate and reissue its write operation, which adds complexity to the protocol design [7]. Hence, SYNTHIA takes the second formulation where any transition to `dst` with write access permissions must receive the data before completing the write operation irrespective of whether `src` has valid cache line data contents. Therefore, for transitions (`src`, OwnWrite) → `dst` where `src` has *clean* data state, SYNTHIA also constructs **_AD** and **_D** states. Receiving the cache line data contents before completing the OwnWrite operation simplifies the protocol design when taking into account interleaving memory operations from other cores to the same cache line [7].

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. On the other hand, (`M`, OtherWrite) → `I` does not require a write-back to shared memory, and the core that has the cache line in **M** can send the data to the requesting core.

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 Fig. 3. 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 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. If c_0 performs a write-back to shared memory, c_0's updates to X will be overwritten by c_1's write operation to X. Furthermore, c_0 need not inform the shared memory about the change in its data authority of X. This is because c_1 receives X with *active* data authority, and hence, c_1 responds to subsequent requests to 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 to a shared memory; the `M` state has *dirty* data state and *active* data authority. As a result, this scenario requires t-states. Note that conventional cache coherence protocols do not need t-states in this scenario. This is because conventional protocols are deployed on multicore models where the shared bus does not allocate exclusive time slots to cores. Hence, from a cache coherence protocol standpoint, a cache line can directly transition to a stable state and the core can send the data responses on observing other memory operations on the shared bus [7].

In Algorithm 1, SYNTHIA only considers transitions triggered on other memory operations where `src` has either *dirty* data state or *active* data authority. Transitions triggered on other memory operations cannot upgrade their data state from *clean* to *dirty* and data authority from *passive* to *active*. 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 nonzero 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 Fig. 4, consider (`M`, OtherRead) $\rightarrow$ `S`. Line 11 returns *ot* $=$ (`I`, OwnRead) $\rightarrow$ `S` and (`S`, OwnRead) $\rightarrow$ `S`. *ot* $=$ (`S`, OwnRead) $\rightarrow$ `S` violates the SWMR invariant as one core has the cache line `M` and another core has the cache line in `S` simultaneously. Hence, the only valid *ot* is (`I`, OwnRead) $\rightarrow$ `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`. The original transition (`M`, OtherRead) $\rightarrow$ `S` is replaced with (`M`, OtherRead) $\rightarrow$ `MS_A` and (`MS_A`, Ordered) $\rightarrow$ `S`. Since `M` has *dirty* data state and *active* data authority, the transition (`MS_A`, Ordered) $\rightarrow$ `S` is causes the core with the cache line in `MS_A` to write-back the modified data contents to shared memory (WD) and send the data to the requesting core. On the other hand, consider (`M`, OtherWrite) $\rightarrow$ `I`. The valid *ot* returned in line 11 is (`I`, OwnWrite) $\rightarrow$ `M`. In this case, the source states in *ot* and *t* are `M` and `I` and the destination states in *ot* and *t* are `I` and `M`, respectively. Line 13 returns `false` as there are no cumulative changes in data authority and data state. Hence, this transition does not have t-states.

C. *t-States and Transitions Due to Interleaving 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 Fig. 4 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 line while it is in `IM_D`. 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 a 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 memory operations on a cache line from other cores 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: *preordered* and *post-ordered* t-states. Recall from Section II that a memory operation is ordered when all cores and shared memory observe the coherence messages corresponding to the memory operation on the snooping bus. Hence, *preordered* t-states are t-states that appear before the coherence messages corresponding to the memory operation are observed on the snooping bus and *post-ordered* t-states are t-states that appear after the coherence messages corresponding to the memory operation are observed on the snooping bus. 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 stable state change due to interleaving other memory operations. For example, (`I`, OtherRead) $\rightarrow$ `I` and (`S`, OtherRead) $\rightarrow$ `S` do not need t-states as the stable states do not change due to interleaving memory operations.

Pre-Ordered t-States: A cache line is in a *preordered* t-state if the core's pending memory operation is *not* yet ordered on the snooping bus. **_AD** and **_A** t-states are preordered t-states as they wait for the core's memory operation

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 $oev \in$ {OtherRead, OtherWrite} do
4:          if ISPREORDERED($ts$) then
5:              $newDst$ = GETDST($src$, $oev$)
6:              if $newDst \neq dst$ then
7:                  if ISOWN($ev$) then
8:                      if ISINVALID($newDst$) then
9:                          $s_1$ = NEWTRANSIENTSTATE($newDst$, $dst$, _AD)
10:                         NEWTRANSITION($ts$, $s_1$, $oev$)
11:                     else
12:                         $s_1$ = NEWTRANSIENTSTATE($newDst$, $dst$, _A)
13:                         NEWTRANSITION($ts$, $s_1$, $oev$)
14:                 else
15:                     $s_1$ = NEWTRANSIENTSTATE($newDst$, $newDst$, _A)
16:                     NEWTRANSITION($ts$, $s_1$, $oev$)
17:                     NEWTRANSITION($s_1$, $newDst$, Ordered)
18:             else
19:                 NEWTRANSITION($ts$, $ts$, $oev$)
20:         if ISPOSTORDERED($ts$) then
21:             $newDst$ = GETNEWDST($dst$, $oev$)
22:             if $newDst \neq dst$ then
23:                 $s_1$ = NEWTRANSIENTSTATE($ts$, $newDst$)
24:                 NEWTRANSITION($ts$, $s_1$, $oev$)
25:                 $nt$ =(dst, $oev$) → newDst
26:                 if ISTSNEEDEDBUSCOMM($nt$) then
27:                     $s_2$ = NEWTRANSIENTSTATE($dst$, $newDst$, _A)
28:                     NEWTRANSITION($s_1$, $s_2$, RD)
29:                 else
30:                     NEWTRANSITION($s_1$, $newDst$, RD)
31:             else
32:                 NEWTRANSITION($ts$, $ts$, $oev$)
```

to be ordered on the snooping bus. For example, IM_AD and MS_A are examples of preordered t-states. IM_AD waits for the coherence message for its write request to be ordered on the bus and the requested data, and MS_A waits for the coherence message for its data write-back to shared memory to be ordered on the bus. A cache line in a preordered t-state observes interleaving memory operations from other cores on the bus *before* it sees its own memory operation ordered on the bus. As a result, a cache line in a preordered t-state reacts to other memory operations *in the same way as if the cache line is in the source state*. This means that IM_AD and MS_A react to other memory operations in the same way as I and M, respectively.

Lines 4–19 in Algorithm 2 describe the constructing of new t-states and transitions for a preordered 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 (lines 7–13), 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 preordered 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 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. Remaining in SM_A state on an OtherWrite operation violates the SWMR invariant. On lines 8–10, **_AD** t-state is required as the interleaving memory operation invalidates the cache line data contents (*newDst* has *invalid* access permissions). Lines 15–17 handle interleaving memory operations that change the destination state of transitions triggered on other memory operations. For such cases, SYNTHIA creates a new **_A** t-state with *newDst*, and the corresponding transitions. For example, consider t-states MS_A, which lies on (M, OtherRead) → S. MS_A is waiting for the coherence message to complete its data write-back to shared memory. An OtherWrite on M transitions to $nextDst$ = I, which is different than the current *dst* (S). Hence, SYNTHIA constructs t-state II_A, and the transitions (MS_A, OtherWrite) → II_A and (II_A, Ordered) → I. SYNTHIA adds the send data action to the requesting cores (SD) to the transition (MS_A, OtherWrite) → II_A (not shown in Algorithm 2). Note that II_A on observing the ordered coherence message for the data write-back, which was issued when the cache line was in MS_A, simply transitions to I. For transitions that do not change state due to interleaving memory operations (line 19), SYNTHIA adds a self transition on t-state *ts*. For example, interleaving memory operations on IM_AD behaves in the same way as I, which does not change state. Hence, SYNTHIA adds self transitions (IM_AD, OtherRead) → IM_AD and (IM_AD, OtherWrite) → IM_AD.

Post-Ordered t-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 and IS_D on (I, OwnRead) → S reacts to other memory operations in the same way as M and S, respectively.

Lines 20–30 in Algorithm 2 construct t-states and transitions for post-ordered t-states. In contrast to preordered t-states, SYNTHIA applies other memory operations on the destination s-state (line 21), and constructs t-states and transitions on a state change due to the other memory operations (lines 22–30). Consider IM_D. An OtherRead on M transitions to S. Hence, SYNTHIA constructs a new t-state IM_DS_D as shown in Fig. 5, which captures the state change. This state conveys that a pending write operation on a cache line observed an OtherRead memory operation on the same cache line, and on receiving the requested cache

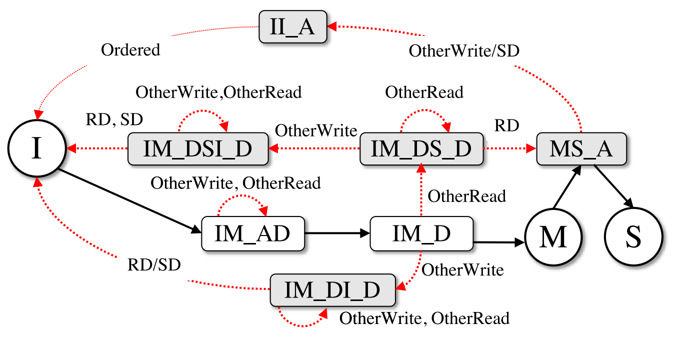

Fig. 5. MSI protocol refinement for interleaving memory operations. Constructed t-states and transitions are highlighted.

line, the final s-state is S. 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. An OtherWrite on M transitions to I. Similarly, SYNTHIA constructs a new t-state IM_DI_D on an OtherWriteas shown in Fig. 5. On receiving data, IM_DI_D will transition to the stable state I. SYNTHIA applies Algorithm 2 on the new post-ordered t-states IM_DS_D and IM_DI_D. Since IM_DI_D lies between M and I, an other memory operation on IM_DI_D reacts in the same way as I. Hence, SYNTHIA creates a self transition for other memory operations on IM_DI_D as shown in Fig. 5.

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 Fig. 5. In this case, IM_DS_D lies on M and S, and ISTSNEEDEDBUSCOMM returns true for $(\text{M}, \text{OtherRead}) \rightarrow \text{S}$ (details in Section V-B). Hence, SYNTHIA constructs t-states MS_A and the transition $(\text{IM\_DS\_D}, \text{RD}) \rightarrow \text{MS\_A}$ to denote that on receiving the requested data, the cache line is marked for write-back. On the other hand, IM_DI_D can directly transition to I on receiving the data as ISTSNEEDEDBUSCOMM returns false for $(\text{M}, \text{OtherWrite}) \rightarrow \text{I}$. This check for shared bus accesses in response to interleaving other memory operations, and the construction of t-states to indicate pending shared bus accesses is a key distinguishing feature between predictable coherence protocols and conventional cache coherence protocols.

D. Handling Replacements, Transition Actions, and Shared Memory Protocol Construction

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, replacements to cache lines with *passive* data authority and *clean* data state do not require t-states and can directly transition to the invalid state. SYNTHIA also generates the protocol state machine at the shared memory. For any input protocol specification, the shared memory must maintain one state to denote if a cache line's data contents are unmodified, and another state to denote if a core has a modified copy of the cache line in its private cache. Since a core may write-back updated data contents of a cache line to shared memory, the shared memory maintains a **_D** state that waits for the pending data communication from the core. Additional shared memory states are dependent on the s-states defined in the input specification.

Finally, for each transition in the core and shared memory protocols, SYNTHIA determines whether any action (SD, WD, and BM) must be executed by a cache controller or shared memory on completing the transition. SD denotes that the cache controller or shared memory must send data to a requesting core, WD denotes that cache controller must send data to the shared memory, and BM denotes a coherence message broadcast. For a transition, the data state and data authority of the source state (s-state or t-state) denotes the type of action executed on exercising the transition. For example, a source state with *dirty* data state will perform a WD action on transitioning to a destination state with *clean* data state.

E. Properties of Protocols Constructed by SYNTHIA

1) Correctness: We prove that the cache coherence protocols generated by SYNTHIA satisfy the SWMR invariant provided the input specification satisfies the SWMR invariant. The SWMR invariant is a key correctness invariant for cache coherence protocols [7]. The key intuition behind the correctness proof is that during each step of the algorithms in SYNTHIA, constructed transitions and transient states do not violate the SWMR correctness invariant.

Theorem 1: If the input specification is correct, then the protocol implementation constructed by SYNTHIA is correct.

Proof: We highlight two observations from Algorithms 1 and 2 that show that SYNTHIA does not construct protocols that violate the SWMR invariant. First, SYNTHIA does not remove s-states listed in the input SYNTHIADSL specification or add new s-states during the protocol construction; SYNTHIA only adds new t-states and transitions with these t-states. Hence, in the protocol constructed by SYNTHIA, multiple cores that have copies of a cache line in s-states satisfy the SWMR invariant. Second, the construction of t-states (lines 5, 6, and 14 in Algorithm 1 and lines 9, 12, 15, 23, and 27 in Algorithm 2) are based on the transitions between s-states (*src*, *dst*, and *newDst*) defined in the input SYNTHIADSL specification. SYNTHIA modifies the original transitions between s-states to include the new t-states but does *not* change the source and destination s-states of the transition. Since the state encoding of t-states is either the source s-state for preordered t-states or destination s-states for post-ordered t-states, multiple cores that have the cache line in t-states also satisfy the SWMR invariant. Furthermore, putting these two observations together, multiple cores that have the cache line in t-states and s-states also satisfy the SWMR invariant. ■

2) Timing Predictability: The cache coherence protocols generated by SYNTHIA are deployed on the multicore model described in Section IV. This multicore model has the hardware structures described in [1] that work in tandem with the cache coherence protocol to guarantee timing predictability. Table II lists the design invariants for predictable cache coherence mechanisms proposed in [1] and shows that the combination of the multicore model and cache coherence protocols generated by SYNTHIA satisfy each design invariant. As a result, there does not exist a scenario under the SYNTHIA generated cache coherence protocols that has unbounded execution time, and therefore, they are timing predictable.

3) Necessary and Sufficient State Transitions: In this section, we argue that SYNTHIA generates necessary and sufficient state transitions that guarantee correctness and timing predictability.

TABLE II
IMPLEMENTATION OF DESIGN INVARIANTS [1] IN CACHE COHERENCE PROTOCOLS GENERATED BY SYNTHIA DEPLOYED ON MULTICORE MODEL

Invariant description	Implementation detail
Invariant 1: A predictable bus arbiter must manage coherence messages and data on the bus such that each core broadcasts a coherence request or communicates data on the bus if and only if it is granted an access slot to the bus	Satisfied by multi-core model – Predicable shared bus arbitration logic [1]
Invariant 2: The shared memory services requests to the same line in the order of their arrival to the shared memory	Satisfied by multi-core model – Pending request lookup table (PRLUT) in shared memory [1]
Invariant 3: A core responds to coherence requests in the order of their arrival to that core	Satisfied by multi-core model – Per-core pending response FIFO buffer such as a pending write-back (PWB) buffer [1]
Invariant 4: A write request from a core that is a hit to a non-modified line in core's private cache has to wait for the arbiter to grant the core an access to the bus	Satisfied by *generated protocol* – Accesses to cache lines with read-only access permissions must first broadcast a write request on the shared bus before completing the request (lines 3-9 in Algorithm 1)
Invariant 5: A write request from a core that is a hit to a non-modified line in the core's private cache has to wait until all waiting cores that previously requested the same cache line receive the cache line	Satisfied by *generated protocol* – A cache line in a transient state due to a write request transitions to other transient states on observing interleaving memory operations to the same cache line and does not complete its write operation until it receives the requested data (Algorithm 2)
Invariant 6: Each core has to deploy a predictable arbitration between its own generated requests and its responses to requests from other cores	Satisfied by *generated protocol and multi-core model* – Each core deploys a predictable arbitration applied between pending request [1] and response buffers and the protocol implementation marks cache lines with pending responses with a new transient state (Lines 10-16 Algorithm 1)

For the necessary argument, we argue that suppressing any one of the lines in SYNTHIA's algorithms, that is, responsible for constructing transitions will result in a protocol that violates correctness or timing predictability. Algorithm 1 constructs transitions due to shared bus communication. Suppressing lines 8 or 9 will violate timing predictability or correctness—a cache line in a read-only state (shared) can directly transition to a modified state without waiting to broadcast its request (violates Invariant 5 in Table II) or a cache line in an invalid state will transition to a valid state before receiving the requested data (violates correctness). Suppressing line 16 will violate timing predictability as a cache line will not transition to a transient state indicating pending actions—a core that performs some shared bus actions in response to another core's request must wait for its own time slot. Algorithm 2 constructs transitions due to interleaving memory operations. Suppressing lines 10, 13, 17, and 24 will violate correctness as a core will not record the final destination state for a cache line due to state changes caused by interleaving memory operations from other cores to the same cache line. Suppressing line 28 will violate timing predictability for the same reasons as described for line 16. Suppressing lines 30 and 32 will violate correctness as the transition between stable states will be incomplete. In summary, suppressing any lines in Algorithms 1 and 2 will violate either correctness or timing predictability.

For the sufficient argument, we argue that the set of all the necessary transitions constructed by SYNTHIA is sufficient to ensure correctness and timing predictability. Algorithm 1 is applied to all transitions specified in the input specification, which is assumed to be correct. Algorithm 2 iterates over all transitions in the input specification, and considers all possible combinations of transitions and interleaving memory operations from other cores. As a result, Algorithms 1 and 2 generate all necessary transitions and transient states for correctness and timing predictability, and this set of all necessary transitions and states is sufficient for correctness and timing predictability.

F. Limitations of SYNTHIA

There are three main limitations of SYNTHIA. First, SYNTHIA assumes that the input protocol specification is correct, and does not verify or validate the correctness of the input protocol specification. Second, the final nonstalling protocol implementation is described in SYNTHIADSL. To empirically evaluate the constructed nonstalling protocol, a designer must convert the protocol implementation in SYNTHIADSL into an alternate implementation for micro-architectural simulation, such as SLICC [9] or hardware logic implementation (Verilog or VHDL). Third, SYNTHIA generates snooping bus-based cache coherence protocols, and assumes all cores execute hard real-time tasks. As a result, SYNTHIA cannot generate predictable cache coherence protocols for mixed-critical systems deployed on multicore platforms, such as [2] and [3].

G. Future Extensions to SYNTHIA

We envision two extensions to SYNTHIA to handle multilevel cache hierarchies and directory cache coherence protocols.

Support for Multilevel Cache Hierarchies: Handling cache line replacements and invalidations based on the inclusion policy of the multilevel cache hierarchy is a challenge for automating the construction of coherence protocols for multilevel cache hierarchies. For example, consider a multilevel cache hierarchy with an inclusive LLC. An eviction of a cache line in the LLC must broadcast invalidation messages on the snooping bus to invalidate the cache line in the cores' private caches. This requires additional state transitions in the cache coherence protocol implemented at different cache levels. For future work, we will extend SYNTHIA to synthesize predictable cache coherence protocols for multilevel cache hierarchies using insights from [6] and [15].

Synthesize Directory Cache Coherence Protocols: Conventional directory cache coherence protocols have different transient states and additional trigger events for transitions (acknowledgments) compared to snooping bus-based cache coherence protocols [7]. To the best of our knowledge, we are not aware of works that explore the design of predictable directory cache coherence protocols. Hence, a key challenge with synthesizing predictable directory cache coherence protocols is first understanding the interactions between the interconnect design and cache coherence mechanism on timing predictability.

```
M : (write, dirty, active)
S : (read, clean, passive)
E : (exread, dirty, active)
F : (read, clean, active)
I : (invalid, clean, passive)
(I, OwnReadM)    -> E
(I, OwnRead)     -> F
(I, OwnWrite)    -> M
(I, OtherRead)   -> I
(I, OtherWrite)  -> I
(S, OwnRead)     -> S
(S, OwnWrite)    -> M
(S, OtherWrite)  -> I
(S, OtherRead)   -> S
(S, Replacement) -> I
(M, OwnRead)     -> M
(M, OwnWrite)    -> M
(M, OtherRead)   -> S
(M, OtherWrite)  -> I
(M, Replacement) -> I
(E, OwnRead)     -> E
(E, OwnWrite)    -> M
(E, OtherRead)   -> S
(E, OtherWrite)  -> I
(E, Replacement) -> I
(F, OwnRead)     -> F
(F, OwnWrite)    -> M
(F, OtherRead)   -> S
(F, OtherWrite)  -> I
(F, Replacement) -> I
```

Fig. 6. MESIF protocol specification in SYNTHIADSL.

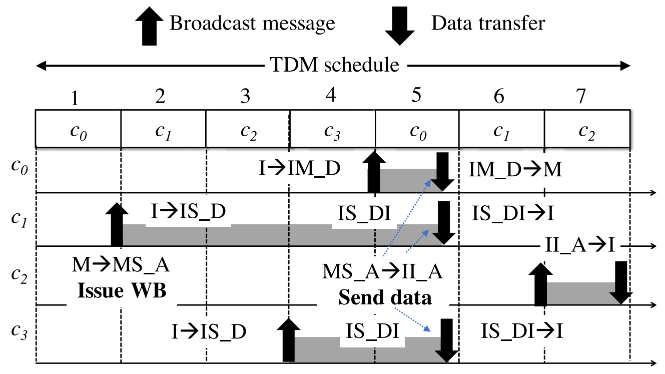

Fig. 7. Execution example under PMESIF protocol.

VI. Case Study: Predictable MESIF Protocol

Recently, Sensfelder *et al.* [10] reverse engineered the hardware cache coherence protocol deployed in the NXP QorIQ multicore processors. The NXP QorIQ multicore processors are positioned for use in safety-critical systems such as avionics [16]. Their reverse engineering efforts revealed that the NXP QorIQ multicore processor deployed a snooping bus-based MESIF cache coherence protocol. Their discovery is important as it shows that there exist multicore processors used for safety-critical real-time systems that implement hardware snooping bus-based cache coherence mechanisms for shared data communication between cores. To satisfy the timing constraints of the deployed safety-critical tasks, the snooping bus-based cache coherence mechanism must be predictable [1]. For the NXP QorIQ multicore processor, this means that the implemented MESIF cache coherence mechanism must be predictable. While we are not aware of all the implementation details of the MESIF cache coherence mechanism in the NXP QorIQ multicore processors and hence, their predictability guarantees, we design one predictable variant of the MESIF cache coherence mechanism using SYNTHIA. The constructed predictable and high-performance variant of the MESIF protocol, PMESIF, satisfies the design invariants listed in [1].

The MESIF protocol consists of 5 s-states. The **M**, **E**, **S**, and **I** states are the same as described in Section V-A. The forwarding state **F** allows a core with a read-only copy of the cache line to send the data to another core requesting for the same cache line. Hence, the state encoding of **F** is (*read*, *clean*, *active*). The SYNTHIADSL specification of the MESIF protocol in stable states is shown in Fig. 6.

We highlight one key feature about the MESIF protocol that makes it different from the MSI, MESI, and MOESI cache coherence protocols. Consider the transitions (I, OwnRead) → F and (F, OtherRead) → S. Notice that a core that has a cache line in **F** and observes an OtherRead not only sends the data to the requesting core but also changes the cache line coherence state to **S**; the core releases its *active* data authority on the cache line. As a result, the requesting core that receives data from another core (OwnRead), receives the cache line in **F** state. The benefit of transferring the **F** state of a cache line between requesting cores is that the likelihood of a cache line in **F** state to be a replacement candidate due to a cache capacity miss is lowered. A core that receives a cache line for its read request from another core will be the most recently used cache line in the cache set, and hence, will most likely not be a replacement candidate. Hence, the likelihood of cores receiving their requested cache line from other cores is high in the MESIF protocol. On the other hand, the MSI, MESI, and MOESI cache coherence protocols do not have a forwarding state, and do not transfer data authority to cores performing read requests on cache lines.

Table III shows the private cache coherence states of PMESIF cache coherence protocol generated by SYNTHIA. The cells in red indicate t-states and transitions that differ from the conventional MESIF cache coherence protocol. Table IV maps the PMESIF t-states and transitions constructed by SYNTHIA to Algorithms 1 and 2. As an example, consider the construction of (E, OtherRead) → ES_A. In the input MESIF protocol description in Fig. 6, a core that has a cache line in **E** transitions to **S** on observing OtherRead. The state encoding of **E** and **S** are (*exread*, *dirty*, *active*) and (*read*, *clean*, *passive*), respectively. The requesting core on receiving the data transitions from **I** to **F** state. The input to Algorithm 1 is (E, OtherRead) → S, and this transition exercises lines 14–20 since ev = OtherRead. The condition on line 14 returns true as **E** has *dirty* data state and *active* data authority. OWNTRANSITIONS(OtherRead) returns (I, OwnRead) → F. The condition on line 17 returns true as there is a cumulative change in data state from *dirty* to *clean* across all cores when the cache line in one takes the transition (E, OtherRead) → S and the cache line in the requesting core takes the transition (I, OwnRead) → F. Hence, SYNTHIA constructs the new t-state **ES_A**, and transitions (E, OtherRead) → ES_A and (ES_A, Ordered) → S.

We highlight two key features of the PMESIF protocol.

Feature 1: Unlike the conventional MESIF coherence protocol where a core with a cache line in **M/E** sends the requested data to *the first requesting core*, a core that has a cache line in **MS_A/ES_A** can send data to multiple requesting cores through the point-to-point data interconnects. This is because a core that has a cache line in **MS_A/ES_A** can observe multiple requests to the same cache line from different cores while waiting for its allocated slot to perform the write-back to shared memory. Fig. 7 shows an execution example on cache line A accessed by four cores (c_0-c_3) where c_2 has A in **M** state. For

TABLE III
PRIVATE MEMORY STATES FOR PMESIF CACHE COHERENCE PROTOCOL GENERATED BY SYNTHIA. *issue msg/state* MEANS THE CORE ISSUES THE MESSAGE *msg* AND MOVE TO STATE *state*. CHANGES TO CONVENTIONAL MESIF ARE IN BOLD RED

	Core events			Bus events			
	OwnRead	**OwnWrite**	**Replacement**	**Receive data (RD)**	**Ordered**	**OtherRead**	**OtherWrite**
I	Issue OwnReadM/OwnRead/IS_AD	Issue OwnWrite/IM_AD				—	—
S	Hit, Complete read	Issue OwnWrite/SM_AD	-/I			—	-/I
M	Hit, Complete read	Hit, Complete write	Issue write-back/MI_A			Issue write-back/MS_A	Send data/I
E	Hit, Complete read	Hit, Complete write/M	Issue write-back/EI_A			Issue write-back/ES_A	Send data/I
F	Hit, Complete read	Issue write/FM_AD	Issue coherence message/FI_A			Send data/S	Send data/I
IS_AD					/-IS_D	—	—
IM_AD					/-IM_D	—	—
IS_D				If data from memory Complete read/E, else Complete read/F		—	-/IS_DI
IM_D				Complete write/M		-/IM_DS	-/IM_DI
IM_DS				Complete write, issue write-back/MS_A		—	-/IM_DSI
IM_DI				Complete write, send data/I		—	—
IM_DSI				Complete write, send data/I		—	—
IS_DI				Complete read/I		—	—
SM_AD			Stall		/SM_D	—	-/IM_AD
SM_D			Stall	Complete write/M		-/SM_DS	-/SM_DI
SM_DI			Stall	Complete write, send data/I		—	—
SM_DS			Stall	Complete write, issue write-back/MS_A		—	-/SM_DSI
SM_DSI			Stall	Complete write, send data/I		—	—
FM_AD			Stall		/FM_D	Send data/SM_AD	-/IM_AD
FM_D			Stall	Complete write/M		-/FM_DS	-/FM_DI
FM_DI			Stall	Complete write, send data/I		—	—
FM_DS			Stall	Complete write, issue write-back/MS_A		—	-/FM_DSI
FM_DSI			Stall	Complete write, send data/I		—	—
MI_A	Hit, Complete read	Hit, Complete write	—		Write-back to memory/I	—	Send data/II_A
MS_A	Hit, Complete read	Hit, Complete write	-/MI_A		Write-back to memory, send data/S	—	Send data/II_A
EI_A	Hit, Complete read	Hit, Complete write/MI_A	—		Write-back to memory/I	—	Send data/II_A
ES_A	Hit, Complete read	Hit, Complete write/MS_A	-/EI_A		Write-back to memory, send data/S	—	Send data/II_A
FI_A	Hit, Complete read	Stall	—		-/I	—	Send data/II_A
II_A	Stall	Stall	—		-/I	—	—

TABLE IV
PMESIF T-STATES AND TRANSITIONS CONSTRUCTED BY SYNTHIA

Algorithm lines	t-states and transitions constructed
Lines 4-9 Algorithm 1	(I, OwnRead/OwnReadM) → IS_AD, (I, OwnRead/OwnReadM) → IM_AD, (IS_AD, Ordered) → IS_D, (IM_AD, Ordered) → IM_D, (IS_D, RD) → E/F, (IM_D, RD) → M, (S, OwnWrite) → SM_AD, (SM_AD, Ordered) → SM_D, (SM_AD, RD) → M, (F, OwnWrite) → FM_AD, (FM_AD, Ordered) → FM_D, (FM_D, RD) → M
Lines 10-16 Algorithm 1	(M, OtherRead) → MS_A, (E, OtherRead) → ES_A
Lines 8-10 Algorithm 2	(SM_AD, OtherWrite) → IM_AD, (FM_AD, OtherWrite) → IM_AD
Lines 11-13 Algorithm 2	(FM_AD, OtherRead) → SM_AD
Lines 15-17 Algorithm 2	(MS_A, OtherWrite) → II_A, (ES_A, OtherWR) → II_A, (MI_A, OtherWrite) → II_A, (EI_A, OtherWrite) → II_A
Line 19 Algorithm 2	(SM_AD, OtherRead) → SM_AD, (IM_AD, OtherRead) → IM_AD, (IS_AD, OtherRead) → IS_AD, (IM_AD, OtherWrite) → IM_AD, (IS_AD, OtherWrite) → IS_AD, (ES_A, OtherRead) → ES_A, (MS_A, OtherRead) → MS_A
Lines 23-25 Algorithm 2	(IM_D, OtherRead) → IM_DS, (IM_DS, OtherWrite) → IM_DSI, (IM_D, OtherWrite) → IM_DI, (IS_D, OtherWrite) → IS_DI, (SM_D, OtherRead) → SM_DS, (SM_D, OtherWrite) → SM_DI, (SM_DS, OtherWrite) → SM_DSI, (FM_D, OtherRead) → SM_D, (FM_D, OtherWrite) → FM_DI, (FM_D, OtherRead) → FM_DS, (FM_DS, OtherWrite) → FM_DSI
Lines 26-28 Algorithm 2	(IM_DS, RD) → MS_A, (SM_DS, RD) → MS_A, (FM_DS, RD) → MS_A
Line 30 Algorithm 2	(IS_DI, RD) → I, (IM_DI, RD) → I, (IM_DSI, RD) → I, (SM_DI, RD) → I, (SM_DSI, RD) → I, (FM_DI, RD) → I, (FM_DSI, RD) → I
Lines 31-32 Algorithm 2	(IS_D, OtherRead) → IS_D, (IS_DI, OtherRead) → IS_DI, (IS_DI, OtherWrite) → IS_DI, (IM_DS, OtherRead) → IM_DS, (IM_DI, OtherRead) → IM_DI, (IM_DI, OtherWrite) → IM_DI, (IM_DSI, OtherRead) → IM_DSI, (IM_DSI, OtherWrite) → IM_DSI, (SM_DS, OtherRead) → SM_DS, (FM_DS, OtherRead) → FM_DS, (SM_DSI, OtherRead) → SM_DSI, (SM_DSI, OtherWrite) → SM_DSI, (SM_DI, OtherRead) → SM_DI, (SM_DI, OtherWrite) → SM_DI, (FM_DSI, OtherRead) → FM_DSI, (FM_DSI, OtherWrite) → FM_DSI, (FM_DI, OtherRead) → FM_DI, (FM_DI, OtherWrite) → FM_DI

this execution example, we assume the shared buses deploy a TDM predictable bus arbitration, and each core is allocated one-time slot. In slot 2, c_1 broadcasts a read request on A. This causes c_2 to issue a write-back coherence message and change the coherence state of its cache line copy of A to **MS_A**. For this example, assume that c_2 completes the data write-back in its allocated slot (slot 7). In slot 4, c_3 also broadcasts a read request on A. c_2 observes c_3's read request and does not change the coherence state of its A copy. c_0's write request to A causes A in c_2 to transition to **II_A**. c_2 sends the cache line data contents of A to the requesting cores c_0, c_1, and c_3. This is shown in Fig. 7 where c_2 sends A to all the requesting cores in slot 5. Since each core is connected to every other core through point-to-point data interconnects, all the cores receive A in slot 5 and complete their pending memory requests on A.

Feature 2: In the conventional MESIF coherence protocol, a core that has a cache line in **IS_DI** completes the pending read request on receiving the data, and *sends* the data to another requesting core that has a pending write operation on the same cache line [10]. Recall that under the MESIF coherence protocol, a core that has a pending read request to a cache line receives the cache line data contents with *active* data authority (**F** state). For example, consider the execution example in Fig. 7. Under the conventional MESIF coherence protocol, c_1 sends the data to c_0 on receiving A. On the other hand, under PMESIF coherence protocol, a core that has a cache line in **IS_DI** does not send the data to another core on receiving the requested data. This is because under PMESIF coherence protocol, a core that does not complete its read request to a cache line in its allocated slot means that there exists another core that has the same cache line in either **MS_A** or **ES_A** state and is waiting for its allocated slot to perform a write-back of the requested cache line to shared memory. In the execution example in Fig. 7, cores c_1 and c_3 do not complete their read requests to A in their allocated time slots (slots 2 and 4, respectively) because c_2 has to complete the write-back of A to shared memory. **MS_A** and **ES_A** have the same state encoding as **M** and **E** states, respectively, and hence, have

TABLE V
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	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	20	26	96	✓	✓	18 of 51	6 of 57	12 of 72
MESI-P	4	20	26	95	✓	✓	22 of 57	6 of 57	16 of 78
MOESI	5	25	27	103	✓	✓	18 of 57	6 of 60	12 of 78
MESIF	5	25	35	118	✓	✓	22 of 63	6 of 66	16 of 84

active data authority. On observing an OtherWrite, the core with cache line in **MS_A/ES_A** will respond with data to the requesting cores. As a result, the core with cache line in **IS_DI** does not send data responses under PMESIF coherence protocol.

VII. RESULTS

Evaluation of SYNTHIA*:* SYNTHIA successfully constructs nonstalling and predictable coherence protocols from s-states specifications of MSI, MESI, MOESI, and MESIF protocols. The MESI and MOESI protocol specifications are derived from [7] and [17], respectively. Table V shows the number of states and transitions in the input and output. The MSI-P and MESI-P protocols differ from the MSI and MESI protocols in that *all* states 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 described in [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. 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].

SYNTHIA is designed to improve the productivity of protocol designers by automating the exhaustive analysis required to construct predictable and high-performance protocols. To highlight this key utility of SYNTHIA, we perform the following experiment. Suppose a protocol designer manually designed the protocols listed in Table V, 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 V). 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 preordered), 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. In summary, a protocol designer can construct protocols where more than a third of all the transitions are stalling transitions by missing some scenarios. SYNTHIA addresses this by automating the analysis and construction as described in Section V.

TABLE VI
PREDICTABILITY AND PERFORMANCE EVALUATION

Protocol	Predictability (WCL in cycles)		Performance
	Observed	Analytical	Synthetic
MSI	3061	6450	3.45×
MESI	3214	6450	3.35×
MOESI	2019	6450	3.99×
MSI-P	6364	7250	1.72×
MESI-P	5964	7250	1.69×
MESIF	3741	7250	3.24×

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]. The conversion was straightforward as it involved describing the states, transitions, and actions in the output protocol in SLICC syntax. We used synthetic workloads from [1] and verified the data correctness of the protocols. We modeled an 8-core multicore platform where the shared bus deploys a TDM arbitration. Each core is allocated one TDM slot. The analytical WCL bounds of a memory request in Table VI are computed using the timing analyses described in [1] and [18]. Table VI shows the maximum observed memory latency 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 achieves predictable data sharing between cores by disabling private caching of shared data. From Table VI, the observed maximum latency 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.

Among the predictable cache coherence protocols, the PMOESI cache coherence protocol offers the best performance speedup for synthetic workloads (3.99× average speedup over cache bypassing and 15% over PMSI, which is the next best cache coherence protocol). The key reason for this is that the PMOESI cache coherence protocol has the least number of transitions that require communication with the shared memory. In the PMOESI protocol, a core that has a cache line in **E** state must write-back the data contents to the shared memory on an OtherRead event. A core that has a cache line in **M** and observes an OtherRead event does not trigger a write-back to shared memory. In this scenario, the core responds with the requested data and transitions the cache line from $\mathbf{M} = (\mathit{write}, \mathit{dirty}, \mathit{active})$ to $\mathbf{O} = (\mathit{read}, \mathit{dirty}, \mathit{active})$; ISCUMULATIVECHANGE() in Algorithm 1 returns false. In the PMESI and PMESIF protocols, a core that has a cache line in **E** or **M** states must write-back the data contents to the shared memory on an OtherRead event. As a result, the PMESI and PMESIF protocols have lower performance benefits compared to the PMOESI protocol, yet still maintain over 3× average performance improvement over cache bypassing.

Verification of Cache Coherence Protocols: We use the framework by Sensfelder *et al.* [19] to formally verify the correctness, liveness, and safety properties of the protocols generated by SYNTHIA. This framework uses the UPPAAL modeling framework. We changed the models in [19] to reflect our system model assumptions (Section IV), and verified correctness (SWMR invariant, data value invariant), liveness (program termination), and safety (latency of each memory request is within the WCL bound) properties of the PMSI, PMSI-P, PMESI, PMESI-P, PMOESI, and PMESIF protocols.

VIII. CONCLUSION

We present SYNTHIA, an automated tool for constructing correct, predictable, and high-performance cache coherence protocols from simple protocol specifications. SYNTHIA automates the analyses of scenarios involving interleaving memory operations from multiple cores to shared data and 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 apply SYNTHIA on multiple coherence protocol implementations found in existing multicore platforms, such as the MSI, MESI, MOESI, and MESIF cache coherence protocols. 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] A. M. Kaushik, M. Hassan, and H. Patel, "Designing predictable cache coherence protocols for multi-core real-time systems," *IEEE Trans. Comput.*, early access, Nov. 16, 2020, doi: 10.1109/TC.2020.3037747.

[2] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "CARP: A data communication mechanism for multi-core mixed-criticality systems," in *Proc. RTSS*, 2019, pp. 419–432.

[3] N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *Proc. RTSS*, 2019, pp. 433–445.

[4] M. Hassan, "Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems," in *Proc. Euromicro Conf. Real-Time Syst. (ECRTS)*, Jul. 2020, pp. 1–22.

[5] S. Hessien and M. Hassan, "The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Houston, TX, USA, Oct. 2020, pp. 1–12.

[6] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications," in *Proc. ISCA*, Los Angeles, CA, USA, 2018, pp. 247–260.

[7] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," in *Synthesis Lectures on Computer Architecture*. San Rafael, CA, USA: Morgan Claypool, 2011.

[8] A. M. Kaushik and H. Patel, "Automated synthesis of predictable and high-performance cache coherence protocols," in *Proc. Design Autom. Test Eur. Conf. (DATE)*, Grenoble, France, 2021, pp. 816–821.

[9] N. Binkert *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, May 2011.

[10] N. Sensfelder, J. Brunel, and C. Pagetti, "On how to identify cache coherence: Case of the NXP QorIQ T4240," in *Proc. ECRTS*, 2020, pp. 1–22.

[11] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[12] *Arm® Cortex®-R8 MPCore Processor*, ARM, Cambridge, U.K., 2019.

[13] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: Specifying protocols with concolic snippets," in *Proc. PLDI*, 2013, pp. 287–296.

[14] M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan, "VerC3: A library for explicit state synthesis of concurrent systems," in *Proc. DATE*, Dresden, Germany, 2018, pp. 1381–1386.

[15] N. Oswald, V. Nagarajan, and D. J. Sorin, "HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, Valencia, Spain, 2020, pp. 888–899.

[16] "QorIQ® T4240, T4160 and T4080 Multicore Processors." NXP. 2016. [Online]. Available: https://www.nxp.com/docs/en/fact-sheet/T4240T4160FS.pdf

[17] *AMD64 Architecture Programmer's Manual Volume 2: System Programming 24593 Programmer's Manual 2*, Adv. Micro Devices, Santa Clara, CA, USA, 2006.

[18] A. M. Kaushik and H. Patel, "A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Nashville, TN, USA, 2021, pp. 105–117.

[19] N. Sensfelder, J. Brunel, and C. Pagetti, "Modeling cache coherence to expose interference," in *Proc. ECRTS*, 2019, pp. 1–22.

Anirudh Mohan Kaushik (Member, IEEE) is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, University of Waterloo, Waterloo, ON, Canada.

His research interests are in real-time embedded systems architecture and high performance computer architecture.

Hiren Patel (Member, IEEE) received the Ph.D. degree from Virginia Tech, Blacksburg, VA, USA.

He was a Postdoctoral Fellow with the University of California, Berkeley, CA, USA. He is a Professor with the Electrical and Computer Engineering Department, University of Waterloo, Waterloo, ON, Canada. His current research interests include real-time embedded systems, computer architecture, hardware architectures for machine learning and artificial intelligence, and security.