Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores

Zhuanhao Wu, Anirudh Mohan Kaushik, Paulos Tegegn and Hiren Patel
Electrical and Computer Engineering Department
University of Waterloo, Canada
{z284wu, anirudh.m.kaushik, ptegegn, hiren.patel}@uwaterloo.ca

***Abstract*—This work presents MapleBoard: a set of open-source hardware tools to implement predictable cache coherence protocols in hardware. MapleBoard consists of the following: (1) a novel domain-specific language (DSL) for specifying coherence protocols and synthesizing the corresponding hardware; and, (2) a real-time multicore hardware platform that seamlessly integrates the coherence protocols synthesized from the DSL. This platform has a memory hierarchy, a real-time bus interconnect between cores, and various predictable arbiters. As a demonstration of MapleBoard's efficacy, we explore hardware implementations of data bus organizations, and their impact on the worst-case communication latency (WCL). An important discovery we make is that a dedicated data bus (DDB) organization that allows bidirectional data communication offers lower analytical WCL bounds than any state-of-the-art predictable cache coherence protocols. The analytical WCL bounds are improved by 84%, 90% and 94% for 2-core, 4-core and 8-core systems respectively compared to prior works. We synthesize MapleBoard on the Xilinx Virtex Ultrascale+ VCU1525 board, and validate using both synthetic workloads and SPLASH-2 benchmark suite.**

I. Introduction

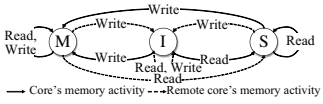
Safety-critical systems applications in avionics and automotive domains are deployed on multicore platforms to reduce cost and improve average-case performance [1]–[4]. Recent evidence of data communication in real-world safety-critical systems applications [5] has prompted research efforts in *predictable data communication* between cores on real-time multicore platforms [6]–[11]. One recent approach employs *hardware cache coherence*, which is the primary mechanism for data communication in conventional multicore platforms [6]–[10]. A hardware cache coherence mechanism enables coherent data communication between cores and allows multiple cores to simultaneously cache the same data in their private caches. The main component of a hardware cache coherence mechanism is the cache coherence protocol, which is a set of rules that dictate the communication of data between cores. Prior research efforts on predictable cache coherence mechanisms modified conventional cache coherence mechanisms such that they are amenable to timing analysis and therefore, allow one to statically compute worst-case latency (WCL) bounds of a memory request under cache coherence [6]–[9]. The resultant mechanisms retained most of the performance benefits of conventional cache coherence mechanisms, and outperformed alternate predictable data communication approaches such as cache bypassing and task mapping [6]–[9]. Although we are unaware of multi-core platforms that implement predictable cache coherence mechanisms, we expect them to be an attractive alternative to existing data communication mechanisms given their predictability and performance guarantees. Thus, we anticipate further research into predictable cache coherence mechanisms, and we are hopeful of its industry adoption.

Prior works in designing predictable cache coherence mechanisms [6]–[8] relied on micro-architectural simulators to prototype and evaluate the mechanisms [12]. Simulation prototypes are invaluable for rapid design space exploration and validation. However, we contend that concrete hardware implementations of predictable data communication mechanisms have additional merits. (1) Certification processes for safety-critical systems require detailed knowledge of the underlying hardware platform to ensure that applications running on the hardware never violate their WCL. Releasing the hardware implementation to the public domain enables everyone to ascertain this detailed knowledge. (2) The abstraction used in simulation prototypes may not accurately capture certain properties and optimizations of the hardware implementation. Hence, solely relying on the simulation prototype may result in deriving inaccurate or loose WCL bounds. (3) Finally, we believe that a concrete hardware implementation of prior and future hardware design research works on predictability including predictable cache coherence mechanisms facilitates hardware vendors and start-ups to adopt and integrate these implementations into their products.

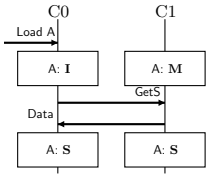
Our main contribution is MapleBoard[1], which consists of a set of hardware tools to facilitate the research community to implement and evaluate predictable data communication mechanisms in hardware. MapleBoard consists of the following: (1) MapleDSL, a domain specific language (DSL) that allows designers to rapidly describe and implement predictable cache coherence protocols (protocol states and transitions between states); and, (2) a real-time multicore platform with configurable components such as the cache hierarchy, and real-time bus interconnect arbitration policies. To show the efficacy of MapleBoard, we present a case study that explores the design space of data bus organization to achieve lower WCL bounds of memory requests under predictable cache coherence. In this case study, our proposed dedicated data bus (DDB) organization lowers the WCL of memory requests

[1]MapleBoard available at https://github.com/caesr-uwaterloo/MapleBoard

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.



(a) MSI cache coherence protocol



(b) An MSI example of two core system

Fig. 1: Examples show the state transitions and an execution of MSI.

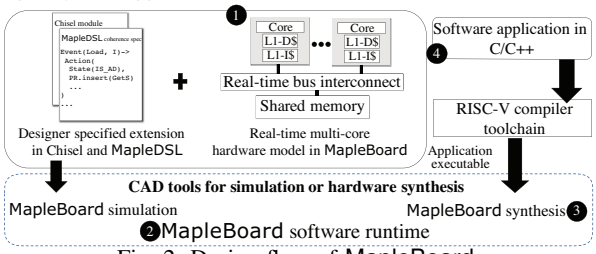


Fig. 2: Design flow of MapleBoard.

under predictable cache coherence compared to previous works [6], [7] by approximately 84%, 90% and 94% for 2-core, 4-core and 8-core configurations, respectively. We adopt this organization in MapleBoard to empirically evaluate its effectiveness and show it can be implemented in hardware; consequently, a first hardware implementation of predictable cache coherence on a real-time multicore platform on the Xilinx Virtex Ultrascale+ VCU1525. Our empirical evaluation shows that the observed WCL are reduced by more than 71%, 84% and 93% for 2-core, 4-core and 8-core configurations respectively for PMSI and PMESI. Our evaluation also validates that the observed WCL of memory requests under prior proposed predictable cache coherence protocols are within the analytical WCL bounds derived in prior works.

II. Background

A. Chisel

Chisel [13] is a hardware-description language (HDL) with hardware constructs embedded in Scala with features such as object-orientation, functional programming and meta-programming. In Chisel work-flow, the compiler converts the Chisel hardware model into an intermediate FIRRTL model [14] and then into a Verilog design that traditional CAD tools for FPGAs or ASICs can synthesize.

B. Hardware cache coherence

Hardware cache coherence is a data communication mechanism prevalent in multicore platforms that enables cores to simultaneously cache and access data in their private caches [15], [16]. Cache coherence protocol is the key component in a hardware cache coherence mechanism and consists of rules to ensure the *single-writer-multiple-reader invariant* and the *data-value invariant* so that all cores observe the same value of a given datum at the same (logical) time.

The key component in a hardware cache coherence mechanism is the cache coherence protocol, which defines a set of rules that maintain two variants These rules are typically defined at the cache line granularity. A cache coherence protocol consists of two components: (1) **coherence states** that capture access permissions (read/write) of data, and (2) **state transitions** between coherence states that are triggered based on the memory activity of cores on the same data. Each core's cache controller implements the coherence protocol. MapleBoard implements snooping bus-based cache coherence protocols [16], which we find appropriate based on the core count in current real-time multicore systems [17]–[19]. Recent research on predictable cache coherence protocols such as [6], [7] also assume snooping bus.

Figure 1a shows the Modified-Shared-Invalid (MSI) cache coherence protocol. Coherence protocols deployed in commercial multicore platforms build on top of the MSI cache coherence protocol [20], [21]. The MSI cache coherence protocol consists of three coherence states: **Modified (M)**, **Shared (S)**, and **Invalid (I)** [16]. A cache line in **M** state means that it has read and write access permissions, and its data contents have been modified. Only one core can have a cache line in **M** state at any time instance for data correctness [16]. A cache line in **S** state means that it has read access permissions, and its data contents have not been modified. Cores that have a cache line in **S** are referred to as *sharers*. A cache line in **I** state means that it is not available in the core's cache or its data contents are not up-to-date. State transitions between the coherence states are triggered on memory activity by cores on the same cache line.

For example, consider a two-core system with cores C0 and C1 in Figure 1b. C0 issues a read request to cache line A (Load A) that C1 has in the **M** state. Based on the state transitions in MSI protocol (Figure 1a), C1 changes its coherence state to **S** on observing C0's read request (GetS), and C0 moves from **I** to **S** state on receiving data for A; thereby allowing both cores to subsequently read A from their respective private caches.

Extending the MSI cache coherence protocol with transient states enables the deployment of the protocol on a non-atomic snooping bus, where memory requests from multiple cores interleave and the performance improves. For example, in MSI, when a reading core is waiting for a cache line to change from **I** to **S**, a non-atomic snooping bus allows for another core to request the same cache line in **M** before the cache line arrives the reading core. As a result, the final state of the reading core will be **I** instead of **S**. Adding transient states between **I** and **S** allows the reading core to account for such changes in the final coherence state [16].

C. Predictable hardware cache coherence

Prior works on predictable snooping bus-based hardware cache coherence protocols [6], [7], [22] used design guidelines and micro-architectural extensions put forth in [6]. In these works, the snooping bus interconnect was shared between cores and the shared memory, and this bus communicated coherence messages and data between cores and shared memory. The shared snooping bus interconnect deployed a predictable

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

arbitration policy to manage simultaneous bus interconnect accesses by the cores [6], [7]. There were three main architectural extensions proposed in these prior works: (1) pending request lookup table (PRLUT) at the shared memory, (2) per-core pending request (PR) buffer, and (3) per-core pending write-back (PWB) buffer [6], [7].

The PRLUT records pending requests to cache lines at the shared memory, and ensures that multiple pending requests to the same cache line are serviced by the shared memory in the broadcast order. Let core C1 and core C2 request cache line A which core C0 holds in modified state. If PRLUT is not present, the shared memory can always service C0 and C1, starving C2 and making the latency of memory requests for C2 unbounded. A core's PWB buffer records pending write-back responses due to memory activity from other cores. Let core C1 request cache line A, and core C2 request cache line B. Both A and B reside in C0's cache in modified state. If PWB is not present, core C0 can choose to write back A and B in any order. If core C0 chooses to write back B for core C2, additional requests from core C2 can prevent core C0 from writing back A, rendering the latency of memory requests for core C1 unbounded. A core's PR buffer records the core's pending requests that are ready to be broadcast on the bus. Let core C1 requests a cache line A which core C0 holds in modified state. If PR and PWB is not present, core C0 can prioritize its own requests and does not write back cache line A, starving C1 and making the latency of memory requests for C1 unbounded. The PR and PWB buffers were implemented within the core's cache controller. If a core includes both instruction and data cache, each will be equipped with a PR and a PWB. Since a core's requests and write-back responses contend for bus accesses, an additional predictable arbitration was applied in each core's cache controller for servicing requests and responses in the PR and PWB buffers [6]. The real-time multicore architecture in MapleBoard incorporates these micro-architectural extensions, and predictable cache coherence protocols deployed on MapleBoard can make use of these extensions.

III. MapleBoard architecture

We describe four salient features of MapleBoard. These features enable designers to rapidly prototype and comprehensively evaluate predictable cache coherence protocols in hardware. Additionally, MapleBoard encourages exploration of other predictable micro-architectural extensions in hardware.

Feature 1. A new domain specific language (DSL), MapleDSL, that allows designers to rapidly describe and implement predictable and conventional cache coherence protocols in hardware. MapleDSL simplifies the specification, synthesis, and validation of coherence protocols. Section IV describes the DSL.

Feature 2. MapleBoard leverages the Chisel HDL [13], which inherits the benefits of the Chisel ecosystem such as chisel-testers2 [23], an open-source testing framework, and CAD tools that enable hardware logic synthesis. CAD synthesis tools can then be used to target FPGAs or ASICs.

Feature 3. A *configurable* and *synthesizable* real-time RISC-V multicore architecture with a memory hierarchy and real-time bus interconnect. Tools in MapleBoard seamlessly integrate the predictable cache coherence protocols with the real-time multicore architecture. Designers can synthesize MapleBoard into hardware logic, and evaluate key properties such as area, power, and operating frequency. Section III-A describes the multicore model in MapleBoard.

Feature 4. A software runtime component that allows designers to execute single-threaded and multi-threaded applications on the MapleBoard. Designers can comprehensively *evaluate* and *validate* runtime properties such as performance and observed WCL bounds. The software runtime component emulates key Linux operating system (OS) calls. Section III-B describes MapleBoard's software runtime component.

Figure 2 shows the design flow of MapleBoard. A designer describes micro-architectural extensions in Chisel, and integrates them with the multicore architecture provided by MapleBoard (❶). As an example, MapleDSL is a Chisel-based tool in MapleBoard to specify predictable cache coherence protocols. Designers use tools provided by the Chisel framework to generate a Verilog design that can be simulated through Verilog simulators for design verification (❷). Note that MapleBoard can be extended to use FireSim [24], a FPGA-accelerated hardware simulation framework. Designers use FPGA synthesis tools to generate logic for the extended multicore architecture (❸). We only target FPGAs in this work, but ASICs can also be targeted. We successfully synthesized MapleBoard on the reconfigurable fabric in the Xilinx Virtex Ultrascale+ VCU1525 board. MapleBoard uses the RISC-V compiler toolchain [25] to convert software applications written in C/C++ to a static application binary executable (❹). The software runtime component in MapleBoard enables application execution on the hardware.

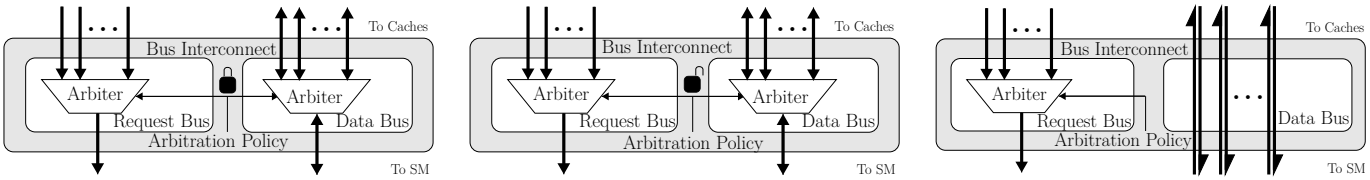
A. Hardware components of MapleBoard

MapleBoard has four main hardware components.

Core architecture. MapleBoard deploys RISC-V cores that implement the RV64IA instruction-set architecture. Each RISC-V core implements an in-order pipeline with support for memory, arithmetic, and atomic instructions. In-order cores are chosen as they are predictable over dynamically scheduled cores [26]. However, other predictable core architectures such as PRET [27] or SIC [26], could trivially be dropped in as replacements. The provided RISC-V core also includes a criticality register (CR) that records the criticality level of the application executing on the core. Each RISC-V core has a split level one (L1) instruction and data cache. The number of cores deployed in MapleBoard is configurable.

Memory hierarchy. MapleBoard has a two-level memory hierarchy with a private cache level per core (L1) and the shared main-memory. When synthesizing MapleBoard on FPGAs, the shared main-memory is the on-board memory module; otherwise, for simulation purposes, the shared main-memory is a component with constant latency. Caches are byte-addressable, and operate at 64-byte cache line granularity

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.



(a) Atomic configuration (b) Shared configuration (c) Dedicated configuration

Fig. 3: MapleBoard provides three bus interconnect configurations.

by default. Designers can configure the cache capacity, cache line size, and associativity.

Each cache has a cache controller that manages the contents of the cache's tag and data arrays, and handles a core's memory request to the cache. The shared memory controller (SMC) manages data contents in the shared memory, and handles data movement between the shared main-memory and cores' caches. When the shared main-memory is DRAM, the DRAM controller manages data movement between DRAM memory modules and the cores' caches. MapleBoard does not include a DRAM controller component, and relies on the availability of a DRAM controller model (hard or soft cores) on the target FPGA board.

Bus interconnect architecture. Cores communicate with each other and the shared memory via a bus interconnect. The bus interconnect communicates memory requests and its corresponding coherence messages, and data responses between the cores and shared memory. Currently, MapleBoard supports three interconnect configurations as shown in Figure 3: (1) atomic, (2) shared, and (3) dedicated. In all configurations, the interconnect has a request bus which communicates memory and coherence messages that is implemented as a shared snooping bus interconnect. Multiple accesses to the shared request bus are managed by a predictable bus arbitration policy. MapleBoard provides the following predictable bus arbitration policies: (1) round-robin (RR), (2) time division multiplexing (TDM), (3) weighted TDM, and (4) weighted TDM augmented with RR [28]. Designers can add new bus arbitration policies in MapleBoard based on their requirements.

The atomic and shared bus share the same representation in Figure 3b where both are equipped with a shared data bus. The difference between atomic and shared bus is that atomic bus processes memory transactions *atomically*: other caches' requests are blocked until the current request is completed, but shared bus allows *non-atomic* transactions on the bus where requests can interleave and are serviced when other cores stall for data response.

The key difference between the shared and dedicated configurations for the bus interconnects is in organization of the data bus. As shown in Figure 3b, in the shared configuration, cores and the shared memory communicate data over a shared bus interconnect. Concurrent accesses to the data bus interconnect are managed by a predictable bus arbitration policy similar to the request bus. Note that for this configuration, the arbitration schedule for the data bus is the same as that of the request bus. For the dedicated configuration shown in Figure 3c *each private cache has two separate data buses: one data bus to receive data from shared memory, and another data bus to send data to the shared memory*. The dedicated configuration has no need for an arbitration policy for data.

Cache coherence mechanism. The cores' cache controllers implement the cache coherence protocol that is responsible for maintaining a coherent view of data. MapleBoard can deploy both conventional cache coherence protocol such as the MSI or MESI protocols [16] or predictable cache coherence protocols such as PMSI [6] and CARP [7]. For deploying predictable cache coherence protocols, MapleBoard provides the necessary hardware structures discussed in [6] that were reviewed in section II-C such as the per-core PR and PWB buffers, and the PRLUT at the shared memory. Section IV describes MapleDSL, a DSL for designing cache coherence protocols in MapleBoard. Currently, we provide a choice of the following cache coherence protocols in MapleBoard: (1) MSI [16], (2) MESI [16], (3) PMSI [6], (4) PMESI [22], and (5) CARP [7].

Differences from prior works Compared to prior predictable coherence protocol efforts such as PMSI and CARP [6], [7], which are evaluated using micro-architecture simulators, this work differs in two ways. First, while prior works allocated at least one time slot to each core and neglected instruction cache accesses in their designs over the interconnects, this work allocates at least two slots to each core: one slot for the read-only instruction cache and one slot for the read-write data cache. Second, to allow for the Linux host supporting system calls to perform predictable memory accesses, we allocate one data cache to the host and allocate one slot for the data cache in the arbitration schedule.

Prior works [6], [7] overlooked an issue of sharing a slot between instruction and data cache. The issue is that there are interferences between instruction fetching and data access from the same core and such interferences are out of the scope of the analyses in these works. Note that such interferences do *not* invalidate the timing analyses in prior works, but requires extra investigation when integrating the cache with a predictable pipelined processor such as SIC [26] whose analysis takes as input the end-to-end latency of cache accesses. Moreover, sharing a slot between instruction and data cache requires extra logic to differentiate bus data responses for instruction and data. A more pragmatic hardware approach is to have separate slots for instruction cache and data cache, which eliminates such implicit interference and results in simpler hardware implementation. Our extended WCL analyses

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

take into account such changes to prior works.

B. Software component of MapleBoard

The software runtime library consists of implementations of system calls used by applications. Examples include those used in threading libraries to create and manage multiple thread contexts, file management operations to open and close files, and memory management operations to allocate memory regions. When the application execution encounters a system call, MapleBoard traps the system call, and emulates it using the implementation in the software runtime library. Designers can add software implementations of system calls into MapleBoard's software runtime library to enable execution of wider variety of applications. Currently, MapleBoard emulates system calls used in the `pthread` library, and memory management system calls. For other system calls, MapleBoard relies on a compute system that runs a Linux OS, which we refer to as the host, to execute system calls on behalf of MapleBoard. For the Xilinx Virtex Ultrascale+ VCU1525, we designed a custom communication interface between MapleBoard and the compute host using the shared memory. The MapleBoard services memory requests of host by the same predictable cache as other cores to ensure predictability per memory request and cache coherency among the host and other cores. In MapleBoard, the system calls delegated to the host mostly set up the application data before the execution of the program and has minimal effects on the execution of the programs. In an in-field deployment, one of the cores can play the role of the host with proper runtime libraries that provide full support for time management, I/O and drivers for other components such as a screen. These functionalities are not in the scope of this work.

IV. MapleDSL: A DSL for Describing Cache Coherence Protocols

MapleDSL is a novel domain specific language (DSL) for specifying cache coherence protocols for MapleBoard. Designers can specify *both* predictable cache coherence protocols and conventional high-performance cache coherence protocols with MapleDSL. While we show the feasibility of MapleDSL describing conventional cache coherence protocols in Section VI, our focus in this work is on predictable cache coherence protocols.

A. MapleDSL compiler flow

Figure 4 shows the MapleDSL compiler flow. The MapleDSL compiler takes as input the protocol specification written in MapleDSL, and outputs the hardware implementation of the protocol in Chisel. The input protocol specification is a set of Scala files that specify protocol states and transitions between states. Section IV-C describes the format for specifying protocol state transitions in MapleDSL. For each protocol state transition, the MapleDSL compiler generates one Chisel *when/elsewhen* branch which is equivalent to an if/else-if branch in Verilog. The branch condition checks for the corresponding initial coherence state and an observed

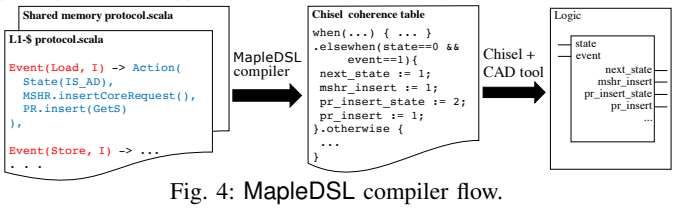


Fig. 4: MapleDSL compiler flow.

event. Statements in the branch body drive control and data signals to inform the hardware of actions to perform. The transformed transitions form a combinational Chisel coherence table module, which is synthesized to hardware using CAD tools.

B. MapleDSL features

Taking inspiration from SLICC, a coherence protocol DSL in gem5 simulator [12], MapleDSL has three key features that fill the gaps between SLICC DSL and real-time systems' requirements.

Criticality level of memory requests. MapleDSL allows designers to specify state transitions for a cache line based on the *relative* criticality level of remote memory requests to the same cache line. We define relative criticality level as the difference in criticality levels of a core observing a remote memory request and the remote core that broadcasted the memory request. This feature allows specifying predictable cache coherence protocols for *mixed criticality systems* (MCS), which are common in avionics and automotive domains [29]. Applications running on MCS have different criticality levels, which in turn have different temporal and certification requirements [29]. There are three possible relative criticality levels: (1) LOCRIT: the criticality level of the remote core's memory request is *lower* than that of the core observing the memory request on the bus, (2) HICRIT: the criticality level of the remote core's memory request is *higher* than that of the core observing the memory request on the bus, and (3) SAMECRIT: the criticality levels of the core observing the remote memory request and the remote core that broadcasted the memory request are the *same*. Since MapleBoard and MapleDSL are pure Scala, MapleBoard and MapleDSL can be extended to support other variants of criticality level specifications such as ASIL in ISO-26262 and safety-critical levels in DO178. On the other hand, the SLICC DSL allows designers to specify state transitions from a coherence state based on only the remote memory request type. As a result, predictable cache coherence protocols for MCS such as CARP [7] can be specified in MapleDSL, whereas specifying the same in SLICC DSL is a challenge.

Access to hardware structures. Hardware structures in MapleBoard that participate in cache coherence operations are exposed to MapleDSL for designers to use in their protocol specifications. This allows the designer to control the management of memory requests and responses based on the protocol specifications. For example, consider a specification that requires a core to prioritize write-back responses based on the criticality level of the remote core's memory request that

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

TABLE I: LOCs of protocol specification.

Protocol	LOCs in MapleDSL	LOCs in Verilog	% reduction
PMSI	63	518	91%
PMESI	76	1009	94%
CARP	133	981	90%

```
// PMSI private cache protocol transitions
class PMSICacheCoherence extends CoherenceTableGenerator () { Map (
    // ...
    Event(Data,  IM_D)-> Action(State(M), MSHR.cleanAndRespond(),
                         TAG.insert(dirty=true), PR.remove(),
                         DATA.update()),
    // More transitions ...
)}
```

Fig. 5: PMSI [6] specification in MapleDSL.

triggers a write-back. In MapleDSL, a protocol designer can realize this specification by defining multiple PWBs based on the criticality level, and queuing up write-back responses to cache lines based on the criticality level of the remote core's memory request [2]. On the other hand, the SLICC DSL does not expose the hardware structures that feature in the cache coherence operation. In SLICC DSL, management of memory requests and responses in the cache coherence protocol is done transparent to the protocol designer. As a result, deriving WCL bounds requires non-trivial investigation of the hardware structures and their management.

Generation of synthesizable HDL. Built upon Chisel, MapleDSL is designed to describe the actions taken by the hardware components in MapleBoard. For example, state transitions in MapleDSL have direct access to hardware components such as PR, PWB and PRLUT that map directly to synthesizable HDL modules. Actions in state transitions map directly to control and data signals of the corresponding hardware components. On the other hand, the state transitions in SLICC only have access to components that *model* real hardware. These components, such as the TBETable, are pure software behavioral description of their hardware counterparts. Moreover, the actions in SLICC state transitions are designed to be translated into C++ code and it requires non-trivial transformation from software-based code to HDL description. As a result, the compiler of MapleDSL can generate synthesizable HDL from coherence protocols described in MapleDSL, a hardware aware DSL.

C. Protocol specifications in MapleDSL

A protocol state transition for a cache line in MapleDSL has two components: (1) *Event* and (2) *Actions*.

Event. The *Event* takes as input observed events on the bus (coherence messages or data), the current coherence state and an optional relative criticality level. During application execution, a core's cache controller computes the relative criticality level based on its criticality level and the criticality level of the observed coherence message, and selects the appropriate event based on the relative criticality level. Each cache controller stores its criticality in a criticality register. The request broadcasted on the bus includes the criticality of the issuing cache controller.

Actions. For an event on a cache line, the *Actions* describe the set of actions to be executed by cache or shared memory controllers, and the next coherence state the cache line transitions to. Controller actions operate on hardware structures in MapleBoard such as the per-core PR and PWB buffers, per-core miss status handling registers (MSHR), tag and data arrays of caches, and shared memory PRLUT. MapleDSL also allows a designer to define new hardware structures and add actions on these newly defined hardware structures.

Illustrative example. Figure 5 shows a protocol transitions in the private cache for the PMSI protocol [6] in MapleDSL. The private cache transitions are encapsulated in `PMSICacheCoherenceTable`. State transition highlighted in Figure 5 is triggered when data response is observed on bus and the cache line is currently in `IM_D` state (`Event(Data, IM_D)`). A cache line in `IM_D` state denotes a store request on the cache line that is waiting for data. On receiving the data, the cache controller records the data and completes its store request with four controller actions for this state transition: (1) the outstanding request in the MSHR is removed and the data is sent to the core with `cleanAndResponse()`, (2) the data is recorded in a cache line and is marked as dirty (`TAG.insert(dirty=true)`, `DATA.update()`), and (3) remove the pending request from the PR (`PR.remove()`). According to the PMSI protocol specifications, a cache line in `IM_D` state changes to `M` on receiving data. These controller actions and the final coherence state (`M`) are specified in the actions as shown in Figure 5. The coherence protocol state transitions in the shared memory controller is defined in a similar way.

D. Conciseness of MapleDSL

We use lines of code (LOCs) to provide an abstract perspective of the conciseness of MapleDSL. Table I shows the LOCs to describe the PMSI, PMESI, and CARP implementations in MapleDSL, and the corresponding lines of code in the generated Verilog model that describes the protocol implementation in hardware. We agree that there are differences between the grammar of Chisel and Verilog where the latter only provides limited support for crafting a DSL, and it is challenging to compare the two languages fairly. Nevertheless, we use lines of code to provide an abstract perspective on the conciseness of MapleDSL.

Note that it takes around a minute to generate Verilog for a 4-core PMSI configuration from Chisel, and the compilation of the PMSI protocol takes less than a second. The Verilog model is generated from the MapleDSL model by the Chisel compiler. Across all three implementations, MapleDSL offers more potential for rapid protocol design (91% average reduction in LOCs) compared to specifying the same protocol as a Verilog model. Note that the generated Verilog models for a coherence protocol specification defined in MapleDSL are the same across configurations with different core counts.

[2]Note that the arbitration policy across multiple PWBs must be described separately from the protocol specification.

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

On average, MapleDSL reduces the LOCs needed to specify protocol specification by more than 91% compared to a Verilog model.

V. Case study using MapleBoard

MapleBoard enables designers to explore designs on both predictable cache coherence protocols and the broader scope of the memory systems. To illustrate the use of MapleBoard, we present a case study that explores the relationship between the WCL of a memory request under predictable cache coherence protocols and the communication bus configurations.

A. Exploring data bus configurations for predictable cache coherence protocols

Cache coherence protocols communicate data and coherence messages between cores and shared memory via bus interconnects. For performance and hardware overhead considerations, these buses have different design configurations such as width, atomicity, and hardware distribution. In this section, we explore the impact of *data* bus hardware distribution (*shared* and *dedicated* buses) and atomicity (*atomic* and *non-atomic*) on the WCL of a memory request under predicable cache coherence protocols using MapleBoard. A key takeaway from our exploration is that predictable cache coherence protocols deployed on non-atomic *dedicated* bus configurations where each core has dedicated non-atomic data buses to the shared memory have lower WCL of memory requests and better application performance than those deployed on atomic and non-atomic shared bus configuration.

1) Data bus configurations: In this case study, we explore three data bus configurations.

Atomic bus (ATM): Figure 3a shows the configuration of ATM. In this configuration, cores communicate data with the shared memory through a shared data bus interconnect. Furthermore, this bus configuration process memory transactions from cores *atomically*. This means that the bus configuration prevents a core from communicating coherence messages and data on the shared buses before the current request that is using the buses completes [16].

Shared data bus (SDB): Figure 3b shows the configuration of SDB. SDB allows *non-atomic* transactions on the bus where requests can interleave and are serviced when other cores stall for data response, providing improved performance. Simultaneous accesses to the shared data bus are resolved with a predictable bus arbitration policy. Prior work by Hassan et al. [6], Kaushik et al [7], and Sritharan et al. [8] deployed predictable cache coherence protocols on SDB.

Dedicated data buses per core (DDB): In DDB configuration, each core has two dedicated data buses (one for each direction) for data communication with the shared memory. One bus allows the core to send data to the shared memory (dirty data write-backs), and the other bus allows the shared memory to send data responses to the core. Figure 3c shows the configuration of DDB. This configuration allows a core to send data to the shared memory *and* receive data from the shared memory at any time instance.

2) Preliminaries: The WCL of a memory request under a predictable cache coherence protocol has three components [7]: (1) the latency for a core to broadcast its request on the request bus, or the **request latency**, (2) the latency for another core with an updated copy of the requested data and/or the shared memory controller to place the up-to-date data on the data bus, or the **communication latency**, and (3) the latency of the data response to arrive at the requesting core, or the **response latency**. The latency analysis uses the PMSI predictable cache coherence protocol [6], and assumes the shared bus interconnect deploys a TDM arbitration policy that allocates one slot to each *cache* in the multicore platform. This means that each core is allocated two slots: one slot for its L1 data cache and another slot for its L1 instruction cache. Note that the focus of our case study is to explore the communication bus configurations and to show the benefits of deploying DDB configuration. We use the PMSI protocol as an example and omit the analysis of other coherence protocols.

For this analysis, we use the following symbols:

- L^{acc}: WCL to access the shared memory. We rely on prior works such as [30], [31] for deriving this value.
- N: The number of cores in the multicore platform.
- S: The TDM slot width. The slot width is large enough to complete one read or write memory operation between the core and shared memory. This slot width takes into account the latency to broadcast coherence messages and shared memory data communication.
- c_{ua}: Core under analysis.

3) High-level overview of latency analysis: We summarize the WCL for the PMSI [6], PMESI [22] and CARP [7] predictable cache coherence protocols deployed on different bus configurations in Table II. These protocols were deployed on the SDB bus configuration, and we refer the readers to [6], [7], [22] for the derivation of WCL of memory requests under these protocols. The following lemmas and theorems derive the WCL of a memory request under the PMSI protocol deployed on the DDB configuration. We refer to the PMSI protocol deployed on DDB configuration as PMSI-DDB. We use the notation A to denote a cache line with memory address A.

For the protocols deployed on SDB configuration, the worst-case scenario is when a request from c_{ua} to A waits for all prior cores to complete their write requests to A [6], [22]. Each prior core must wait for its allocated slot to receive A and modify the data contents of A and then must wait for another allocated slot to write back the updated data contents of A to the shared memory. A core writes back the updated data contents of A to the shared memory as a response on observing another core's request to A. On the other hand, for protocols deployed on DDB configuration, a core can write back the updated data contents of A to shared memory *immediately* on observing another core's request to A. Unlike the SDB configuration, dedicated data buses between a core and the shared memory eliminate the need for any arbitration and hence, there is no arbitration latency incurred. Thus, compared to SDB, memory requests under protocols deployed

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

on DDB configuration exhibit lower WCL. Note that in DDB configuration, the shared memory handles simultaneous write-back requests due to dirty cache line replacements in a round-robin fashion and performs write-back requests in the same order as they are accepted.

Under protocols deployed on ATM configuration, a core is guaranteed to complete its memory request in its allocated slot. To achieve this, each allocated time slot is large enough to perform three operations: (1) request broadcast, (2) write-back response from another core due to request, and (3) data response from shared memory to requesting core. As a result, the slot width in ATM is larger than that in SDB and DDB.

4) Latency analysis: We present the latency analysis of PMSI-DDB. Among the three different latency components described in Section V-A2, we focus on the worst-case communication latency analysis. The worst-case communication latency is dependent on the memory activity from other cores and dominates the total WCL [6], [7], [22]. Lemma V.1 describes the worst-case scenario that results in the worst-case communication latency and Lemma V.2 expresses the worst-case communication latency.

Lemma V.1. *A memory request to A from c_{ua} under PMSI-DDB exhibits the WCL when all data cache controllers other than the one of c_{ua} broadcast write requests to A and all instruction cache controllers other than the one of c_{ua} broadcast read requests to B before c_{ua} broadcasts its request to A, and the request to A from each core's data cache causes an eviction of a dirty cache line in their respective private caches.*

Proof. In PMSI-DDB, a core that has a cache line in **M** state means that the core has modified the cache line data contents. As a result, a core performs a data write-back of a cache line in **M** state to shared memory on a cache line eviction due to replacement or on observing a remote core's read or write request. Since each core is allocated two slots (one for instruction access and the other for data access), the worst-case scenario is when the remaining $N-1$ cores broadcast data write requests to A and instruction read requests to B before c_{ua}'s request to A. Furthermore, in the worst-case, each core's request to A causes an eviction to a cache line that is also in **M** state. Recall that the dedicated data buses in DDB configuration enable cores to write back dirty cache lines immediately. As a result, c_{ua}'s memory request to A that is in the **M** state in another core's cache must wait for four memory operations to complete: (1) $N-1$ write-backs of replaced cache lines due to requests to A from remaining cores, (2) $N-1$ updates and write-backs of A to shared memory, (3) $N-1$ instruction accesses to B from remaining cores, and (4) the shared memory to send the updated A to c_{ua}. □

Lemma V.2. *For DDB configuration, the worst-case communication latency of a memory request by c_{ua} is given by:*

$$WCL^{Comm}(c_{ua}) = \Big(4\times N - \lfloor\frac{(2N+1)\times S)}{L^{acc}}\rfloor\Big)\times L^{acc} + L^{acc} \quad (1)$$

Proof. From the worst-case scenario described in Lemma V.1, each prior core communicates with the shared memory four times (instruction access to B, write-back of evicted dirty line due to A request, receive A from shared memory, and write-back of A). As a result, the total latency across all $N-1$ cores is $4\times(N-1)\times L^{acc}$ cycles. Since MapleBoard relies on a host for handling system calls (Section III-B), the arbitration schedule allocates slots to the host for predictable handling of system calls. As a result, the total latency including the host is $4\times N\times L^{acc}$ cycles.

However, c_{ua} does not wait for $4\times N\times L^{acc}$ cycles to receive A as dedicated data buses allow cores to write back to shared memory immediately. As a result, some of these write-back responses from other cores complete before c_{ua}'s request to A is broadcasted. To compute the number of memory requests from other cores that are pending after c_{ua}'s request to A is broadcasted, we first compute the number of write-back responses and data requests that can complete in a TDM period. This is computed as $\lfloor\frac{(2N+1)\times S}{L^{acc}}\rfloor$. Therefore, the remaining number of write-back responses and data requests from other cores that are pending after c_{ua} broadcasts its request is $\Big(4\times N - \lfloor\frac{(2N+1)\times S}{L^{acc}}\rfloor\Big)$. In the worst-case, c_{ua}'s request to A also causes a replacement to a dirty cache line due to capacity miss, which triggers a write-back to shared memory. Hence, $WCL^{Comm}(c_{ua}) = \Big(4\times N - \lfloor\frac{(2N+1)\times S}{L^{acc}}\rfloor\Big)\times L^{acc} + L^{acc}$. □

Theorem V.3. *For DDB-configuration, the total worst-case latency of a memory request by c_{ua} is given by:*

$$\begin{aligned} WCL^{Total}(c_{ua}) &= WCL^{Req}(c_{ua}) + WCL^{Comm}(c_{ua}) \\ &\quad + WCL^{Resp}(c_{ua}) \\ &= (2N+1)\times S + \Big(4N - \lfloor\frac{(2N+1)\times S}{L^{acc}}\rfloor + 2\Big)\times L^{acc} \end{aligned} \quad (2)$$

Remarks The WCL^{Total} of a memory request under PMSI-DDB grows linearly with the number of cores (N). On the other hand, the WCL^{Total} of a memory request under PMSI-SDB grows quadratically with N as shown in Table II. This is because a shared data bus between cores and the shared memory (SDB configuration) forces each core to wait for their allocated time slot on the data bus to complete their write-back response to shared memory. As a result, a memory request under PMSI-DDB has lower WCL than PMSI-SDB. Figure 6 shows the analytical WCL bounds for PMSI-DDB, PMSI-SDB and PMSI-ATM when the number of cores changes and $S = 256$ cycles and $L^{acc} = 150$ cycles.

B. Other design space exploration avenues with MapleBoard

In the preceding subsection, we presented one case study to explore the impact of different hardware configurations on the WCL guarantees under predictable hardware cache coherence mechanisms. While the tools discussed in the work such as MapleDSL allow for rich design space exploration of predictable hardware cache coherence protocols, MapleBoard enables several other areas for design space exploration that may be of interest to real-time system designers and users.

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

TABLE II: $WCL^{Total}(c_{ua})$ for ATM, SDB and DDB

Protocol	ATM	SDB	DDB
PMSI	$(6N+5) \times S + L^{acc}$	$2(2N+1)(N+2) \times S + L^{acc}$	$(2N+1) \times S + (4N - \lfloor \frac{(2N+1) \times S}{L^{acc}} \rfloor + 2) \times L^{acc}$
PMESI	$(6N+5) \times S + L^{acc}$	$2(2N+1)(N+2) \times S + L^{acc}$	$(2N+1) \times S + (4N - \lfloor \frac{(2N+1) \times S}{L^{acc}} \rfloor + 2) \times L^{acc}$
CARP	-	$2(2N+1)(N+2) \times S + L^{acc}$	-

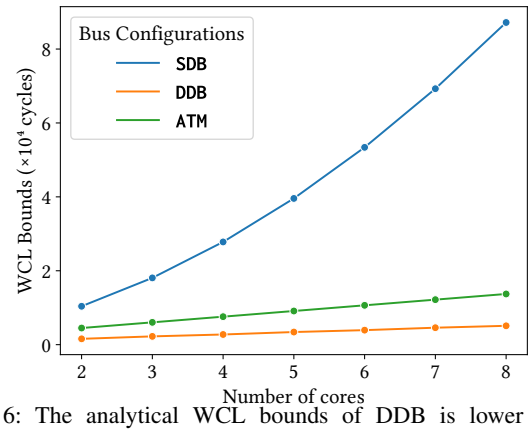


Fig. 6: The analytical WCL bounds of DDB is lower than the analytical WCL bounds of ATM and SDB for the same number of cores.

This is because MapleBoard is fully customizable and researchers can customize MapleBoard at different levels to fit their design space exploration interests. Some potential exploration avenues with MapleBoard include predictability and performance trade-offs of shared memory arbitration schemes, and cache update and replacement policies.

VI. Evaluation

We implemented conventional cache coherence protocols MSI, MESI [16] and predictable cache coherence protocols PMSI, PMESI, CARP [6], [7] in MapleBoard, and synthesized their implementations to hardware logic on the on-board FPGA in the Xilinx Virtex Ultrascale+ VCU1525 board. The operating frequency of the resulting MapleBoard implementations is 100MHz. In Section VI-D, we describe the hardware utilization of these protocol implementations for different core counts (2, 4, and 8 cores) and data bus configurations (shared and dedicated configurations). We use the notation **`[P.N.B]`** to describe the configuration where **`P`** is the coherence protocol, **`N`** is the core count, and **`B`** is the data bus configuration. Possible values for **`P`** are **`MSI`**, **`MESI`**, **`PMSI`**, **`PMESI`** and **`CARP`**. Possible values for **`B`** are Shared and Dedicated for shared and dedicated data bus configurations, respectively. For example, **`[PMSI.2.SDB]`** represents the configuration of a 2-core systems deployed with PMSI coherence protocol and SDB configuration. In Section VI-A, we execute benchmarks from the SPLASH-2 benchmark and synthetic workloads, and report the total execution time and the observed WCL. We show that the observed WCLs across all implementations are within their respective analytical WCL bounds. For all implementations, the private caches are implemented as 4-way set associative caches, and the shared buses deploy TDM bus arbitration policies with slot width set to 256 cycles. For the PMSI and PMESI implementations, we use an arbitration schedule that allocates two slots to each core: one for the instruction cache and one for the data cache. For the CARP implementation, we mark one core as non-critical, and mark the other cores as critical for synthetic benchmarks. The arbitration schedule in CARP allocates slots to cores marked as critical and memory requests from non-critical cores are serviced in unused allocated slots (slack slots).

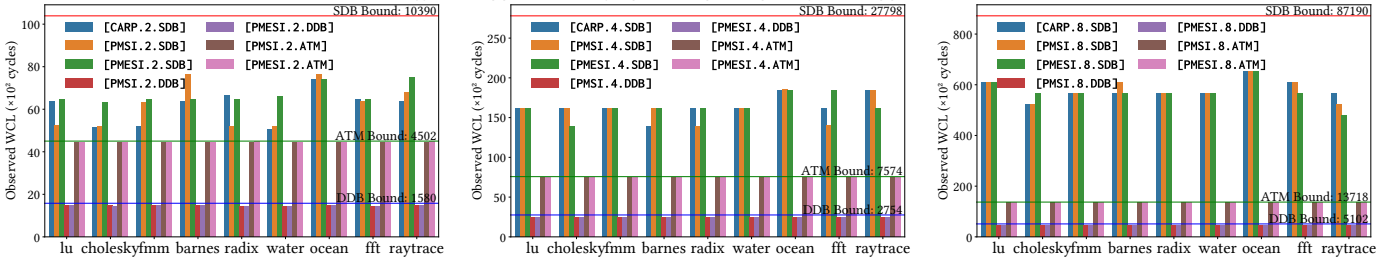
A. SPLASH-2 and synthetic workloads

We execute the multi-threaded benchmarks in the SPLASH-2 benchmark suite, and synthetic workloads on the different MapleBoard implementations [32]. We executed these benchmarks on the MapleBoard implementations to completion, and we ensured the correctness of the multi-threaded computations using the in-built single-threaded verification routines in the SPLASH-2 benchmark suite. To highlight the predictability properties of MapleBoard, we designed a synthetic workload that stresses the protocol implementations and exercised the worst-case scenarios for the different protocol implementations. The synthetic workload forced all cores to issue memory requests to the *same* memory location at the same time.

We derive the WCL memory request latency bounds for the protocol implementations on the shared data bus, the dedicated data bus and atomic data bus configurations taking into account the multicore model on MapleBoard. Sections VI-A1 and VI-A2 presents the total execution time and observed worst-case memory request latency for the SPLASH-2 and synthetic workloads for different protocol implementations on MapleBoard respectively.

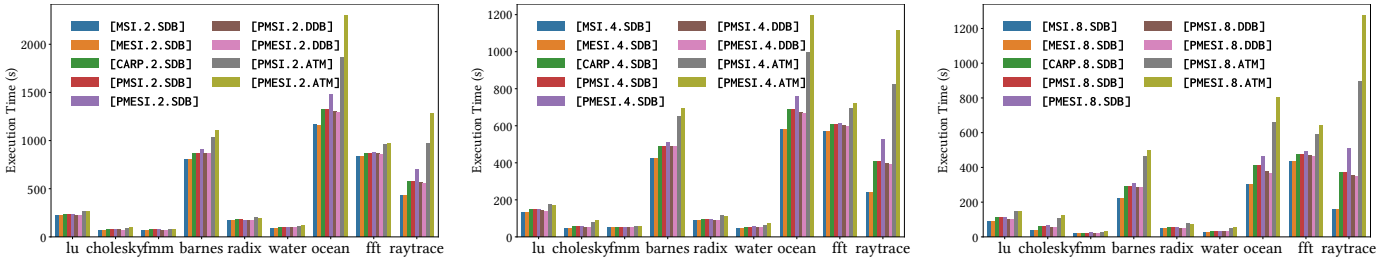
1) Observed WCL: Figure 7 shows the observed WCL for the SPLASH-2 benchmarks (Figures 7a-c). The observed total WCL is the maximum memory request latency to shared data across all cores. Note that the SPLASH-2 benchmarks have thread barriers that require all threads to update before making forward progress. Hence, all cores that execute a SPLASH-2 benchmark must be able to read *and* write to shared data for correct benchmark execution. In the CARP implementation, non-critical cores cannot write to shared data [7], so all cores are critical cores in our evaluation for SPLASH-2 benchmarks.
Observations. We make two observations. First, for all benchmarks (SPLASH-2 and synthetic), the observed WCL is within the analytical WCL bounds across all implementations. We also confirmed that the individual latency components that make up the WCL such as the arbitration latency, communication latency, and data response latency are within their

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.



(a) Worst-case latency for 2 cores (b) Worst-case latency for 4 cores (c) Worst-case latency for 8 cores

Fig. 7: Observed WCL for SPLASH-2 workloads for different protocol implementations on MapleBoard.



(a) Execution time for 2 cores (b) Execution time for 4 cores (c) Execution time for 8 cores

Fig. 8: Execution time for SPLASH-2 benchmarks of MapleBoard.

respective analytical WCL bounds across all implementations. Note that for synthetic workloads on CARP, we report the observed WCL for the critical cores; non-critical cores are not provided with predictability guarantees [7].

Second, the disparity between the observed WCL and analytical WCL bound in the SPLASH-2 benchmarks is because of the following two reasons. (1) The SPLASH-2 benchmarks partition the data on which computations are performed across multiple cores, which minimizes the data sharing between cores during execution. (2) The host does not perform write requests to the shared data at the same time as the other cores to stress the worst-case scenario. On the other hand, the synthetic benchmark stresses the worst-case scenario as shown in Table III. Note that for the synthetic benchmark, we ensure that the host does *not* access the same cache lines operated on the cores. Hence, the analytical WCL bounds for the synthetic benchmark does not include the host.

2) Total execution time: Figure 8 shows the execution time of SPLASH-2 benchmarks on the PMSI and PMESI protocols with the shared, dedicated and atomic data bus configurations. We also include MSI and MESI deployed on shared bus. Recall from Section III-A that each core has two dedicated data buses to the shared memory in the dedicated data bus configuration. The atomic bus only processes the next memory transaction after current memory transaction is finished. As a result, the DDB configuration trades hardware cost for improved performance over the SDB configuration.

Observations. We make two observations. First, across all benchmarks, the DDB configurations for each protocol implementation performs better than the corresponding SDB configuration. For example, for PMSI and PMESI implementations with 4-cores, the DDB configurations offer 2.59% and 13.89% average performance improvements over the corresponding SDB configurations respectively. This is because the DDB configuration eliminates any arbitration delay to communicate data between the shared memory and cores. Thus, the shared memory can send the requested data to cores and cores can write back data to the shared memory at any time instance. The SDB configuration requires cores to communicate data on the shared data bus in their allocated time slots.

Second, for the SDB data bus configuration, the PMESI implementation exhibits higher execution time compared to the PMSI implementation (12.84% on average). In PMESI, a core's read request receives the requested cache line in the exclusive state (**E**) if it is the only core that will have the cache line in its private cache. As a result, the core with a cache line in the **E** state can complete a write request to the cache line without broadcasting any coherence messages. To this end, the shared memory marks a cache line it sends to a core in the **E** state as dirty. As a consequence, an eviction of a cache line in the **E** state triggers a write-back even though the core did not update the data contents of the cache line. We observe that evictions of cache lines in the **E** state in the instruction and data caches in the PMESI implementation increase the execution time of benchmarks in the PMESI implementation. On the other hand, the PMSI implementation receives cache lines due to read requests in the **S** state, and evictions of cache lines in **S** state do not trigger write-backs. DDB minimizes this performance drawback of PMESI implementation as the write-backs need not wait for the allocated time slots to complete. Third, the ATM provides the worst performance across the benchmarks despite providing worst-case latency that scales linearly with N. The reason for the degraded performance is a result of absence of interleaving requests. Moreover, the latency for every request is similar and close to the worst-case latency. Finally, the MSI and MESI provides

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

TABLE III: Observed WCL and Analytical WCL for synthetic benchmark on MapleBoard in cycles.

Core #	SDB WCL	SDB Observed WCL			DDB WCL	DDB Observed WCL		ATM WCL	ATM Observed WCL	
		PMSI	PMESI	CARP		PMSI	PMESI		PMSI	PMESI
2	7830	5227	5235	3948	1580	1462	1471	4502	4457	4459
4	23190	23021	16200	18455	2604	2453	2450	7574	7536	7536
8	78486	69731	78445	60913	4802	4539	4513	13718	13683	13683

the best average performance across all the implementations. In MSI and MESI, the memory requests are serviced with first-come-first-serve arbitration. Unlike PMSI and PMESI with predictable arbitration where each core has the same chance to be serviced, cores with more memory requests are more likely to be serviced as they can issue the request more frequently and do not suffer from arbitration delays.

B. Discussions on bus configurations

1) Shared data bus and dedicated data bus: In Section V, we analytically show that the WCL of a memory request under PMSI or PMESI deployed on the DDB configuration is lower than the WCL of a memory reqeust under PMSI and PMESI deployed on the SDB configuration. In Table III, the observed WCLs of memory requests under PMSI and PMESI deployed on SDB and DDB are within the analytical WCLs for 2, 4 and 8 cores. For PMSI and PMESI, the observed WCLs of memory requests on DDB are lower compared to SDB for 2, 4 and 8-core setup. Thus, our results empirically show that the WCL of a memory request on DDB is lower than the WCL of a memory request on SDB.

2) Atomic data bus and dedicated data bus: In Section V, we show the WCLs of memory requests on ATM and on SDB for PMSI grow linearly with N. However, the WCL of a memory request on DDB is lower than ATM, which we validate with empirical data. In Table III, the PMSI deployed on DDB shows lower observed WCLs than the PMSI with ATM for 2, 4 and 8 cores respectively.

C. Protocol validation

MapleDSL compiler maps each transition in MapleDSL directly to one branch in the generated combinational logic. As a result, MapleDSL compiler accurately translates MapleDSL specification into a coherence state machine.

We validated the protocols implemented in our platform using a variety of methods. (1) We generate random requests in Chisel to verify the correctness of the implementation by running RTL simulation. (2) We use hand-crafted synthetic micro-benchmarks as state transition coverage tests in Chisel by running RTL simulation. During testing, we record every exercised state transitions and make sure that all valid transitions are exercised and that invalid transitions are not present. (3) We integrate the platform in QEMU [33] to perform full-system RTL simulation. The full-system simulation is capable of running pthread application. (4) We synthesize the platform on an FPGA and the platform can successfully execute SPLASH-2 benchmarks and various micro-benchmarks. Throughout all the steps of validation, the worst-case observed latencies are within the calculated bounds for predictable cache coherence protocols. For conventional cache coherence protocols, the worst-case observed latencies exceed the calculated bounds for their respective counterpart.

D. Hardware utilization

Table IV shows the hardware utilization of MapleBoard implementations between the RISC-V cores, private caches, buses, and shared memory controller for different configurations.

Observations. The row highlighted in gray shows the hardware utilization of private cache in PMSI with SDB and DDB configurations across 2, 4 and 8 cores. We observed that the private caches account for most of the LUT utilization, and is the largest hardware component in MapleBoard. Note that while a cache holds the data content in BRAMs, it also contains logic such as PR, PWB that utilize LUTs. Caches consuming most of the resources applies to all other implementations. The shared memory controller implementation is sensitive to the data bus configurations where the shared memory controller has higher LUT utilization in the dedicated data bus configuration. Cells highlighted in blue shows an example comparing the shared memory controller of **`[PMSI.2.SDB]`** and **`[PMSI.2.DDB]`**. This is because the dedicated data bus configuration adds more input and output port interfaces to the shared memory compared to the shared data bus configuration. The bus interconnects (request and data buses) have low hardware utilization across all implementations, and accounts for less than 2% of the total hardware logic of MapleBoard. The PMESI implementation has close resource utilization compared to PMSI since the differences between the implementation of PMESI and PMSI are only in the coherence table. The CARP implementation has higher hardware utilization than PMSI for 4 and 8 cores. The red cells show an example comparing **`[CARP.4.SDB]`** and **`[PMESI.4.SDB]`**. This is because CARP introduces additional hardware such as criticality registers and non-critical PWB per core, and stores criticality information for each pending request in the shared memory controller [7]. Note that for two cores, CARP utilizes less LUTs in private caches, bus and cores compared to PMSI and PMESI, which might be a result of synthesis tool optimization.

VII. Related Works

Prior works by Salloum et al. [18] (ACROSS MPSoC) and Schoeberl et al. [19] (T-CREST) are examples of open-source multi-core platforms for safety-critical systems. MapleBoard differs from the above platforms in two main ways. First, MapleBoard facilitates designers to rapidly prototype and configure predictable hardware extensions using the Chisel

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

TABLE IV: Hardware utilization of MapleBoard implementations.

Modules	LUT	BR	FF	LUT	BR	FF	LUT	BR	FF	LUT	BR	FF	LUT	BR	FF	LUT	BR	FF
	[PMSI.2.SDB]			[PMSI.2.DDB]			[PMSI.4.SDB]			[PMSI.4.DDB]			[PMSI.8.SDB]			[PMSI.8.DDB]		
SMC	9009	0	7642	10680	0	7673	15448	0	13076	17988	0	13113	27821	0	23953	33160	0	23977
Cache	27383	180	21815	27601	180	21670	52828	324	43846	52857	324	43370	112487	612	100132	98096	612	94171
Bus	453	0	156	448	0	156	773	0	195	755	0	195	1370	0	276	1848	0	261
Cores	8212	0	6020	8212	0	6020	16566	0	12040	16564	0	12040	32745	0	24080	31248	0	24187
	[PMESI.2.SDB]			[PMESI.2.DDB]			[PMESI.4.SDB]			[PMESI.4.DDB]			[PMESI.8.SDB]			[PMESI.8.DDB]		
SMC	9008	0	7650	10706	0	7686	15193	0	13091	18173	0	13120	28289	0	23865	33192	0	24000
Cache	27210	180	21820	27418	180	21675	52604	324	43855	52639	324	43379	96630	612	95718	97925	612	94217
Bus	460	0	161	455	0	161	786	0	204	767	0	204	2221	0	279	1261	0	278
Cores	8210	0	6020	8211	0	6020	16566	0	12040	16564	0	12040	31252	0	24187	31250	0	24187
	[CARP.2.SDB]			[CARP.2.DDB]			[CARP.4.SDB]			[CARP.4.DDB]			[CARP.8.SDB]			[CARP.8.DDB]		
SMC	9462	0	7717	10985	0	7758	19511	0	13387	21525	0	13397	50381	0	24877	55897	0	25015
Cache	26283	180	23320	26589	180	23119	55874	324	52527	55674	324	51738	136046	612	140740	134172	612	137678
Bus	390	0	148	373	0	147	772	0	190	1092	0	188	1408	0	275	1857	0	272
Cores	7979	0	6187	7982	0	6187	15915	0	12361	15916	0	12361	31255	0	24187	31253	0	24187
	[MSI.2.SDB]			[PMSI.2.ATM]			[MSI.4.SDB]			[PMSI.4.ATM]			[MSI.8.SDB]			[PMSI.8.ATM]		
SMC	8964	0	7605	8881	0	7670	15689	0	13014	15024	0	13093	27648	0	23851	27736	0	23951
Cache	24093	180	20640	23888	180	19057	46063	324	41640	42293	324	34170	99123	612	95695	79572	612	64356
Bus	505	0	214	465	0	210	914	0	283	951	0	258	1550	0	381	1796	0	338
Cores	7984	0	6187	7980	0	6187	15918	0	12361	15917	0	12361	31246	0	24187	31251	0	24187
	[MESI.2.SDB]			[PMESI.2.ATM]			[MESI.4.SDB]			[PMESI.4.ATM]			[MESI.8.SDB]			[PMESI.8.ATM]		
SMC	8964	0	7605	8885	0	7656	15705	0	13004	15229	0	13106	27648	0	23851	28216	0	23922
Cache	24093	180	20640	23892	180	19062	46004	324	41597	42465	324	34177	99123	612	95695	78664	612	64367
Bus	505	0	214	468	0	215	862	0	273	797	0	267	1550	0	381	2453	0	355
Cores	7984	0	6187	7978	0	6187	15913	0	12361	15921	0	12361	31246	0	24187	31234	0	24187

HDL [13], and the MapleDSL for specifying predictable cache coherence protocols. On the other hand, [18], [19] were not designed keeping in mind configurability, which limits their use for design space exploration. Rather, they were designed as complete implementations for chip tape-out. Second, cores in MapleBoard implement the RISC-V ISA. As a result, MapleBoard leverages the RISC-V toolchain to execute applications on the modeled hardware in MapleBoard [25], [34]. On the other hand, T-CREST and ACROSS MPSoC rely on custom software tool chains as they implement customized ISAs.

The ESP platform [35] is a recent open-source research platform for exploring heterogeneous system-on-chip (SoC) designs for embedded platforms. Similar to MapleBoard, ESP accepted hardware designs in Chisel HDL, and the multicore model in ESP was based on the RISC-V ISA [35]. However, the purview of MapleBoard is different from that of the ESP platform. In particular, the ESP platform enabled designers to explore and prototype heterogeneous on-chip accelerators. ESP provided a tiled architecture that allowed for seamless integration of accelerators and facilitated communication between accelerators [35]. On the other hand, MapleBoard enables designers to explore and prototype predictable hardware extensions including predictable cache coherence protocols to real-time multicore platforms. Furthermore, hardware components in MapleBoard are designed keeping in mind predictability requirements of real-time applications, whereas ESP platform is not specifically designed for real-time systems. Recently, Kandikar et al. [24] designed the FireSim simulation framework that used cloud FPGAs to primarily accelerate simulations of data center-scale hardware models. The FireSim framework is an open-source framework that accepts hardware designs implemented in Chisel HDL, and allows the input designs to be integrated with different RISC-V multicore models [36]. MapleBoard can be simulated using the FireSim framework.

Recent works such as Pendulum [8] and DISCO [37] investigate mechanisms to enable predictable data sharing with cache coherence. Pendulum [8] is a time-based cache coherence protocol for MCS, in which each cache line per core has two timers to configure how long a cache line stays valid before it invalidates itself in response to another critical or non-critical core. DISCO, on the other hand, prevents modified shared data to be cached and removes the coherence delays when accessing modified cache lines at the cost of longer latency for write accesses to shared data. These techniques are alternatives to guarantee predictable memory accesses and are orthogonal to this work.

VIII. Conclusion

In this work, we present MapleBoard, a set of open-source hardware tools to prototype and evaluate cache coherence protocols in hardware. The tools in MapleBoard consist of a domain specific language (MapleDSL) for describing cache coherence protocols, and a real-time multicore platform with a memory hierarchy, predictable bus interconnects, and a software runtime component to execute single-threaded and multi-threaded workloads. We use MapleBoard to propose a new mechanism for communication, DDB, which tightens the WCL compared to SDB. We validate the effectiveness of DDB analytically and empirically with MapleBoard. We successfully synthesized MapleBoard implementations with different core counts, predictable cache coherence protocols, and bus interconnect configurations on the Xilinx Virtex Ultrascale+ VCU1525 board.

Z. Wu, A. M. Kaushik, and H. Patel, "A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores ," in proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2021, pp. 1–12.

REFERENCES

[1] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *2012 Ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[2] J. Bin, S. Girbal, D. Gracia Pérez, A. Grasset, and A. Mérigot, "Studying co-running avionic real-time applications on multi-core COTS architectures," in *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, Feb. 2014.

[3] L. Cavigelli, M. Magno, and L. Benini, "Accelerating real-time embedded scene labeling with convolutional networks," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[4] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst, "System level performance analysis for real-time automotive multicore and network architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 979–992, 2009.

[5] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. LIPIcs, M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 10:1–10:20.

[6] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 235–246.

[7] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "Carp: A data communication mechanism for multi-core mixed-criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 419–432.

[8] N. Sritharan, A. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 433–445.

[9] A. Bansal, J. Singh, Y. Hao, J. Wen, R. Mancuso, and M. Caccamo, "Reconciling predictability and coherent caching," in *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, 2020, pp. 1–6.

[10] N. Sensfelder, J. Brunel, and C. Pagetti, "Modeling Cache Coherence to Expose Interference," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 18:1–18:22.

[11] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, "Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 57–68.

[12] 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.

[13] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.

[14] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 209–216.

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

[16] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, Nov. 2011.

[17] E. Bost, "Hardware support for robust partitioning in freescale qoriq multicore socs (p4080 and derivatives)," Freescale Semiconductor, Inc., Tech. Rep., 2013.

[18] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek, "The across mpsoc – a new generation of multi-core processors designed for safety-critical embedded systems," in *2012 15th Euromicro Conference on Digital System Design*, 2012, pp. 105–113.

[19] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann *et al.*, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, pp. 449–471, 2015.

[20] A. Cortex, "Cortex-A9 MPCore," *Technical Reference Manual*, 2009.

[21] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[22] A. M. Kaushik, M. Hassan, and H. Patel, "Designing predictable cache coherence protocols for multi-core real-time systems," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[23] "chisel-testers2," 2020. [Online]. Available: https://github.com/ucb-bar/chisel-testers2

[24] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[25] "Risc-v gnu compiler toolchain," 2019. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain

[26] S. Hahn and J. Reineke, "Design and analysis of sic: A provably timing-predictable pipelined processor core," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, dec 2018, pp. 469–481.

[27] S. A. Edwards and E. A. Lee, "The case for the precision timed (pret) machine," in *2007 44th ACM/IEEE Design Automation Conference*, 2007, pp. 264–265.

[28] B. Cilku, B. Frömel, and P. Puschner, "A dual-layer bus arbiter for mixed-criticality systems with hypervisors," in *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, 2014, pp. 147–151.

[29] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys*, vol. 50, no. 6, Nov. 2017. [Online]. Available: https://doi.org/10.1145/3131347

[30] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015, pp. 307–316.

[31] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 157–167.

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

[33] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC'05. USA: USENIX Association, 2005, p. 41.

[34] "riscv/riscv-tests," 2019. [Online]. Available: https://github.com/riscv/riscv-tests

[35] L. P. Carloni, "Invited: The case for embedded scalable platforms," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[36] K. Asanovic, D. A. Patterson, and C. Celio, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[37] M. Hassan, "Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems," in *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, ser. LIPIcs, M. Völp, Ed., vol. 165. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 16:1–16:24.