PENDULUM: A Cache Coherence Protocol for Mixed Criticality Systems, A Technical Report

NIVEDITA SRITHARAN, University of Waterloo
ANIRUDH MOHAN KAUSHIK, University of Waterloo
MOHAMED HASSAN, McMaster University
HIREN PATEL, University of Waterloo

1 INTRODUCTION

In this technical report, we explain the details of the PENDULUM, the cache coherence protocol proposed for Mixed Criticality Systems in [1]. The coherence protocol is realized in the hardware through a state machine deployed by the cache controller in the private cache of each core. We first explain the main three stable states deployed by the traditional Modified-Shared-Invalidate (MSI) protocol, then we detail the coherence messages, and then finally we explain the transient states.

2 STABLE STATES

PENDULUM has the same three stable states as the Modified-Shared-Invalidate (MSI) protocol. The semantics of these states are as follows: 1) Invalid (I) indicates that the cache line does not have valid data. 2) Modified (M) represents that the core has modified the cache line data; hence, it *owns* the cache line, and has the most up-to-date data. Only one core can have a cache line in the *M* state. This is also referred to as maintaining the single writer multiple reader (SWMR) invariant. 3) Shared (S) identifies that the cache line was read, but not modified. Multiple cores may have the same cache line in the *S* state. This allows read hits in their respective private caches.

3 COHERENCE MESSAGES

In bus snooping coherence, data correctness is maintained across different cores by exchanging messages on the bus. A core will broadcast a message on the bus to indicate a new coherence request or state change that other cores have to be aware of to maintain data correctness. There are three standard coherence messages that conventional MSI cache coherence protocols include, which we illustrate in Table 1. In addition to these messages, PENDULUM also introduces three new coherence messages: SelfInv, AllInv, and SendData, which we explain in Table 2.

Table 1. MSI coherence messages

message	Explanation
GetM	Coherence message is broadcasted on stores to signal that data is going to be modified.
GetS	Coherence message on loads to signal that data is going to only be read.
PutM	Coherence message on dirty cache line replacements to signal that data needs to be written back.

4 TRANSIENT STATES

If the bus connecting all cores in the system is an atomic in-order bus, then each initiated request will complete without interference from other requests. Therefore, a request that is being serviced will move from its initial stable state to another stable state before the system starts servicing another

Authors' addresses: Nivedita Sritharan, University of Waterloo; Anirudh Mohan Kaushik, University of Waterloo; Mohamed Hassan, McMaster University; Hiren Patel, University of Waterloo.

Table 2. New coherence messages introduced by PENDULUM

message	Explanation
SelfInv	Coherence message is broadcasted when the core needs to downgrade or self-invalidate from a shared to invalid state.
AllInv	Coherence message is broadcasted by the shared memory when all sharers have self-invalidated. This coherence message is necessary to signal a pending store to a shared data that all sharers have self-invalidated, which eliminates violation of the fundamental Single-Writer-Multiple-Reader invariant.
SendData	Coherence message is broadcasted by a core that is ready to send a cache line to a requestor

request. However, an atomic in-order bus is not practical since it introduces significant performance degradation. The bus arbiter has to stall all cores, while a request is waiting for its data. This data can take hundreds of cycles if it is fetched from main memory. Unnecessarily stalling other requests, which can be ready and can be serviced immediately. Therefore, most commodity systems implement non-atomic out-of-order buses to enhance performance. In such buses, the stall time of one core waiting for data response can be used to service requests of other cores. However, the cache controller has to remember the status of this waiting interrupted request. This is achieved by introducing transient states that indicate the status of each request, while it is waiting to move to a final stable state. Tables 3 and 4 shows all possible states, coherence messages, and transitions between these states. We categorize the transient states of PENDULUM into three categories.

1) Transient states to indicate the waiting for data messages. 2) Transient states to indicate the waiting for coherence messages. 3) Timer-related Transient states introduced by PENDULUM. In addition to these three categories, we also explain in details the states that are criticality-aware, i.e. their transitions differ between Cr and nCr cores in Section 4.4.

4.1 Transient States to Indicate the Waiting for Data Messages

These states indicate that the request message has been sent, but the data response has not yet been received. Examples of these states are: IS^D and IM^D. IS^D indicates that a load request was issued to a cache line that was originally in I state but it is still waiting for the data to be read. Similarly, IM^D indicates that a store request was issued to a cache line that was originally in I state but it is still waiting for the data to be written. If a core Core*i* is waiting for data in IS^D (IM^D) state and another core issues a store request, then Core*i* moves to the IS^DI (IM^DI) state which means that after Core*i* performs its load (store) operation, it moves eventually to the I state (immediately in case of traditional MSI and after a timeout in case of PENDULUM) to enable the other core to perform its store operation. In case of IM^DI, a write back of the data is also needed.

4.2 Transient States to Indicate the Waiting for Coherence Messages

Due to the non-atomic nature of the bus, a core can issue a coherence request on the bus but observes a request from another core on the bus before it observes its own issued request.

4.2.1 Transitioning from I to S or M. Consider the case between I and S stable states. A core that has a load request to a cache line in I state issues a GetS coherence request to the bus and moves to the IS^{AD} transient state. Once the request observes its OwnGetS, it moves to the IS^D state to wait for its own data as explained earlier. A similar situation occurs between I and M states. A core that has a store request to a cache line in I state issues a GetM coherence request to the bus and moves to

the IM^{AD} transient state. Once the core observes its OwnGetM, it moves to the IM^{D} state to wait for its own data.

4.2.2 Transitioning from S or M to I. Now, consider the transition from M to I. To replace a block in M state, the core issues a PutM request to the bus. Until it observes its OwnPutM on the bus, the cache line stays in the MI^{A} state. Once OwnPutM is observed, it sends the data to the memory and moves to the I state. A similar behavior occurs when downgrading from S to I. If the core wants to replace a cache line in I state, it issues a selfInv message to the bus. Until it observes this OwnSelfInv message on the bus, the cache line stays in the SI^{A} state. Once OwnSelfInv message is observed by the core, it moves to the SI state waiting for the replacement to occur and then move to I state. In the meantime, if a core has a cache line in SI state and observes an AllInv message on the bus, it also invalidates and moves to I state.

4.2.3 Transitioning from S to M. In PENDULUM, if a core has a store request to a cache line that it owns in S state, it has to wait for its timer to expire and then issue this store request to the arbiter [1]. Therefore, if a core has a store request to a cache line in S state, it moves to $S^{T}M$ (timer states are explained in Section 4.3), waits for timer expiration, and then issues a SelfInv followed by a GetM message to the arbiter and move to the SM^{A} state. If the core observes its OwnSelfInv, it moves to the IM^{AD} state, and waits to observe the OwnGetM message. If it observes its OwnGetM, it moves to the IM^{D} state.

4.3 Novel Timer-Related Transient states introduced by PENDULUM

In this subsection, we discuss coherence states that are related to the timers operation (shaded in grey in Table 3), while we discuss coherence states that are related to the criticality-awareness in Subsection 4.4. The following rules dictate the operation of timers and the corresponding PENDULUM states and transitions.

(1) Once a core receives a cache line, it starts its corresponding timers (ST in Table 3). In addition, if the cache line was in the $IS^{D}I$ or $IM^{D}I$ state, it moves to the $S^{T}I$ or $M^{T}I$ state, respectively, and waits for the timer to expire before it invalidates itself. The cache line will be in the $IS^{D}I$ or $IM^{D}I$ state if while waiting for its data, a store request is issued by another core as detailed in Section 4.1.
(2) If a timer times out while the cache line is not requested by another core with the same criticality considered by the timer, the timer is replenished (RT in Table 3). This is important to avoid unnecessary invalidations.
(3) If a core owns a cache line in S state and another core requests the same cache line to modify (OtherGetM), the owner moves to $S^{T}I$ state, waits for its timer to expire (WT), and then issues a self-invalidation to the arbiter (SelfInv).
(4) If a core owns a cache line in M state and another core issues a request to this cache line, a similar process to (3) occurs with the exception that the owner moves to $M^{T}I$ state.
(5) Multiple cores can share same cache line in the read-only state (e.g. S) without modification . In this case, every sharer has its own timer counting based on when it obtained this cache line independent of the other cores. A requestor to that cache line has to wait for the core whose timer expires the last before it can obtain the cache line.
(6) If a core has a store request to a cache line that it owns in S state, it has to wait for its timer to expire and then issue this store request to the arbiter. In other words, write hits to non-modified cache lines are disallowed. This is necessary to maintain coherent data sharing, while simplifying the analysis to provide latency guarantees. Therefore, as shown in Table 3, if a core has a store request to a cache line in S state, it moves to $S^{T}M$, waits for timer expiration, invalidates, and then issues the GetM to the arbiter.

4.4 Handling Mixed Criticality

As aforementioned, deploying a criticality-aware arbiter to protect Cr requests from interference by assigning them higher priority is not enough for MCS if tasks are simultaneously sharing data. In such system, requests to shared memory are not necessarily atomic, since the data transfer might not start on the same slot in which the request was issued. This might happen for instance because another core already has the requested data in its private cache. In MCS, the cache coherence protocol has to ensure data correctness, while determining actions based on the criticality of memory requests. Otherwise, a Cr request may have to wait indefinitely because of nCr request to same cache line. Accordingly, some of PENDULUM state transitions are dependent on the owner core as well as the requestor core. These transitions are identified in Table 3 as Criticality-dependent and are detailed in Table 4. It is important to observe that these transitions and their corresponding states cover the situation of pending requests where a core issued a request to a cache line but is still waiting to get the data (this is the *owner* core), and in the mean time another core requests an access to the same cache line (this is the *requestor* core). The transitions in Table 4 can be classified into two categories as follows.

(1) The owner's criticality is higher than or equal to the requestor's criticality. In this case, the owner gets the data first, performs its access, and then invalidates after its corresponding timer's expiry. If the owner's request was a store, the owner sends the data as well after the timer expiration to the requestor. This is the case in Table 4, where the owner is Cr regardless of the requestor's criticality, or both the owner and the requestor are nCr.

(2) The owner's criticality is lower than the requestor's criticality. This is the case where the owner is a nCr core, while the requestor is Cr. In this case, the owner's request will be preempted and the core has to reissue its request. This preemption only affects the nCr's request latency and has no effect on the system state. This is because the request was still pending and data transfer did not yet start. On the other hand, this preemption is necessary to provide the Cr cores with the sufficient independence from the nCr cores behavior on shared data, and thus, ensure a bounded latency for requests from Cr cores.

Table 3. PENDULUM private cache coherence states. Shaded rows are the timer-related states. WT: Wait for timer timeout, RT: Replenish timer, ST: Start timer. $msg/state$ denotes that a core issues the message msg, and moves to coherence state $state$. Cells marked as "×" indicate that a particular transition cannot happen, and cells marked as "–" denote that a cache line in that state does not change state with a core event or bus event.

State	Core events				Bus events - common to Cr and nCr cores								
	Load	Store	Replacement	Timeout	OwnGetS	OwnGetM	OwnPutM	OwnSelfInv	AllInv	OwnSendData	Data	OtherGetS-Cr or nCr	OtherGetM-Cr or nCr
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}	X	X	X	X	X	-	-	X	X	-	-
IS^{AD}	X	X	X	X	/IS^{D}	X	X	-	-	X	X	-	-
IS^{D}	X	X	X	X	X	X	X	-	-	X	load/S and ST	Criticality-dependent (Table 4)	
$IS^{D}I$	X	X	X	X	X	X	X	-	-	X	load/$S^{T}I$ and ST		
S	hit	/$S^{T}M$ and WT	issue SelfInv/SI^{A}	/S and RT	X	X	X	X	X	X	X	-	issue SelfInv/ $S^{T}I$ and WT
$S^{T}I$	hit	$S^{T}M$ and WT	issue SelfInv/SI^{A}	/SI^{A}	X	X	X	X	X	X	X	-	issue SelfInv
SI^{A}	hit	issue SelfInv and GetM/SM^{A}	issue SelfInv	X	X	X	X	/SI	X	X	X	-	issue SelfInv
SI	hit	issue GetM/IM^{AD}	/I	X	X	X	X	X	/I	X	X	-	-
$S^{T}M$	hit	X	X	issue SelfInv and GetM /SM^{A}	X	X	X	X	X	X	X	-	issue SelfInv
SM^{A}	hit	X	X	X	X	/IM^{D}	X	/IM^{AD}	X	X	X	-	issue SelfInv
IM^{AD}	X	X	X	X	X	/IM^{D}	X	-	-	X	X	-	-
IM^{D}	X	X	X	X	X	X	X	-	-	X	store/M and ST	*	*
$IM^{D}I$	X	X	X	X	X	X	X	-	-	X	store/M^{I} and ST	*	*
M	hit	hit	issue PutM/MI^{A}	/M and RT	X	X	X	X	X	X	X	issue SendData / $M^{T}I$ and WT	issue SendData / $M^{T}I$ and WT
$M^{T}I$	hit	hit	issue PutM and SendData/M	/MI^{A}	X	X	X	X	X	X	X	issue SendData	issue SendData
MI^{A}	hit	hit	issue PutM	X	X	X	WB/I	X	X	send data to requestor/I	X	issue SendData	issue SendData

Table 4. Different PENDULUM transitions based on criticality.

Owner Core	State	Requesting Core			
		Cr		nCr	
		load	store	load	store
Cr	IS^D	-	issue SelfInv /IS^DI	-	issue SelfInv /IS^DI
	IS^DI	-	issue SelfInv	-	issue SelfInv
	IM^D	issue SendData /IM^DI	issue SendData /IM^DI	issue SendData /IM^DI	issue SendData /IM^DI
	IM^DI	issue SendData	issue SendData	issue SendData	issue SendData
nCr	IS^D	-	reissue GetS /IS^{AD}	-	issue SelfInv /IS^DI
	IS^DI	-	reissue GetS /IS^{AD}	-	issue SelfInv /IS^DI
	IM^D	reissue GetM /IM^{AD}	reissue GetM /IM^{AD}	issue SendData /IM^DI	issue SendData /IM^DI
	IM^DI	reissue GetM /IM^{AD}	reissue GetM /IM^{AD}	issue SendData /IM^DI	issue SendData /IM^DI

REFERENCES

[1] Nivedita Sritharan, Anirudh M. Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 1–11, December 2019.