A Systematic Methodology to Develop Resilient Cache Coherence Protocols*

Konstantinos Aisopos
Princeton University
Princeton, NJ, USA
kaisopos@princeton.edu

Li-Shiuan Peh
Massachusetts Institute of Technology
Cambridge, MA, USA
peh@csail.mit.edu

ABSTRACT

Aggressive transistor scaling continues to increase integration capacity with each new technology node, but technology downscaling also increases the vulnerability of semiconductor devices and causes silicon failures. Thus, fault-tolerant architectures are emerging to guarantee reliable functionality on unreliable silicon. While tolerating faults within a processor core has been extensively researched, the many-core era introduces the challenge of reliable on-chip communication in Chip Multi-Processors (CMPs). In CMP systems, an unreliable interconnection network can lose or corrupt coherence messages, causing the entire chip to deadlock. In this work, we argue for a system-level resiliency solution to tolerate an unreliable underlying Network-on-Chip (NoC). We introduce a systematic methodology to transform a coherence protocol to a resilient one, by extending its Finite State Machine (FSM) with safe states and incorporating additional handshaking messages into transactions. The modified protocol ensures coherent and reliable transactions over any lossy NoC. Our approach is generic and can be applied to a wide range of protocols. It requires minimal hardware modifications and introduces only a slight performance overhead (an average of 0.8% during fault-free operation, and 1.9% even at an aggressive fault rate of one fault per msec).

Categories and Subject Descriptors

B.4.5 [**Hardware**]: Input/output and data communications—*Reliability, Testing, and Fault-Tolerance*

General Terms

Design, Reliability

Keywords

Coherence Protocol, Resilience, Fault Tolerance

*The authors acknowledge the support of NSF (grant no. CPA-0702341) and Gigascale Systems Research Center, a research center funded under the Focus Center Research Program, a Semiconductor Research Corporation entity.

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil

1. INTRODUCTION

Cache-coherent Chip Multi-Processors (CMPs) have become mainstream in the marketplace with an ever-increasing core count and a sophisticated interconnect fabric, often a Network-on-Chip (NoC). At the same time, aggressive transistor scaling is increasingly threatening the reliability of such chips, since shrinking the critical dimensions of semiconductor devices raises the probability of occurrence for transient and permanent faults. Transient (or soft) faults are much more common than permanent faults [21], especially in caches, and industry expects a significant increase in their fault rates at advanced technology nodes, due to higher integration and lower power consumption. Texas Instruments projects that the soft error rate will exceed 50,000 FITs[1] per chip [6], while MoSys projects an error rate of 10,000 to 100,000 FITs (per megabit, in 0.13-micron technology), which brings the frequency of errors down to months or weeks [13].

Though researchers have extensively explored techniques to protect data in caches and memory, most resilience solutions assume a NoC that always reliably transfers data among cores. However, recent work on fault modeling [3] has indicated that NoCs fabricated at advanced technology nodes become increasingly unreliable, causing a number of faults, such as loss or corruption of network messages. In order to mitigate this trend, resilient NoC designs have been proposed to allow correct operation in the face of faults in router hardware [9, 12] and links [4, 18]. However, no resilient NoC can guarantee 100% reliable data transfers (see Section 6). Thus, a subset of network faults is expected to be exposed to upper layers causing lost coherence messages [3], or corrupted coherence messages that are dropped at the destination node when the packet checksum is recomputed [3, 7]. Unfortunately, the loss of a single coherence message can cause the entire system to deadlock. Consequently, a resiliency solution that can tolerate an unreliable underlying NoC is critical for viable future CMP architectures.

Resilience at the coherence protocol level is currently addressed with checkpointing mechanisms [17, 20] that roll back to a previous safe state. Checkpointing mechanisms work in a pro-active manner, by logging all data changes from a coherent state of the system, to rollback upon loss of coherence messages. Such approaches result in complexity, high storage overheads, and often a performance overhead during error free execution [17]. In light of high expected

[1]Failures In Time: 1 FIT is equivalent to 1 error per billion hours of device operation.

fault rates, we suggest exploring re-active approaches, to lower the overhead of resilience. We propose a re-active approach that extends a coherence protocol to detect deadlocks, and retransmit lost messages for recovery. The benefit of such a re-active approach is that no preventive data replication is required; the protocol can re-read from caches previously transmitted data and re-generate lost messages.

Recently, two resilient protocols were proposed to maintain coherence over an unreliable NoC. *FTTokenCMP* [11] is a token-based protocol and *FTDirCMP* [16] is a directory-based protocol, modified to tolerate loss of coherence messages. Though these protocols effectively tackle network faults, they utilize protocol-specific heuristic approaches to achieve resilient functionality, limiting their applicability to a wider range of protocols (see Section 6).

In this paper, we take resilient coherence protocols a step further: we present a systematic methodology to incorporate resiliency into a wide range of coherence protocols. Our methodology modifies the coherence protocol's Finite State Machine (FSM) by adding safe states, handshake messages, and a retransmission rule, so that caches always remain coherent and all transactions complete, in the face of lost messages. We also demonstrate how to apply our methodology by presenting two case studies on blocking protocols: one for a directory-based coherence protocol and one for a broadcast-based coherence protocol. In both studies, we showcase that the resulting resilient protocols work correctly across many applications in the face of substantial faults, while exhibiting negligible performance degradation. Specifically, we measure an average performance overhead of 0.8% during fault-free operation, and 1.9% for the aggressive fault rate of 1 fault per millisecond. We believe that our methodology is also applicable to non-blocking protocols, such as token coherence [14] (see Section 5.3).

This paper is organized as follows: Section 2 describes our methodology and Section 3 presents two case studies of applying it to specific protocols. Then, Section 4 offers our experimental results. Section 5 discusses the corner cases of our approach, its hardware overhead, and its applicability to non-blocking protocols. Finally, Section 6 presents the related work, and Section 7 concludes the paper.

2. METHODOLOGY TO DEVELOP A RESILIENT COHERENCE PROTOCOL

A coherence transaction begins when a data/ownership request is generated and the requestor (initiator of the transaction) transits to a transient state. The initiator returns to a stable state when its request has been served, while the transaction completes once all nodes participating in the transaction have also transitioned back to a stable state. However, an unreliable NoC may lose messages and prevent a transaction from completing. We consider a transaction where at least one node remains indefinitely transient, waiting for a lost message to transit back to stable state, as "suspended". In this section, we argue that a "resilient" coherence protocol may recover from a suspended transaction by *detecting* suspension and *replaying* the transaction identically as the first time. Below, we define the properties that a resilient coherence protocol has to observe, and provide a sketch of a proof that these properties are sufficient to recover from suspension. Then, we present a systematic methodology to modify the states, transitions, and messages of a protocol, so that these properties are adhered to.

Property1: All initiators of transactions stay in a transient state till all other nodes involved in the transaction have completed their part and transitioned back to a stable state.

Property2: Previously transmitted messages can be retransmitted: nodes retain sufficient information to regenerate any previous message for each outstanding transaction.

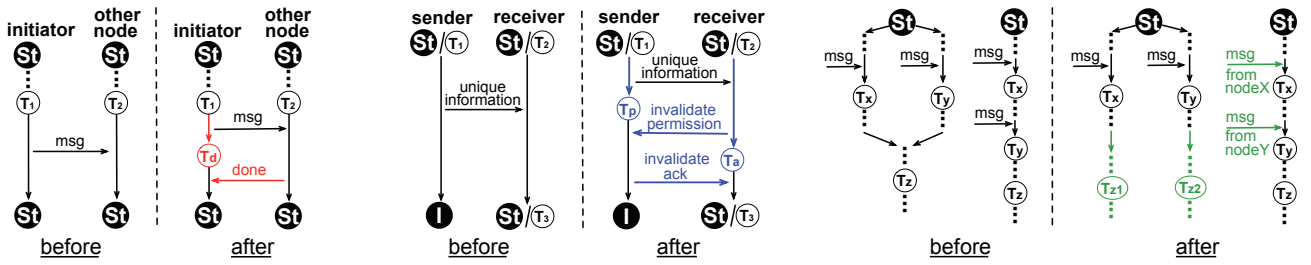
Property3: All nodes involved in a transaction can tolerate duplicate messages and still produce the same outcome, *i.e.* transition to the same state and generate the same message.

Sketch of proof. Here, we argue that a resilient coherence protocol can recover from a bounded number of message losses by providing an intuition that (i) the initiator of a transaction can always detect when the transaction is suspended and resend its request, and (ii) the resulting replayed transaction progresses identically to the original one and eventually completes. We assume that timeout counters are available in each initiating node to trigger the retransmission of a request after a timeout[2]. We also assume that only a single transaction per memory address can be served at each point in time (the protocol is blocking). This restriction guarantees that both original and replayed transactions do not interfere with other transactions for the same address.

(i) *Can the initiating node always detect message loss?* Property1 requires the node that initiates a transaction to remain in a transient state throughout the transaction, in other words to be the last node to transition back to a stable state. This property ensures that if any coherence message of the transaction is lost, leading to a number of nodes indefinitely remaining in transient state (suspended transaction), the initiator will always be one of these nodes. It can thus detect that the transaction exceeded the timeout period and trigger retransmission of its request.

(ii) *Will a replayed transaction complete identically as the original transaction would have completed?* Properties 2, 3 force all nodes participating in a transaction to retain any information required for message regeneration, and exhibit the same behavior, *i.e.* produce a unique outcome, when receiving a message for the first time and anytime thereafter. Thus, upon retransmission of the initiator's request, its receiver(s) will produce the same outcome, *i.e.* transit to the same state(s) and regenerate the same set of message(s) as before. Now, the newly generated set of message(s) will also transit its receiver(s) to the same set of state(s) and regenerate the same message(s) as before. By induction, all nodes involved in the transaction will eventually transit to the exact same state as in the original transaction, and the transaction will subsequently compete, *unless* an additional message is lost. Upon any number of additional message losses, the initiator will always replay the transaction identically as before, until it completes within the timeout. Since we assume a bounded number of message losses, the transaction will complete after an arbitrary number of retransmissions, identically to the baseline transaction.

[2]Section 5.1 discusses how the timeout threshold is appropriately chosen to prevent the duration of a transaction exceeding it, and how we resolve this corner case, if it occurs.



(a) enforcing Property1 (b) enforcing Property2 (c) enforcing Property3

Figure 1: Violated properties and corresponding transformations.

In the remaining of this section, we present our methodology as templates for detecting property violations in a protocol, and a resulting template for transforming the original protocol to a resilient one that satisfies these properties.

2.1 Property1: Initiating Node Must Remain Transient Throughout the Transaction

Property description. Property1 requires the initiator of a transaction to remain in a transient state, as long as any other nodes participating in the transaction are in transient state, and be the last node to transition back to a stable state. Thus, while the transaction is outstanding, the initiator is always waiting for a coherence message, while the last message of any transaction is always destined to the initiating node and enables transition to a stable state.

Property violations. This property is not satisfied in all coherence protocols. For instance, in directory based-protocols, transactions typically complete with an unblock message from the initiator of the transaction to the directory. This unblock message signifies that the initiator has received the requested data or ownership, so the directory may now safely process the next request for this address. In this example, Property1 is violated because though the transaction has not been completed (the directory is still in transient state), the initiator transitions to a stable state. Consequently, if the unblock message is lost, the initiator will not be able to detect the suspended transaction.

Property enforcement. To enforce this property, the coherence protocol needs to be modified so that the initiator remains in transient state after its request has been serviced, until any other nodes that might be in transient state transition back to stable state. To achieve this, we examine each outgoing message from the initiator of a transaction that transitions to a stable state: a violation is committed if the outgoing message causes other nodes to transition to a stable state. Figure 1a demonstrates such a violation, since *msg* from the initiator causes some node to transition from T_2 (transient) to *St* (stable). We prevent this violation by introducing an additional T_d transient state into the initiator's FSM as shown in Figure 1a. Once the recipient of *msg* transitions to a stable state, it sends a *done* message to the initiator, signifying the completion of the transaction and enabling the initiator to transition back to stable state. Note that if multiple *msg* messages are sent from the initiator when transitioning to T_d state, then multiple *done* responses (one from each recipient of *msg*) are required to transition back to a stable state. In practice, once a request has been serviced, all previous sharers are invalid and only a directory or coherence manager may remain transient.

2.2 Property2: Previously Transmitted Messages Can Be Regenerated

Property description. Property2 requires all nodes to retain sufficient information to regenerate any previously transmitted message for all pending transactions. So, throughout a transaction, no information can be discarded after being transmitted, until it is guaranteed that the corresponding message has been successfully received (thus the same message will never need to be retransmitted in the future).

Property violations. However, in most coherence protocols, there are data transfers between nodes where the sender invalidates its data copy right after transmission. For instance, this happens when a sharer caching modified data receives an exclusive request: it then forwards its unique data copy to the requestor and marks the cache line as invalid. Now, the sharer can no longer regenerate the data message. Besides discarding unique data, this property can also be violated when any protocol-specific unique information is discarded from the cache line state (or the Miss Status Holding Register, MSHR, if applicable) after transmission, such as tokens [14], snoop ordering IDs [1], *etc.* We note that there is no automated way to infer which information is unique from the message tables, thus the protocol designer needs to manually identify any unique protocol-specific information/variables, so that messages piggybacking this information are protected as detailed below.

Property enforcement. To adhere to Property2, we modify the coherence protocol to incorporate a triple handshake when transmitting unique information, where the information is discarded only after an acknowledgment that the message has been successfully received. Figure 1b shows how we transform a generic transaction which is violating Property2, through new states and messages. First, we insert an additional transient state in the sender's FSM before transitioning into invalid state (T_p in Figure 1b). Once the unique information is received, the receiver transitions to T_a and sends an invalidate permission message to the sender, indicating that the information can now be safely discarded. This permission enables the transition from T_p to invalid state. Finally, the sender acknowledges the reception of the invalidation permission and the receiver can proceed to sending succeeding messages.

Baseline L1 states

stable	**M**odified
	Exclusive
	Shared
	Invalid
transient	**IM** (I → M)
	IS (I → S)
	SM (S → M)
	ISI (IS → I)
	MI (M → I)

9 baseline states

Resilient states	Property #
Md (M, waiting done)	P1
Ed (E, waiting done)	P1
Sd (S, waiting done)	P1
Id (I, waiting done)	P1
Sp (S, waiting inv. perm)	P2
Ip (I, waiting inv. perm)	P2
Ma (M, waiting inv. ack)	P2
Sa (S, waiting inv. ack)	P2

total 17 states (5 bits)

(a) directory-based coherence protocol

Baseline L1 states

stable	**M**odified
	Owned
	Exclusive
	Shared
	Invalid
transient	**IM** (I → M)
	IS (I → S)
	SM (S → M)
	SE (S → E)
	SS (S → S)
	OM (O → M)
	WB (WB req.)

12 baseline states

Resilient states	Property #
Md (M, waiting done)	P1
Ed (E, waiting done)	P1
Sd (S, waiting done)	P1
Id (I, waiting done)	P1
MIb (MI, waiting done)	P1
Sp (S, waiting inv. perm)	P2
Ip (I, waiting inv. perm)	P2
Ma (M, waiting inv. ack)	P2
Ea (E, waiting inv. ack)	P2
Sa (S, waiting inv. ack)	P2

total 22 states (5 bits)

(b) broadcast-based coherence protocol

Table 1: **Additional resilient states** introduced by our methodology.

2.3 Property3: Nodes Can Tolerate Duplicate Messages and Produce the Same Outcome

Property description. In most coherence transactions, multiple messages are exchanged between sharers (and potentially a directory or a coherence manager) to reassign the ownership permissions and transfer data. Though the initiator of a transaction can detect when the transaction for a specific cache line is suspended, the actual message that was lost and the exact state of each node involved in the transaction is not always known. For example, if the initiator requests a data copy and does not receive any response within the timeout period, there is a high chance that its request was not delivered by the underlying NoC. However, it is also possible that the data response from the sharer that is caching the modified copy was lost. The initiator will attempt to replay the transaction from the last known state, by retransmitting its latest message (its request in this example). This can lead to duplicate messages received at multiple nodes involved in the transaction. For instance, if the sharer caching the modified copy received the initiator's request but its data response was lost, a duplicate request for the same data copy will be received. Property3 requires that a duplicate request produces the same outcome as the initial request, *i.e.* each node will transition to the same state and potentially generate the same coherence message. Thus, if the node has proceeded to any later state of the transaction, once an earlier message is received again, it has to roll-back to the earlier state it transitioned to when receiving this message for the first time. In order to identify the precise state to roll-back to, each earlier message type should lead to a unique state in every potential path of the node's FSM.

Property violations. Any FSM state where a previously received message type does not lead to a unique state violates Property3. Thus, FSM branches that contain the same message violate Property3 upon merging. The first FSM in Figure 1c depicts two such disallowed branches, merging into transient state *Tz*. Note that *msg* leads to a different transient state (*Tx* or *Ty*) in each branch. Property3 is also violated when the same message is received multiple times in the same branch (second FSM in Figure 1c), which occurs when a node receives multiple identical messages (*e.g.,* when an exclusive requestor receives invalidation acknowledgments from sharers). In both cases, the node does not know which preceding state to backtrack to (could be *Tx* or *Ty*) in response to a duplicate *msg* while in *Tz* state.

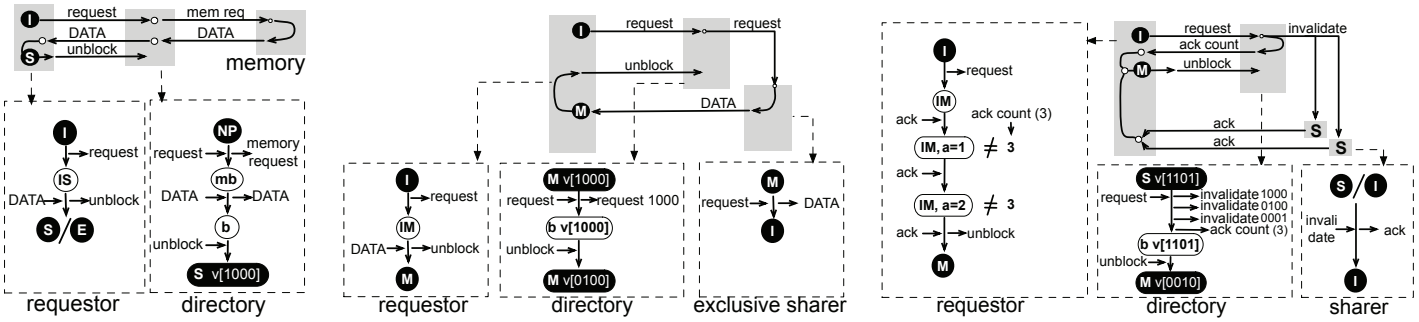
Property enforcement. To enforce Property3, we first scan the FSM of each node to detect distinct branches which contain the same message and merge into a transient state. Then, we replicate the shared branch (from the merging point to transitioning back to a safe stable state) to dissociate the branches as shown in Figure 1c. Next, we scan the FSM of each node to identify branches where an identical message is received multiple times. In all coherence protocols we are aware of, when a node receives multiple identical messages during the same transaction, these are sent by different nodes. However, the identity of the sender is not used as a part of the recipient's state (which consists of the cache state and a message counter), resulting in each message communicating identical information. To make each message distinguishable, we modify the recipient's state: instead of counting the number of received messages, it maintains the vector of message senders together with its cache state. Thus, each message is now distinct, since it also communicates the identity of its sender to the recipient. We note that this modification results in a cleaner and easy-to-validate protocol, while also protecting the coherence layer from misrouted messages[3]. We showcase a detailed example of implementing this in our case study (Section 3).

3. CASE STUDIES: GENERATING RESILIENT COHERENCE PROTOCOLS

3.1 A Resilient Directory-Based Protocol

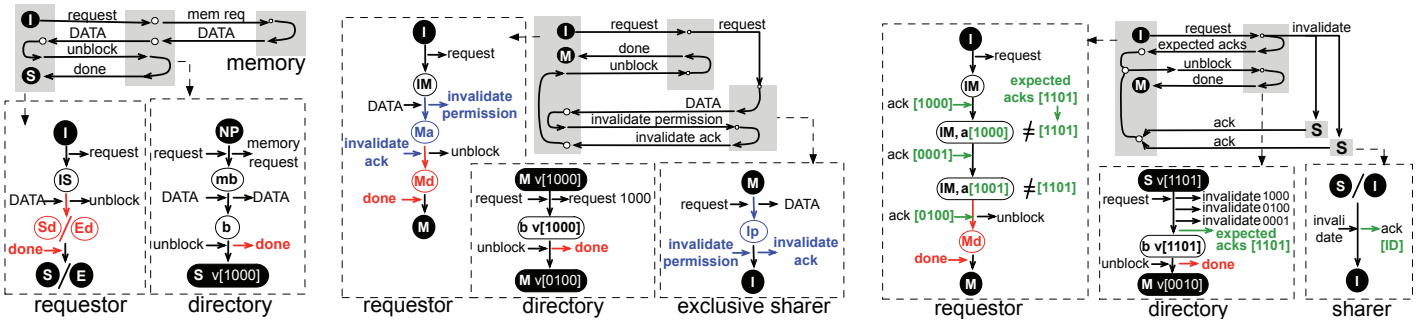
We define as "directory-based", a coherence protocol with the following properties: all data/ownership requests are sent to a directory node (potentially different for each address), which caches the vector of sharers for the corresponding cache block. The directory either directly responds to the requestor providing the data, or forwards the request appointing this task to a node caching the data (*i.e.,* a sharer).

[3]When maintaining the vector of expected senders, the recipient can identify messages intended for other nodes but mistakenly delivered to the recipient (*i.e.,* misrouted), due to a routing fault in the unreliable underlying NoC. These messages should be dropped, since the initiator of the corresponding transaction will ensure that they will be eventually retransmitted to the correct recipient.



(a) data: NP/S, request: S (b) data: M, request: M/S (c) data: S, request: M

Figure 2: **Directory-based coherence protocol**: original / baseline transactions.



(a) data: NP/S, request: S (b) data: M, request: M/S (c) data: S, request: M

Figure 3: **Directory-based coherence protocol**: modified (resilient) transactions.

In case of exclusive data requests, the directory also invalidates all sharers before the requestor accesses the data. Our case study assumes a MESI directory-based protocol with private L1s and a physically distributed shared L2, where the unique L2 copy piggybacks the directory vector. Figure 3 demonstrates how we modify the transactions and cache states of the baseline protocol (Figure 2), to produce a resilient protocol. Also, Table 1a shows the additional transient states that need to be encoded in L1 cache (no additional states required in directory/L2 cache), together with the properties which prompted their addition. We separately consider the following categories of data requests:

- requesting *S* or *NP* data for *S* ownership (Figure 2a)
- requesting *M* data for *M* or *S* ownership (Figure 2b)
- requesting *S* data for *M* ownership (Figure 2c)

Property1. Figure 2a shows the simplest coherence transaction, where the directory can directly serve a coherence request (no invalidations nessesary). If the directory does not cache an up-to-date copy of the data (as shown in the figure), it first generates a memory request to retrieve the data from memory. Transient states are shown in white (*e.g., mb* and *b* stand for the directory being blocked, waiting for the memory and requestor respectively to respond), while stable states are shown in black (*e.g.,* the requestor's state is marked as Invalid (*I*) at the beginning and as Shared (*S*) or Exclusive (*E*) at the end of the transaction). Note that the directory state piggybacks the vector of sharers v. Figure 3a shows the corresponding resilient transaction: the transaction is now completed with a *done* message destined to the requestor. That is because, to adhere to Property1, the requestor should remain in a transient state until all nodes have transitioned to a stable state, in order to detect a potential suspension.

The requestor may detect a suspended transaction while in *IS* or *Sd/Ed* state. The recovery mechanism will regenerate its latest transmitted message (*request* for *IS* and *unblock* for *Sd/Ed*). While in *IS*, any of the following messages might have been lost: *request*, *mem request*, *DATA*. Independently of the lost message, *request* will force the directory to recover the memory data again and send it to the requestor. Suspension while in *Sd/Ed* implies that either *unblock* or *done* has been lost. The requestor will regenerate *unblock*, which might find the directory in stable state or serving a succeeding request (if *unblock* has been previously received). Since *unblock*'s sender is not currently served, the directory identifies this as a duplicate message and responds *done*, indicating that this transaction has been previously completed.

Property2. In Figure 2b, a transaction where Modified data is requested for Modified ownership is shown: the directory forwards the request to sharer 1 (*1000*), which is caching the Modified copy (exclusive sharer). The sharer invalidates its copy and forwards the data to the requestor. This transaction violates Property2, since a unique data copy is invalidated during the data transfer, consequently a replicate request to the sharer (if the message with the unique data is lost) cannot be served. Figure 3b demonstrates how a triple handshake is incorporated into the transaction to pro-

tect the unique data copy, as detailed in Section 2.2. We note that even if the data request is for Shared ownership, thus the current sharer can retain its data copy (downgraded in shared *S* state), this triple handshake is still necessary. That is because S state can be invalidated without write-back (during cache replacement). If the downgraded sharer could invalidate its copy while the unique data copy is in-transit to the requestor, upon message loss the sharer would receive a replicate data request, which could not be served.

Property3. Finally, in Figure 2c, a transaction where Shared data is requested for Modified ownership is shown. The directory generates an invalidation request for each node marked as sharer in its sharing bitvector v (*1101*), *i.e.* nodes 1, 2, 4 (*1000*, *0100*, *0001*). In addition, it responds to the requestor (message *ack count*), indicating the number of expected acknowledgements and providing the shared data. As shown in the requestor's FSM, once the expected number of acks (provided by *ack count*) matches the number of received *acks* (a), the requestor can safely transition to Modified state. The requestor's FSM conflicts with Property3 for this transaction: a single message type (*ack*) is received multiple times and leads to multiple states ({*IM,* a=*1*}, {*IM,* a=*2*}, *etc.*) during the same transaction.

Though each *ack* message seems identical, it implies that a different sharer has been invalidated. On the other hand, the requestor does not exploit the identity of each invalidated sharer, since it just counts the total number of invalidated sharers. Consistently with Section 2.3, we modify the requestor's state to maintain the vector of invalidated sharers instead of their count. This scheme also requires the directory to communicate to the requestor the vector of expected acknowledgments (*expected acks*), instead of *ack count*. In Figure 3c, the modified transaction is shown: an *ack* from *nodeX* communicates that *nodeX* has been invalidated and transitions to a state where the *X*th bit of the acknowledgment bitvector (a) is set. Thus, a replicate *ack* message will not affect a, since the corresponding bit is already set. Similarly, a replicate *expected acks* message will just overwrite the existing (identical) vector of expected acknowledgements. The requestor will transition to the succeeding cache state (*Md*) whenever *expected acks* equals the acknowledgment vector (a), independently of how many times each distinct *ack* (from a specific node) has been received, or how many times *expected acks* has been received.

Performance impact of transformations. A key principle of our methodology is that all additional messages, incorporated to make transactions resilient, are only sent after the requestor has been served. No additional message is introduced into the critical path of transferring data/ownership between sharers. For instance, in Figure 3a, the requestor may use the data once received by the directory, though its state remains transient (*Sd* or *Ed*). Note that the data is still received in 4 hops (or 2 hops if the directory caches an up-to-date copy) as in baseline (see Figure 2a). Then, in Figure 3b, data becomes available once *DATA* is received (at which point the requestor transitions to *M*a) which takes 3 hops as in baseline (see Figure 2b). Similarly, in Figure 3c, data becomes available once the directory response and the acknowledgements have been received (*Md*) as in baseline (see Figure 2c). Thus, though more messages are incorporated into transactions (more messages are injected to the network and transactions last longer), no direct delay is introduced when serving a request.

However, an indirect delay might affect a request (compared to the non-resilient baseline protocol) in the following cases: (i) if the request is waiting for a previous transaction to complete, which lasts longer due to the additional resilient handshake messages, (ii) due to higher utilization of caches, since data copies remain cached for longer (till the reception of the invalidate permission, rather than being immediately invalidated), (iii) due to an increase in the average network latency, due to higher network contention as a result of additional handshake messages. For moderate network traffic, sufficiently large caches, and adequate NoC bandwidth, we expect negligible overall performance overhead.

3.2 A Resilient Broadcast-Based Protocol

We define as "broadcast-based", a coherence protocol where all data/ownership requests are broadcasted to all nodes in the system, including the memory node. All sharers and non-sharers acknowledge the reception of the request, while a sharer caching a Modified copy (or owner) provides the up-to-date data. If no such sharer is present, the requestor uses the data retrieved by memory. Once the requestor gathers all responses, it transitions to a stable state and the transaction completes. While such a protocol is easily realizable when the communication medium is a snoopy bus, a mechanism to achieve global ordering among requests is required for coherence protocols built on the top of unordered NoCs (*e.g.,* meshes). This mechanism typically introduces a serialization point. In this section, we assume a broadcast-based coherence protocol similar to that used by AMD (in their Opteron systems), also known as AMD Hammer [2]. AMD Hammer leverages the home node of each memory line as a serialization point: all requests are sent to their home node and ordered there, before being broadcasted.

Baseline. A transaction is initiated with a coherence request being sent to the home node of the memory address. Then, the requestor (transaction initiator) remains in transient state until all nodes have responded with acknowledgments or data. As shown in the requestor's FSM (Figure 4), a transaction completes when the number of responses (a) equals the number of nodes in the system (all nodes have to respond independently of their state). If a sharer caches a Modified copy (exclusive sharer), its response (*i.e., DATA*) also piggybacks the up-to-date data copy. Also, upon reception of a request, an exclusive sharer is invalidated (Modified ownership request) or downgraded to *S* (Shared ownership request). Figure 4 assumes a request for Modified ownership, but the transaction is similar for any other request (the only difference is the requestor/sharer states).

Properties. In Figure 5, we demonstrate how we modify the baseline transaction and its cache states, to produce a resilient coherence protocol. If a sharer caches a Modified copy, a triple handshake is incorporated into the transaction, to protect the unique data copy that is transferred (Property2). The requestor's FSM during the triple handshake is shown in a dashed line. Also, the transaction now completes at the initiating node (with a *done* message) to adhere to Property1. Finally, the requestor maintains the vector of

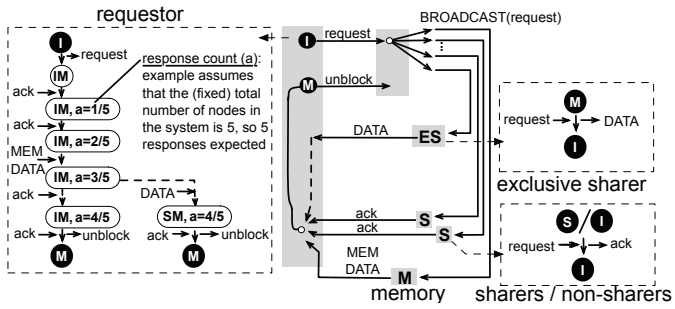


Figure 4: **Broadcast-based coherence protocol**: original / baseline transactions.

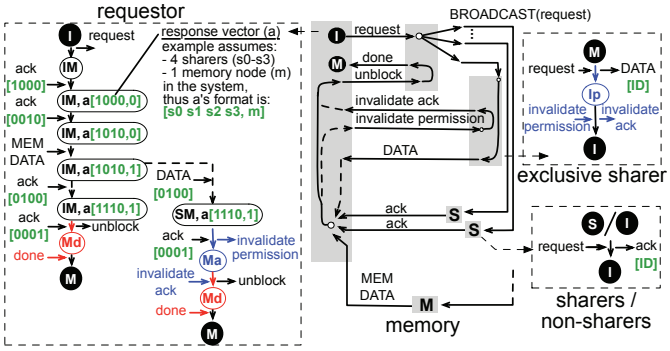


Figure 5: **Broadcast-based coherence protocol**: modified (resilient) transactions.

nodes that have responded, instead of the response count, to adhere to Property3. Each response marks the entry corresponding to its sender as "1" in the acknowledgment bitvector (a); once all bits are set, the request has been served and the data can be delivered to the processor core. Table 1b lists all the additional transient states that are encoded, together with the properties which prompted their addition.

Message retransmission. If the requestor detects that a transaction has been suspended before receiving responses from all nodes, the recovery mechanism will regenerate the *request* message. *Request* will replay the transaction for the very beginning, consequently all sharers and non-sharers will (again) acknowledge. If the Modified sharer has already provided its data copy and already received an invalidate permission, it will acknowledge the request as a non-sharer. Otherwise, if either the data copy has not been sent, or the invalidate permission has not been received (*M* or *Ip* states respectively when replaying), the sharer will resend its data copy. On the other hand, if the requestor has received all responses (*Md* state), the *unblock* message is regenerated.

Performance impact of transformations. Similarly to the directory-based protocol (Section 3.1), no direct delay is introduced when serving a coherence request. The critical path of transferring data/ownership from any sharer to the requestor remains three hops (requestor → serialization node → sharer → requestor) as in baseline. Any additional resilience messages (*done* and potentially *invalidate permission*/*invalidate ack*, if a sharer caches the copy in *M* state) are only sent after the requestor has been served.

System Configuration

Processors		In-order SPARC cores 1GHz clock frequency
L1 Caches (split I&D)	Size Line width Associativity Latency	2 x 32KB / node 64-byte 4-way set associative 3 cycles
L2 Caches	Size Line width Associativity Latency	1MB / node 64-byte 4-way set associative 6 cycles
Memory	Size Controllers Latency	1GB / controller 4 160 cycles
Protocols		MESI_SCMP_bankdirectory MOESI_SMP_hammer

Network-on-Chip

Network Topology	8 x 8 2D Mesh
Channel Width	64-bit
Virtual Networks	5
Memory Controllers	Attached in chip corners
Routing Protocol	XY routing
Arbitration	Queueing RoundRobin arbiters
Message Ordering	Point-to-Point ordered NoC

Table 2: **Simulated system** configured in GEMS.

4. EVALUATION

4.1 Simulation Framework and System Configuration

We simulate a 64-core tiled CMP architecture with in-order SPARC cores (Table 2) in the Wisconsin Multifacet GEMS simulator [15]. For the directory-based coherence protocol, we model private split instruction/data L1 caches and a physically distributed shared L2 cache, where the unique L2 copy piggybacks the directory vector. For the broadcast-based coherence protocol, we model private split instruction/data L1 caches and private L2 caches, where L2 misses are broadcasted to all nodes. We evaluate both SPLASH-2 [19] and PARSEC [8] parallel benchmarks. Each run consists of 64 threads of the application running on the 64-core CMP. For each experiment, we perform ten runs with small random perturbations (we feed ten different random seeds to GEMS) to capture the variability in parallel workloads [5], and then average the results of the runs.

To evaluate resilient functionality, we simulate various fault rates, ranging from 1 fault per millisecond to 1 fault per 10 microseconds. At each rate, we inject faults in a uniform random distribution across all routers' hardware, corrupting the in-transit packets that are buffered in the Network-on-Chip. Packet headers are protected with a Single-Error-Correcting Double-Error-Detecting Hamming code (SECDED) [7], while the entire packet is protected with a checksum. Thus, upon detection of two errors in the header the packet is instantly dropped and the corresponding network resources are deallocated. On the other hand, packets whose data has been corrupted are dropped later, when the checksum is recomputed at the destination node.

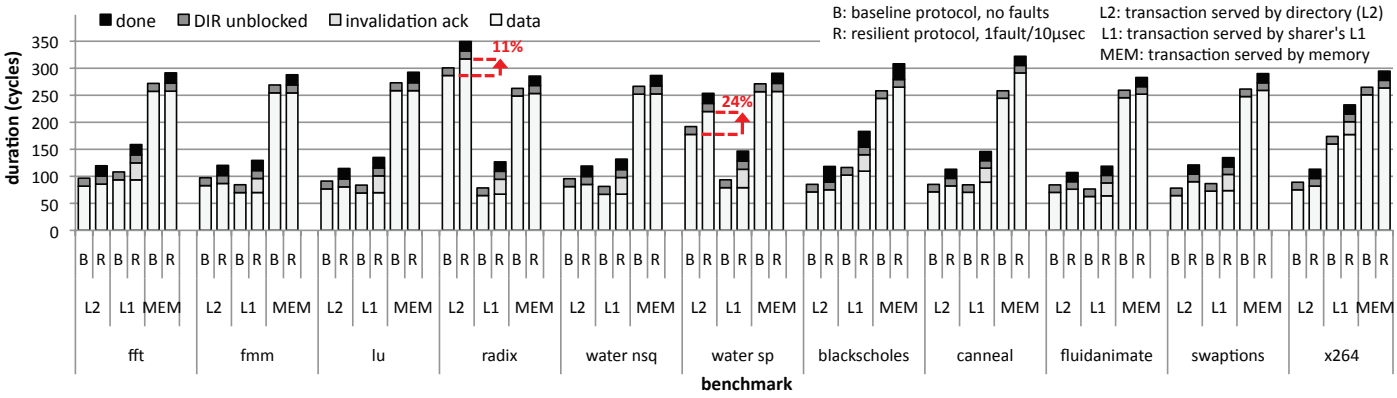


Figure 6: **Transaction duration breakdown**. baseline vs. resilient directory-based coherence protocol.

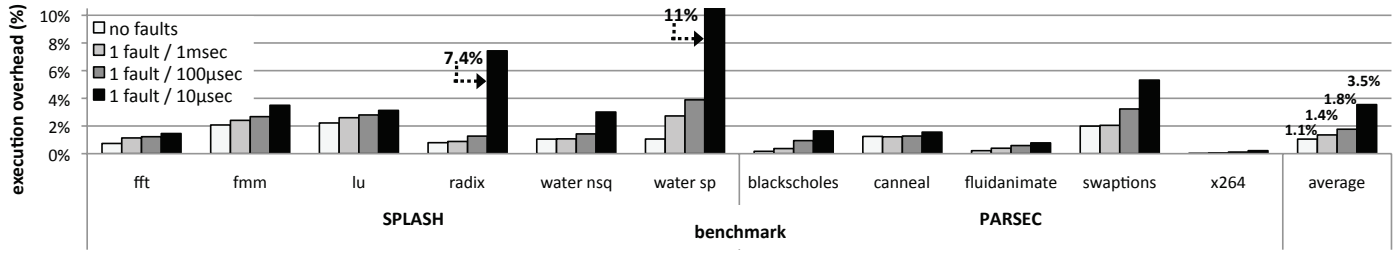


Figure 7: **Execution overhead** for the resilient directory-based protocol.

4.2 Evaluation Results

Directory-based protocol. We first evaluate the impact of our resilience transformations on transaction duration. Figure 6 depicts the duration of each transaction for the directory-based protocol discussed in Section 3. In x-axis, we separately consider the average duration of each baseline transaction in a lose-less NoC (*i.e.,* `B`) and its corresponding resilient transaction in a lossy NoC (*i.e.,* `R`), for three transaction types: transactions directly served by the L2 cache line that the directory is attached to (*i.e.,* `L2`), served by the L1 cache of a sharer (*i.e.,* `L1`), or served by the memory (*i.e.,* `MEM`). The duration of each transaction is further broken down to the time till data is received by the initiator (`data`), time till the invalidation acknowledgment is received by the initiator (`invalidation ack`), time till the directory is unblocked and the next transaction can be served (`DIR unblocked`), and time till the transaction is completed at the initiator with a done message from the directory (`done`). Note that only transactions which involve transferring data from another cache (*i.e.,* `L1`) incorporate invalidation handshaking, requiring the initiator to wait for the `invalidation ack` message.

We observe that the overall duration of each baseline transaction (`B`) increases when incorporating resilience (`R`). That is because the requestor has to wait for two additional messages to complete a transaction (`done` and `invalidation ack`). This translates to an indirect performance overhead: requests from other nodes will remain queued in the directory node longer. In addition, since we are modeling in-order processor cores, the initiator itself cannot generate succeeding memory requests for other memory addresses, until its current transaction has been completed. We note that in all benchmarks we simulate, the majority of requests (over 99%) is either served by the directory (L2 cache) or a sharer's L1 cache, thus this indirect performance overhead mostly depends on the increase of the `L2` bar (for benchmarks with read-only data) or `L1` bar (for benchmarks with exclusive data). Since the `L1` bar also accounts for the triple invalidation handshake, it results in higher (percent) increase in the transaction duration. Thus, as we show later in this section, benchmarks that are frequently requesting exclusive data incur a higher performance overhead.

Though the overall duration of transactions increases, the actual time till the initiator's request is served with data remains almost constant (less than 5% difference) in most benchmarks. There are a few exceptions to this trend, such as *radix* and *water spatial*. These benchmarks have the largest working sets and share mostly read-only data. Thus, they generate high read-request traffic (almost 2x compared to other benchmarks), which is served by the L2 cache (because the data copy is not modified by any sharer most of the times). The high traffic to the directory node is reflected by delayed data delivery in the baseline bar of Figure 6 (the `L2 B` bar is higher for *radix* and *water spatial*). In a lossy NoC, these benchmarks will suffer the most as faults increase, since re-transmissions saturate the network and result in higher network latencies. This is reflected in the corresponding resilient bar (`L2 R`) of Figure 6 as additional delay in data delivery (11% *radix* and 24% *water spatial*).

Figure 7 shows the overall execution overhead of the resilient directory based protocol, for increasingly lossy NoCs. For a fault-free NoC, our methodology only introduces a 1.1% performance overhead (on average). Benchmarks with exclusive requests are mostly affected by resilient transactions, as explained before: *fmm*, *lu*, and *swaptions* have the highest

execution overhead (around 2%) when there are no faults, since their exclusive requests are 2x, 1.5x, and 1.3x higher than other benchmarks. On the other hand, while the fault rate increases, the benchmarks with the largest working sets exponentially degrade due to network traffic saturating the NoC (7.4% overhead for *radix* and 11% overhead for *water spatial* at highest fault rate). The performance of benchmarks with low network demands (such as *x264*) remains unaffected by our modified transactions.

Broadcast-based protocol. The performance of broadcast-based protocols is very sensitive to network traffic load. Since each transaction generates a large number of messages (a request is broadcasted to all nodes, and all nodes respond to the requestor), it results in a traffic spike while packets compete for network resources. Consequently, when the number of requests increases, the network quickly saturates, imposing high network latencies to messages. Figure 8 depicts the average duration of baseline (`B`) and resilient (`R`) transactions for the broadcast-based protocol discussed in Section 3. Note the high data delivery time (over 500 cycles) compared to the directory-based protocol, due to increased network latencies during the broadcast.

In a lossy NoC, on top of the high volume of broadcast traffic, retransmission of lost messages further increases network contention and the duration of data delivery, especially in benchmarks with large working sets (28% for *radix* and 39% for *water spatial*, Figure 8). On the other hand, the duration of resilient handshaking (`invalidation ack` message and `done` message) is insignificant compared to the data delivery duration, since these messages are always sent after the initiator has been served (all nodes have responded and network resources have been deallocated). Thus, we expect the performance overhead of incorporating resilience into transactions to be negligible for small fault rates.

Figure 9 depicts the execution overhead of the resilient broadcast protocol, for increasingly lossy NoCs. For a fault-free NoC, our methodology introduces a negligible 0.5% performance overhead (on average), which exponentially increases as the rate of faults increases. We note that the fault-free performance overhead is even smaller than the directory-based scheme, since resilient messages account for a smaller portion of coherence transactions. On the other hand, when the fault rate increases, performance exponentially degrades due to network traffic saturating the already-congested NoC. Since the execution overhead mostly depends on network traffic, benchmarks generating a large number of requests are penalized the most (51% degradation for *radix*, 56% degradation for *water spatial*).

We note that such high fault rates are unrealistic and are intended to showcase that our methodology is viable for the highest possible fault rate where a system may recover with retransmissions. That is because for 1 fault per 10 microseconds (1 fault every 10,000 cycles for 1GHz clock) a fault is injected as often as twice the fault detection threshold (5,000 cycles), nearing the limit that we can handle (a system that loses packets at a higher rate than triggering retransmission of lost packets is not viable). For a fault per week, as discussed in Section 1, we expect the performance overhead of our methodology to be close to the fault-free scenario.

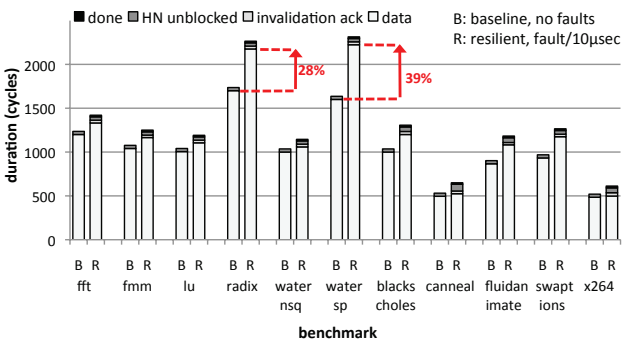


Figure 8: **Transaction duration breakdown**. baseline vs. resilient broadcast-based coherence protocol.

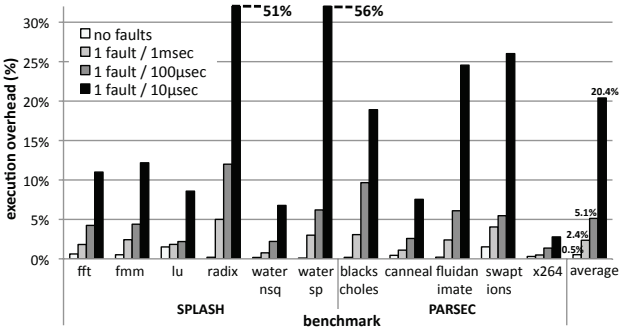


Figure 9: **Execution overhead** for resilient bcast protocol.

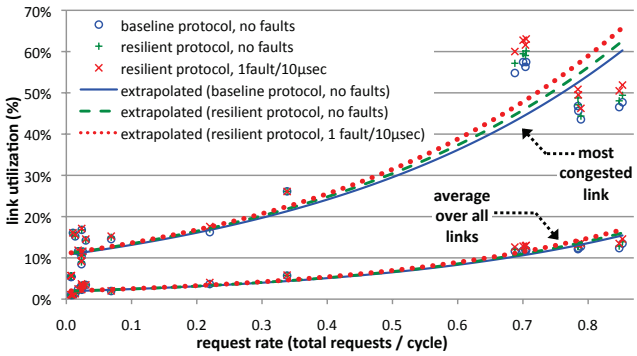


Figure 10: **Traffic overhead** due to resilient functionality.

4.3 Network Congestion

Figure 10 quantifies the increase in link utilization, as a result of the additional traffic introduced by resilient functionality, for all configurations and coherence protocols we have simulated. The figure depicts the link utilization of a resilient protocol (without faults and for 1 fault every 10 microseconds), as well as its corresponding baseline protocol. We observe that though for small request rates the effect of resilient transactions on link utilization is negligible, for higher request rates the utilization of the most congested link (bottleneck) increases up to 8%. This might be a relatively small overhead, but note that for high traffic loads the effect of congested links on performance degradation is non-linear, as demonstrated in Section 4.2.

5. DISCUSSION

5.1 Corner Cases

Redundant re-transmission. As discussed in Section 2.1, the recovery mechanism is triggered once the state of a cache line remains in a transient state beyond a timeout threshold. However, a transaction's duration cannot be bounded, due the non-deterministic nature of the underlying NoC. Thus, any timeout threshold can lead to redundant retransmissions. In other words, if the NoC excessively delays the delivery of a message (*e.g.,* due to an occasional traffic spike), the transaction initiator will retransmit its request to replay the transaction. If this occurs, the newly injected request will co-exist in the NoC with message(s) from a later phase of the same transaction. Intermingling new and stale messages has unpredictable consequences and may result in coherence violations. Hence, we want to eliminate all in-transit messages of a transaction before re-generating the initial request, so that the transaction safely "freezes" in a consistent state and is re-played without interferences.

To implement this, we incorporate a timeout counter into each in-transit packet. Intermediate routers then decrement this counter (once per cycle) and drop the corresponding packet once it exceeds the timeout threshold (counter underflow). The timeout counter of a packet is inherited from the packet that triggered its generation (unless it is the initial packet of a transaction, in which case the counter is set to the timeout threshold). Thus, at any point in time, all in-transit packets of a transaction piggyback the same timeout count (which matches the timeout counter of the initiator). Consequently, if the initiator's counter underflows, triggering the regeneration of the initial request, any in-transit packet of the ongoing transaction will be also dropped by its host router and thus will not interfere with the re-played transaction.

Note that though redundant retransmissions are possible, if the timeout threshold is set properly they occur very rarely. Our 5,000-cycle timeout threshold is an order of magnitude higher than the average transaction duration measured when the network is operating at its saturation point. Thus, the rate of messages regenerated due to timeouts (even under high traffic) is significantly lower than the overall rate of injected messages and does not affect traffic. A potential improvement here, which eliminates the need for profiling, is to adjust the timeout threshold at runtime, based on the number of transactions exceeding it over long time intervals.

Redundant messages after completing a transaction. This corner case occurs in transactions incorporating coherence messages with multiple destinations. Assume that the transaction is suspended due to a message to one of these destinations being lost. The initiator will resend its latest coherence message to replay the transaction, regenerating the message to multiple destinations. However, a number of responses from these destinations could have already been received by the initiating node (before suspension). Thus, when replaying the transaction, the initiating node does not require the destinations that have already responded to respond again in order to complete the transaction. Consequently, the transaction may complete while some of these (redundant) responses are still in transit.

Since the initiator transitions to a stable state, such redundant messages (expected in a transient state) can be safely ignored upon reception. However, it is possible that the initiator will be serving a succeeding transaction for the same address when a redundant message is received. In this case, if the ongoing transaction is also expecting a response from the node that generated the redundant message, the redundant message is erroneously recognized as a message of the current transaction. To prevent such corner cases, each initiator tags its requests with a transaction ID[4], while all messages generated due to a request also need to piggyback the request's transaction ID. Furthermore, each node in transient state should retain the transaction ID of the ongoing transaction and ignore messages tagged with different IDs.

Note that once a transaction that resulted in a redundant message has been completed, its initiator may start a new transaction with the same transaction ID (if it allocates to the same MSHR entry). However, in a NoC with point-to-point ordering, the redundant message will always reach its destination before any messages of the new transaction[5]. Since the destination of the redundant message cannot be tagged with the redundant message's transaction ID (no node other than its initiator may start a new transaction with this transaction ID), the redundant message is always dropped.

5.2 Hardware Overhead

Each outstanding transaction allocates an entry to the Miss Status Holding Register (MSHR), which retains the program counter, memory address, transient state, requesting node, flags, *etc.* In today's systems, a MSHR typically consists of 4 to 32 entries. Implementing our methodology requires the existing MSHR fields to be augmented and new fields to be incorporated, as detailed below:

• **Additional transient states**. The resilient protocols of Section 3 extend the requestor/sharer cache states from 9 to 17 (Table 1a, directory-based protocol) and from 12 to 22 (Table 1b, broadcast-based protocol). This translates to an additional state bit to encode cache states in both protocols (4 to 5). In contrast to stable states that are stored in the cache line, transient states are stored in the MSHR, thus the 1-bit overhead applies only to a few (up to 32) entries.
• **Transaction ID**. In order to prevent the corner case of redundant messages being received after the completion of a transaction, each outstanding transaction (corresponding to a MSHR entry) is tagged with a transaction ID. For 64

[4]The transaction ID consists of two numbers: the initiating node ID and a request ID. The initiating node ID is 6 bits for a 64-node system, while the request ID is a unique number, used to distinguish the request from other concurrent requests of the same core, thus it is constrained by the maximum number of outstanding requests (per core). We set the request ID equal to the index of the MSHR entry of the outstanding request. In today's systems, the number of outstanding requests typically ranges from 1 to 32 (requires 0 to 5bits), thus the transaction ID can be up to 11bits for a 64-node system. In our simulations, we assumed a 6-bit transaction ID (6+0, since we are modeling in-order cores).
[5]In a point-to-point ordered network (as in Table 2), deterministic routing and queuing arbiters ensure that two messages with the same source and destination can never be re-ordered in-network

in-order cores, that is a 6-bit (log64) overhead per entry.
• **Sender bitvector**. In order to enforce Property3, the requestor needs to maintain the "vector of message senders" while aggregating responses from all potential sharers (as described in Section 2.3). For a 64-node system, that is an overhead of 64 bits per MSHR entry.
• **Timeout Mechanism**. The initiator needs to maintain a counter that counts up to the timeout threshold for each outstanding transaction. Our 5,000-cycle timeout threshold imposes a (storage) overhead of 13 bits per MSHR entry.

The total overhead per MSHR entry is below 11 bytes (1 bit + 6 bits + 64 bits + 13 bits), which totals 352 bytes/node, assuming a 32-entry MSHR. In addition to the overhead of the MSHR, a 13-bit adder per buffered packet is required in each router, for decrementing the timeout counters (we assume 4 x 5 such half adders, since state-of-the-art routers typically maintain 2-8 virtual channels in each of their 5 ports -for a mesh topology-). In total, an overhead of 352 bytes/node and 20 x 16-bit half adders per router is a very low area overhead compared to the core gate count, which is in the order of hundred million gates.

5.3 Non-Blocking Decentralized Protocols

This section discusses how our methodology can be applied to token coherence [14]. In token coherence, the system associates a fixed number of tokens with each block. A processor may read a cache block when it holds at least one token, and write a cache block when holding all tokens. Requests are broadcasted to all nodes. Sharers respond to exclusive ownership requests with any tokens they hold, while the owner responds to shared ownership requests with data and one token. Conflicting transactions may result in races, which are tackled by persistent requests.

Token coherence adheres to Property1: the initiator remains transient during a transaction, since all transactions complete at the initiator upon reception of required tokens. Thus, no enforcement (no *done* message) is required. To adhere to Property2, in addition to unique data, all messages that hold tokens need to be protected with a triple handshake: upon reception of a message that holds token(s), the initiator should provide permission to delete the token(s), while the sender should acknowledge once deleting the token(s). No critical path delay is introduced when serving such a request, since the handshake occurs after the requestor has received the token(s). Finally, Property3 requires the requestor to maintain the vector of nodes that responded with token(s), to be able to identify duplicate tokens from the same node.

An interesting optimization here is that since regular requests do not always succeed in fetching the requested data or ownership, it is acceptable if they are lost (there is no need to retransmit a regular request, since a persistent request will be eventually triggered). On the other hand, persistent requests are essential for correctness and should always be regenerated after the timeout period. Note that nodes remember when persistent requests have been received, until they are explicitly cancelled by their sender. Consequently, a duplicate persistent request can be easily identified without retaining any additional state.

6. RELATED WORK

Resilience at network-level. Resilient NoCs explore novel designs to allow correct operation in the face of network faults: BulletProof [9] and Vicis [12] can detect and tolerate the loss of many network components due to hard faults. In addition, a number of resilient routing algorithms [4, 18] have been suggested to re-route network packets around faulty links. In the case of faulty links causing network nodes to become disconnected, DRAIN [10] uses emergency links to transfer architectural state and cached data to nearby connected caches. However, no resilient solution can guarantee 100% fault tolerance in the network layer. Resilient routers utilize redundant resources for resiliency, which are finite: for 100 permanent network faults, half of the routers are not functional in Vicis, while BulletProof's reliability approaches zero. In addition, upon a link failure, though resilient routing algorithms will re-route traffic around the faulty link, they cannot recover the packets that are stranded across multiple routers. Finally, emergency links that recover data from disconnected nodes can themselves fail. If a NoC solution does not succeed in masking a network fault, this fault will be exposed to upper layers, causing lost or corrupted coherence messages, and resulting in coherence violations.

Resilience with checkpointing. Checkpointing requires pro-actively replicating a safe state to roll-back upon the occurrence of a fault. The granularity of checkpointing is the critical factor determining a solution's implementation overhead, thus some solutions (SafetyNet [20]) opt for frequent and lightweight checkpointing, while others (ReVive [17]) suggest infrequent but complex checkpointing. SafetyNet's checkpointing mechanism [20] does not interfere with program execution, since it silently logs every change in cache lines and coherence state in dedicated cache structures called Checkpoint Log Buffers (CLBs). However, logging in CLBs consumes expensive cache space (storage overhead is 12%-25%). To avoid consuming cache space, ReVive [17] opts for infrequent (every 100ms) but more intrusive checkpointing, where processors are interrupted and flush their caches to establish a global checkpoint. Establishing a system-wide coherent checkpoint is a complex procedure and involves a 6.3% performance overhead during error-free execution [17].

Resilient coherence protocols. Recently, Fernández *et al.* modified a directory coherence protocol for CMPs (*FT-DirCMP* [16]) and token coherence protocol for CMPs (*FT-TokenCMP* [11]) to tolerate the loss of their coherence messages. The recovery mechanism of these protocols is also based on retransmissions, but each node participating in a transaction utilizes its own protocol-specific timeout types, generates protocol-specific synchronization messages, and retains protocol-specific information for recovery. Some of these synchronization messages are not present during error-free execution. For example, in *FTDirCMP*, if the directory remains blocked for too long, this triggers a "lost unblock" timeout, which pings the requestor to retransmit its unlock message (*UnlockPing*). Since such synchronization messages are not present in regular transactions, additional network resources need to be allocated (*e.g., FTDirCMP* requires two additional virtual channels to be deadlock-free).

As these approaches utilize a set of protocol-specific timeouts, each invoking a different recovery procedure, it prevents them from being applied to other protocols. Our methodology, on the other hand, utilizes a single timeout counter for each transaction, always at the node which initiated the transaction. Upon a fault, the transaction initiator replays the suspended transaction (from its current state to completion) identically as the fault-free scenario, without introducing any protocol-specific synchronization messages. Consequently, we believe that our methodology is generic and applicable to a wide range of coherence protocols.

7. CONCLUSIONS

We have presented a systematic methodology to incorporate resiliency into a wide range of coherence protocols. Our methodology modifies the protocol states, transitions, and messages, to adhere to three properties. These properties guarantee that any transaction that has been suspended due to the loss of coherence messages will eventually complete. We have also presented two case studies, demonstrating how these properties are enforced in a directory-based and a broadcast-based coherence protocol. Our experimental results indicated negligible hardware overhead and an execution overhead of 0.8% during fault-free operation, which increases to 1.9% at an aggressive fault rate of one fault per millisecond. Thus, we conclude that our methodology is a parsimonious solution that enables reliable functionality on unreliable silicon for future CMP architectures.

ACKNOWLEDGMENTS

The authors would like to thank Edya Mozes for giving them a tutorial on sketching a proof for the proposed methodology, and Tushar Krishna for suggesting a way to tackle redundant retransmissions.

References

[1] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-Network Snoop Ordering (INSO): snoopy coherence on unordered networks," in *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, 2009.

[2] A. Ahmed, P. Conway, B. Hughes, and F. Weber, "AMD opteron shared memory MP systems," in *Proceedings of the 14th HotChips Symposium*, 2002.

[3] K. Aisopos, C.-H. O. Chen, and L.-S. Peh, "Enabling system-level modeling of variation-induced faults in networks-on-chip," in *Proceedings of the 48th Design Automation Conference*, 2011.

[4] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco, "ARIADNE: Agnostic Reconfiguration In A Disconnected Network Environment," in *Proceedings of the International conference on Parallel Architectures and Compilation Techniques*, 2011.

[5] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[6] R. Bauman, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, 2005.

[7] D. Bertozzi, L. Benini, and G. De Micheli, "Error control schemes for on-chip communication links: the energy-reliability tradeoff," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, 2005.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the International conference on Parallel Architectures and Compilation Techniques*, 2008.

[9] K. Constantinides, S. Plaza, J. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky, "Bulletproof: a defect-tolerant CMP switch architecture," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

[10] A. DeOrio, K. Aisopos, V. Bertacco, and L.-S. Peh, "DRAIN: Distributed Recovery Architecture for Inaccessible Nodes in Multi-Core Chips," in *Proceedings of Design Automation Conference*, 2011.

[11] R. Fernández-Pascual, J. M. García, M. E. Acacio, and J. Duato, "A low overhead fault tolerant coherence protocol for CMP architectures," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[12] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester, "Vicis: a reliable network for unreliable silicon," in *Proceedings of the Design Automation Conference*, 2009.

[13] J. Graham, "Soft errors a problem as SRAM geometries shrink," *EE Times*, 2002.

[14] M. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," in *Proceedings of the 30th annual International Symposium on Computer Architecture*, 2003.

[15] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) toolset," *Special Interest Group on Computer Architecture*, vol. 33, no. 4, 2005.

[16] R. F. Pascual, J. M. García, M. E. Acacio, and J. Duato, "A fault-tolerant directory-based cache coherence protocol for CMP architectures," in *Proceedings of the 38th International Conference on Dependable Systems and Networks*, 2008.

[17] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proceedings of the 29th International Symposium on Computer architecture*, 2002.

[18] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide, "Immunet: A cheap and robust fault-tolerant packet routing mechanism," *Special Interest Group on Computer Architecture*, vol. 32, no. 2, 2004.

[19] S. Woo et al, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the International Symposium on Computer Architecture*, 1995.

[20] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[21] L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server g5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5.6, 1999.