2020 26th International Symposium on High-Performance Computer Architecture

DRAIN: Deadlock Removal for Arbitrary Irregular Networks

Mayank Parasar[1], **Hossein Farrokhbakht**[2], **Natalie Enright Jerger**[2], **Paul V. Gratz**[3], **Tushar Krishna**[1], **Joshua San Miguel**[4]
[1]Georgia Institute of Technology, [2]University of Toronto, [3]Texas A & M, [4]University of Wisconsin-Madison
Email: [1]mparasar3@gatech.edu, [2]h.farrokhbakht@mail.utoronto.ca, [2]enright@ece.utoronto.ca, [3]pgratz@tamu.edu, [1]tushar@ece.gatech.edu, [4]jsanmiguel@wisc.edu

***Abstract*—Correctness is a first-order concern in the design of computer systems. For multiprocessors, a primary correctness concern is the deadlock-free operation of the network and its coherence protocol; furthermore, we must guarantee the continued correctness of the network in the face of increasing faults. Designing for deadlock freedom is expensive. Prior solutions either sacrifice performance or power efficiency to proactively avoid deadlocks or impose high hardware complexity to reactively resolve deadlocks as they occur. However, the precise confluence of events that lead to deadlocks is so rare that minimal resources and time should be spent to ensure deadlock freedom. To that end, we propose DRAIN, a *subactive* approach to remove *potential* deadlocks without needing to explicitly detect or avoid them. We simply let deadlocks happen and periodically *drain* (i.e., force the movement of) packets in the network that *may* be involved in a cyclic dependency. As deadlocks are a rare occurrence, draining can be performed infrequently and at low cost. Unlike prior solutions, DRAIN eliminates not only routing-level but also protocol-level deadlocks without the need for expensive virtual networks. DRAIN dramatically simplifies deadlock freedom for irregular topologies and networks that are prone to wear-related faults. Our evaluations show that on an average, DRAIN can save 26.73% packet latency compared to proactive deadlock-freedom schemes in the presence of faults while saving 77.6% power compared to reactive schemes.**

I. Introduction

Correctness is of paramount concern in interconnection networks. *Deadlock freedom* is a cornerstone of correctness.

A. The Problem: Deadlocks

A deadlock occurs when there is a cyclic resource dependence in the network. Designing for deadlock freedom is costly in terms of hardware resources (e.g., large buffering requirements), design-time overheads (e.g., design verification) and performance-limiting constraints (e.g., restrictive routing). Furthermore, as process technology descends into the deep sub-micro realm, continued (albeit degraded) service must be provided by the network in the face of accumulating hard faults over the lifetime of the chip [1]. The network's operation must remain deadlock-free as its network topology evolves over time due to randomly occurring hardware faults. There are two types of deadlock that can arise.

Routing-Level Deadlocks. A *routing-level deadlock* occurs when multiple packets hold on to network buffers

The first two authors contributed equally to this work.

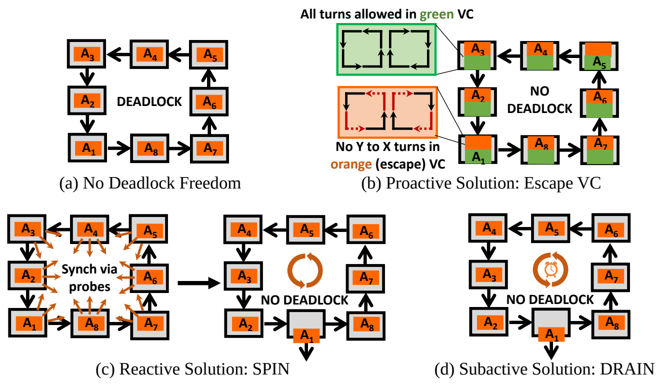

All turns allowed in green VC
DEADLOCK
NO DEADLOCK
No Y to X turns in orange (escape) VC
Synch via probes
NO DEADLOCK
NO DEADLOCK
(a) No Deadlock Freedom
(b) Proactive Solution: Escape VC
(c) Reactive Solution: SPIN
(d) Subactive Solution: DRAIN

Figure 1: Routing-level deadlock and solutions.

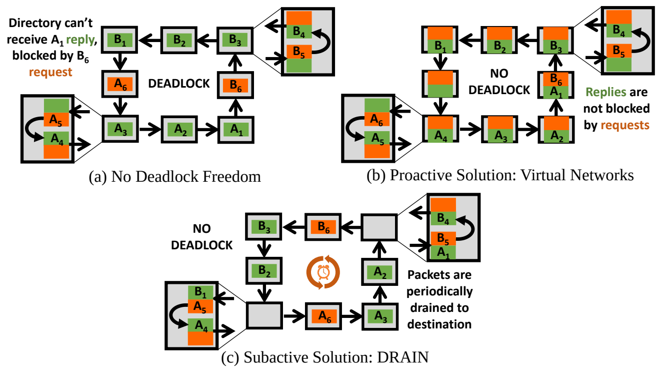

Directory can't receive A1 reply, blocked by B6 request
DEADLOCK
NO DEADLOCK
Replies are not blocked by requests
NO DEADLOCK
Packets are periodically drained to destination
(a) No Deadlock Freedom
(b) Proactive Solution: Virtual Networks
(c) Subactive Solution: DRAIN

Figure 2: Protocol-level deadlock and solutions.

while waiting for other occupied buffers to become available, forcing packets to stall forever. This is shown in Figure 1a, where a packet holding an upstream buffer can only give up its buffer once the downstream buffer becomes free.

Protocol-Level Deadlocks. A *protocol-level deadlock* occurs when packets of different message classes (e.g., requests, responses) are forced to wait for each other. Multiprocessors employ cache coherence protocols that require atomic transactions. However, when implemented on an arbitrary interconnect, these transactions are broken into multiple non-atomic packets that may interleave with those of other transactions. Since they all interact on top of the same physical substrate, deadlocks may occur among them. This is shown in Figure 2a where the packets of different message classes must wait for each other. Each request (orange) generates a response (green) at the directory, but these responses are blocked by other requests.

B. *Prior Solutions*

Some prior solutions to deadlock freedom are listed in Table I. The common approach is *proactive*, where deadlocks are avoided by design. Recent work has also proposed *reactive* solutions that detect and recover from deadlocks at run-time.

Routing-Level Deadlocks: Proactive Solutions. Common proactive mechanisms for routing-level deadlock freedom either impose routing turn restrictions or add escape virtual channels (VCs), which themselves are deadlock-free often via turn restrictions [2] as shown in Figure 1b. By construction, turn restrictions prevent any cyclic resource dependencies from being present in the network. There are several key problems with this approach: 1) they are limited to static, regular topologies, 2) they leave significant performance on the table, and 3) they require expensive buffers in the form of extra VCs.

Routing-Level Deadlocks: Reactive Solutions. Reactive solutions detect a routing-level deadlock and then find a way to recover from it. For example, SPIN [5] detects potential resource cycles and *spins* the messages involved in that cycle, as shown in Figure 1c. Reactive approaches are promising for networks with increasing dynamic irregularity. Unfortunately, reactive approaches like SPIN are limited in scalability since they require significant hardware complexity for online detection of the deadlocks, followed by coordinating a recovery mechanism (e.g., turning on additional buffers [6], [7] or global coordination of multiple routers to induce a spin [5]).

Protocol-Level Deadlocks: Proactive Solutions. Shared-memory systems that support cache coherence typically employ proactive approaches to avoid protocol-level deadlocks. In these systems, the processing of a request in the directory (e.g., write request) can require the creation of dependent packets to be injected into the network (e.g., invalidation requests). This dependency chain through the directory can break the proactive turn-based routing restrictions used to prevent routing-level deadlock, as shown in Figure 2a. To address this, cache-coherent interconnects need to employ per-message-class *virtual networks*. Virtual networks are orthogonal sets of VCs that may only be used by a particular message class. With distinct VCs per message class, it is impossible to form a cyclic dependence between different protocol requests and responses, as shown in Figure 2b. This comes at a high overhead, in terms of the amount of buffering required to implement virtual networks, with each virtual network requiring its own set of distinct buffers. Our experiments will show that VC buffers are the dominant component of area and power usage in the interconnect.

Protocol-Level Deadlocks: Reactive Solutions. There are no existing reactive solutions for resolving protocol-level deadlocks. The dependency chains of protocol messages extend outside the network and into the caches and directories. These deadlocks cannot be detected nor coordinated within the network.

C. *DRAIN: A Subactive Solution*

Designing routing and protocol deadlock freedom in the face of increasing irregularity (as failures increase over time) is challenging. We exploit the insight that deadlocks *very rarely* occur in practice. Deadlocks require a specific confluence of packet routes and timings to actually emerge in a given network. Given the rarity of deadlocks, should designers spend precious runtime power to mitigate this *remote possibility*? We believe the answer is *no*.

We propose *Deadlock Removal for Arbitrary Irregular Networks (DRAIN)*, a solution that periodically flushes (or *drains*) network resources to recover from *potential deadlocks* in the network. We introduce the concept of *subactive* deadlock freedom: DRAIN neither avoids nor reacts to deadlocks but rather lets them happen and eventually cleans them up, as shown in Fig. 1d and 2c. DRAIN guarantees deadlock-free operation without adding any performance restrictions nor expensive buffers, unlike in proactive solutions. DRAIN is the first to employ *oblivious* deadlock removal; it requires no complex detection nor recovery mechanisms, unlike in reactive solutions. This property makes DRAIN unique in its ability to resolve both routing and protocol-level deadlocks simultaneously without the need for virtual networks.

Our work makes the following primary contributions:

- Introduces the first subactive solution to both routing-level and protocol-level deadlock freedom.
- Proposes DRAIN, a network architecture that neither avoids nor reacts to deadlocks, ensuring deadlock-free operation with low hardware complexity.
- Develops an offline algorithm for deciding when and what to drain for arbitrary irregular networks that is robust to link failures.
- Evaluates DRAIN on a wide range of applications, demonstrating the best of both worlds in terms of performance and power compared to prior proactive and reactive solutions.

II. Motivation

First, we illustrate that deadlocks are sufficiently rare, justifying the need for an ultra-low-cost, low-overhead solution to achieve correctness without sacrificing significant area, power or performance. Then, we highlight key advantages and disadvantages of prior techniques. Finally, we motivate the growing importance of designing interconnects with fault tolerance in mind.

A. *Observation: Deadlocks are Rare*

To demonstrate the rare and unlikely occurrence of deadlock, we look at both application workloads and synthetic traffic. In Figure 3, links are randomly removed from an

	Type of Solution	High Performance	Low Area and Power	Low Hardware Complexity	*Resolves Routing-Level Deadlock*	*Resolves Protocol-Level Deadlock*
Turn Restrictions [2]	Proactive	✗	✓	✓	✓	✗
Escape VCs [3]	Proactive	✓	✗	✓	✓	✗
Virtual Networks [4]	Proactive	✓	✗	✗	✗	✓
SPIN [5]	Reactive	✓	✓	✗	✓	✗
DRAIN	*Subactive*	✓	✓	✓	✓	✓

Table I: Comparison of solutions for routing-level and protocol-level deadlock freedom.

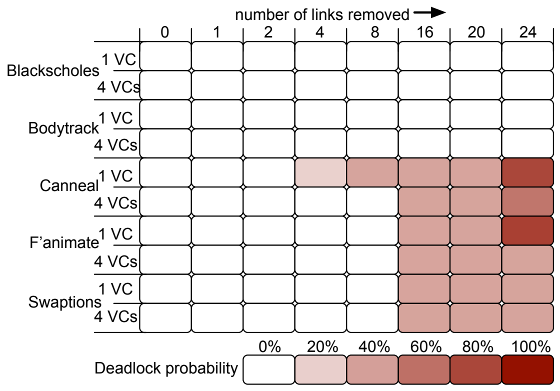


Figure 3: Likelihood of deadlocks for PARSEC workloads as links are removed.

8×8 mesh to simulate faulty, irregular topologies.[1] All nodes remain connected to the network when links are removed. Here the routing algorithm is fully adaptive and not deadlock-free. Each PARSEC [8] workload is run five times with 1 VC and 4 VCs per virtual network. The color scale corresponds to the percentage of runs that result in a deadlock. No deadlocks are observed for the fully functional case (i.e., 0 links removed). Note that because the routing algorithm is not deadlock-free, even with no links removed, deadlocks are possible in this network. Only upon removing four links do we begin to encounter deadlocks for `canneal`, which has the highest injection rate of these five workloads. A higher injection rate implies that there may be enough packets in the network at any given moment for a deadlock to emerge. As more links are removed, deadlocks become more common across several of the workloads; removing more links increases the likelihood that packets can coincidentally form a cycle on the remaining links. Note that the presence of additional VCs may delay the onset of deadlock but is not sufficient to provide deadlock freedom.

Previous work [7] has also shown that faulty topologies are more deadlock-prone than fault-free topologies. This is because faulty topologies limit the path diversity, resulting in higher hop counts in the network. Thus packets stay longer in the network and have a higher chance of being involved in a deadlock cycle. This is why deadlocks can occur at a lower injection rate in faulty topologies.

Takeaway. Even in the absence of any explicit deadlock avoidance or prevention mechanism, the occurrence of deadlocks is quite rare. Thus, we aim to follow the adage of "make the common case fast": deadlocks are uncommon but still need to be handled correctly. We do not want to devote significant hardware resources to such an uncommon case nor do we want to cripple the performance of the common case of deadlock-free operation by imposing routing restrictions on all network packets. We use this insight to guide us towards a new design: we eventually resolve deadlocks, should they occur, but we achieve this at very low cost and design complexity while maintaining flexibility.

[1]See Section IV for our detailed methodology and workloads.

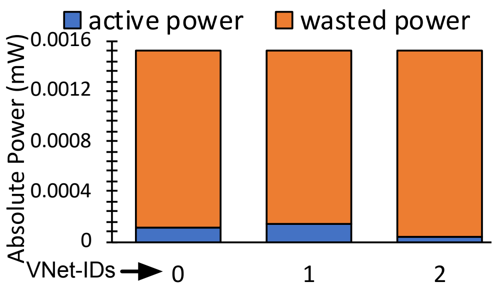


Figure 4: Wasted power in virtual networks for MESI cache coherence protocol

B. Observation: Virtual Networks are Costly

Figure 4 shows the total power consumed by virtual networks, the de facto solution to protocol-level deadlock freedom. The number of virtual networks depends on the cache coherence protocol of the system. In this figure, active power refers to power expended transfer packets through the virtual network, while wasted power refers to power expended even though no packet is in flight. We observe the vast majority of power consumption in virtual networks is wasted.

Takeaway. Despite the wasted power, virtual networks are still needed; otherwise correct execution is not guaranteed due to protocol-level deadlock. We strive for a solution that is capable of simultaneously resolving both protocol- and routing-level deadlocks.

C. Prior Work: Proactive and Reactive

Here, we examine the characteristics of existing schemes for routing and protocol-level deadlock freedom and make a case for a new class of subactive techniques. Table I breaks down the major categories of existing solutions: turn restrictions [2], escape VCs [3], virtual networks [4] and SPIN [5].

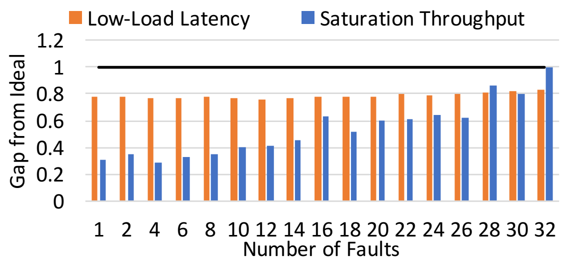


Figure 5: Gap between low-load latency and saturation throughput for up*/down* routing and ideal (shown as a black line at 1).

Turn Restrictions. The simplest way to avoid routing-level deadlock is to place turn restrictions in the routing algorithm so that cyclic resource dependences can never form. This *proactive* deadlock avoidance scheme prevents deadlocks from ever emerging in the network. There are several limitations to this approach. First, it suffers from lower performance. Turn restrictions reduce path diversity, which reduces the achievable saturation throughput. A particular form of turn restriction-based deadlock avoidance often used for routing in irregular and faulty networks is up*/down* [9] routing. Here, all routers in the network are arbitrarily numbered such that each node has a unique number. During operation, turns are only allowed from lower numbered routers to higher numbered routers (or vice versa). Figure 5 shows the performance gap between up*/down* routing and ideal deadlock-free fully adaptive routing.[2] The presence of non-minimal routes in up*/down* increases low-load latency for all fault rates: in the worse case, there is a 24% gap between the ideal latency and the achieved latency with a 22% gap on average. At low fault rates, up*/down* only achieves 19% of the possible saturation throughput. As the number of faults increase, the ideal and up*/down* configurations converge on throughput. As faults increase, the network has substantially less available bandwidth, which hurts all approaches uniformly.

Escape Virtual Channels. Performance loss can be mitigated by increasing the number of VCs in the network. A common approach is to designate one VC as an escape VC. A packet that enters an escape VC is never allowed to enter a non-escape VC thereafter. Turn restrictions are only applied to the escape VC; as long as every packet has a chance of acquiring the escape VC, routing-level deadlock is avoided. This yields better performance and routing flexibility (which in turn can decrease the likelihood of deadlocks [10]) by allowing fully adaptive routing on non-escape VCs but costs additional area and power since VCs are expensive to implement in networks-on-chip (NoCs). Furthermore, this is an expensive solution that must be made *proactively* at design time; extra VCs must be provisioned even if the overall network load is low and the chance of deadlock is rare. As a result, prior work has shown that the escape VC tends to be highly underutilized under common operating conditions [5]. Performance also suffers because fully adaptive routing can only be paired with conservative VC allocation, which reduces throughput [11].

[2]This idealized case quickly detects and recovers from deadlock without imposing any overheads on the network.

Virtual Networks. VC buffers can also be used to construct effectively orthogonal virtual networks which share the datapath of a given router but have orthogonal storage [4]. Shared-memory systems that support cache coherence typically employ virtual networks to avoid protocol-level deadlocks, as discussed in Section I. In these systems, each packet class is only allowed to occupy VCs within its assigned virtual network. While this orthogonality prevents protocol deadlocks, it comes at the high cost of multiplying the per-router VC storage times the number of virtual networks. Further, since different packet classes typically have differing amounts of traffic, this can lead to highly divergent VC utilization between the virtual networks.

Deadlock Detection and Recovery. An alternative approach to the above techniques is to implement a deadlock detection and recovery mechanism. One example is SPIN [5], which requires neither routing restrictions nor extra buffers to ensure routing-level deadlock freedom. Instead, deadlocks are detected via probes that are sent out at specific intervals. Once a deadlock has been identified, the nodes in the deadlock make a coordinated movement to resolve the deadlock. These types of techniques are attractive as they do not require additional VCs nor suffer performance loss in the nominal operating case. However, they suffer from additional complexity in terms of the detection and resolution of the deadlock. This type of *reactive* approach can have significant added complexity that limits the scalability of these designs. We discuss additional recovery-based techniques in Section VII.

Our Subactive Approach. We propose a new class of techniques that we characterize as *subactive* (Table I). As we will show, this allows for 1) high performance since no restrictions are placed on routing in the nominal operating mode; 2) low complexity since there is no need for explicit detection nor global coordination; and 3) significantly reduced area and power since multiple virtual networks are no longer required for protocol-level deadlock freedom.

D. Use Case: Deadlock-Free Faults

As process technologies continue to shrink into the deep sub-micron domain, the breakdown of Dennard Scaling has meant that on-chip current densities are increasing as device density increases. Thus individual devices and wires are exposed to higher operating temperatures and currents, both of which are known to accelerate their eventual breakdown by one of a number of wear-out mechanisms, including Bias Temperature Instability [12], Time-Dependant Dielectric Breakdown [13], Hot-Carrier Injection [14] and

Electromigration [15]. In each case, heat and/or current accelerate wear, increasing the odds of individual device failure. When put together with increasing device and wire density, the odds of component failures on-chip are rising dramatically with each process generation. Traditional techniques, such as adding extra timing guard bands and wire thickening in vulnerable locations are no longer sufficient to address this growing problem; thus architectural techniques to deal with wear-out failures during product lifetimes must be developed [1].

These eventual component failures imply the expectation that individual cores and other components may fail during the lifetime of the product. An individual core or other redundant component failure can be dealt with via detection hardware and associated fail-over software, allowing continued operation at lower capacity [16]. However, failures of the interconnect components (e.g., links, routers) can be more challenging. In NoCs, applying routing restrictions is the most common deadlock-freedom mechanism; yet it requires static and regular network topologies. Thus as links and routers fail, these routing restrictions may be violated, leading to potential deadlocks. Similarly, for topologies that are irregular [17] or random [18] by design, traditional deadlock avoidance techniques that rely on network regularity do not work.

Existing mechanisms to handle router and link wear-out failure [19], [20], [21], [22] require significant extra hardware to support runtime routing reconfiguration and often create strong network hot spots due to the need to ensure deadlock avoidance in the newly irregular network. A scheme such as DRAIN that allows unrestricted adaptive routing in arbitrary, irregular and dynamically changing networks would be a significant improvement on the state-of-the-art for fault-tolerant interconnection network design.

III. DRAIN

This section describes the design of our DRAIN architecture, from theory to implementation.

Theory. The premise behind DRAIN is that routing-level and protocol-level deadlocks fundamentally need neither be detected nor strictly avoided; they just need to be *subactively* removed. Deadlocks are so rare that the cheapest solution is to simply let them happen (if ever) and periodically and obliviously *drain* resources in the network that *may* be in a deadlock. Even if no deadlock exists, correctness is maintained; draining would merely incur an infrequent and minimal performance overhead.

Analogy. Consider an analogy to street sweepers: periodically, sweepers will traverse the city streets along a predefined route to clear out leaves and other debris. This sweep is executed regardless of whether it is needed; fall may come late and the sweepers do their cleaning prior to the majority of leaves falling or fall may come early and the sweepers effectively clear away accumulated debris. DRAIN operates similarly, periodic draining of packets will occur regardless of need (ie., deadlock); ideally, these drains will coincide with the occurrence of an actual deadlock. Should the draining occur just prior to a deadlock, that deadlock will persist until the next scheduled draining.

Implementation Overview. Figures 1d and 2c show a high-level overview of DRAIN. Draining can be performed by any operation that 1) can eliminate a deadlock among packets if one exists, and 2) does not hurt correctness if no deadlock exists. In DRAIN, we employ a very low-cost draining mechanism that is inspired by the forced movement introduced in SPIN [5] yet avoids its complexity. Conventionally packets in a deadlock cannot move forward until they have observed that the packets in front of them have moved forward. In DRAIN, we periodically *drain* the network: we force all packets to move in a predetermined cyclic path (*drain path*). This unhinges any deadlocked packets and gives them the opportunity to eventually exit the deadlock cycle—by either making a turn or ejecting from the network—thus eliminating the deadlock. DRAIN does not need to globally coordinate the deadlocked packets via complex probe messages at runtime (as SPIN does, Figure 1c). Instead, DRAIN determines, offline, a cyclic path through the entire network that covers all links. Then routers *locally* drain the packets in their VC buffers along this path, at preset periods at runtime (*drain windows*), even if no deadlock exists. Since both when to drain (i.e., drain window) and where to drain (i.e., drain path) are statically predetermined, DRAIN incurs very low hardware complexity. Though draining may misroute any packets currently in the VCs, misrouting does not hurt correctness. As our results demonstrate, misroutes are sufficiently infrequent that they have no significant impact on performance.

Routing-Level and Protocol-Level Deadlocks. DRAIN guarantees both routing-level and protocol-level deadlock freedom *simultaneously*. They share the same hardware implementation. If a cyclic dependence exists in the network, regardless of whether the packets belong to the same message class or different classes, the dependence is guaranteed to break eventually via periodically forcing packets to move.

Before describing the details of the architecture, we first state our baseline assumptions (Section III-A). We then describe the two key components of DRAIN:

- An offline algorithm that finds a drain path composed of all links in the network (Section III-B).
- A low-cost router architecture that periodically drains packets along this path (Section III-C).

We conclude this section with the necessary proofs (Section III-D) and a walk-through example (Section III-E).

A. *Assumptions and Definitions*

We make three assumptions about the topology, which are commonly found in networks:

1) All routers are reachable by all other routers, even in the presence of faults. In other words, the network is

connected, and all source-destination pairs are possible. This is a typical assumption since disconnected topologies serve little value in real-world multiprocessors.

2) All routers are connected via bidirectional links (i.e., two opposing unidirectional links). We find that this is true for most topologies. If a single unidirectional link becomes faulty, we assume that both opposing links (and their VC buffers) are disabled.
3) All turns (including U-turns) are possible in every router (i.e., every input port can route to every output port). Networks that employ adaptive routing often already provide this capability. Allowing for U-turns only requires modest changes to the allocators and crossbars.

Any topology that holds all the above assumptions is guaranteed to have at least one cycle (i.e., drain path) that spans all links. Since the network is connected, it is always possible to construct a spanning tree that covers all bidirectional links in the topology. Since each bidirectional link allows for an implicit turn to itself via a U-turn, the spanning tree is equivalent to a unidirectional cycle that covers all links and all routers. This cycle is equivalent to the path taken by a depth-first traversal through the spanning tree.

Definitions. We list key terminology:

- *Drain:* Force all packets currently in the network to take a specific turn, regardless of whether or not they are in a deadlock.
- *Drain Path:* A cycle that covers all links in the network, specifying where each drained packet must turn.
- *Pre-Drain:* Before draining, let any packets currently traversing a link complete.
- *Drain Window:* The predetermined time period reserved for all routers to perform draining.
- *Pre-Drain Window:* The predetermined time period reserved for all routers to perform pre-draining. This immediately precedes the drain window.
- *Epoch:* The time between drain windows.
- *Full Drain:* Allows all packets in the network to traverse the whole topology and eject out when they visit their destination router during traversal.

Protocol-Level Deadlocks. For protocol-level deadlocks, we make two additional assumptions. First, each router's injection and ejection ports use separate queues per message class. This is typical in modern shared-memory multiprocessors, which have dedicated queues for different coherence messages outside the network. Second, we assume that it is not possible for packets of a single message class to occupy all buffers in the network, leaving no space for other message classes. This is typical, since miss status handling registers (MSHRs) are often few enough relative to the amount of VCs in the network, bounding the number of packets per message class. With these assumptions, if the network eliminates protocol-level deadlocks within it, then the multiprocessor system is guaranteed to be free of protocol-level deadlocks.

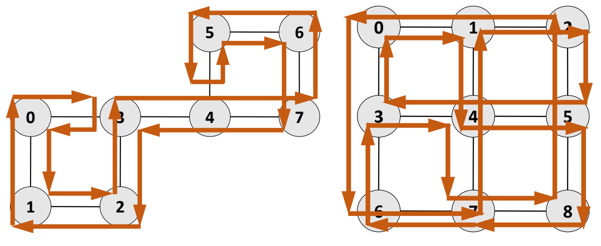


Figure 6: Sample outputs of our offline algorithm for (a) an irregular topology and (b) a regular topology. Each arrow represents a unidirectional link in the drain path.

Draining Only Escape VCs. In networks with multiple VCs per port, we perform draining only on one VC and designate it as an escape VC. Every packet has an opportunity to enter the escape VC, and if it does, it is no longer allowed to move to any non-escape VCs. In contrast to typical escape VCs, our escape VC has no turn restrictions placed on it. Other VCs do not need to be deadlock-free since DRAIN ensures that the escape VC is deadlock-free.

B. *Offline Algorithm*

DRAIN ensures deadlock freedom by conservatively, periodically draining all escape VC buffers in the network. Given any arbitrary network topology, the goal of our offline algorithm is to find the *drain path*: a single cycle composed of all unidirectional links in the network. This can be done offline and rerun whenever a link becomes faulty. At runtime, during each drain window, all packets in the escape VCs are circulated along this path in unison, for some number of hops. Fig. 6 shows example outputs of our algorithm for an irregular topology and a regular topology. In the figures, each edge represents two opposing unidirectional links. All unidirectional links in these topologies are covered by the drain path found by our algorithm. During each drain window, any packet currently in an escape VC buffer is forced to make a turn following the path.

DRAIN supports irregular network topologies in the presence of faulty links. The input topology is represented as a dependency graph G where each node is a unidirectional link in the topology, and each directed edge is a turn between two unidirectional links. We denote the set of all unidirectional links as L. A cycle in G is defined as a sequence of links $l = \{l_1, ..., l_n\}$ where l_i is connected to l_{i+1} via a turn, and l_n is connected to l_1 via a turn. Only *elementary* cycles need to be considered: an elementary cycle visits each link at most once. Non-elementary cycles are impossible since a link can only transfer at most one packet at a time. Our algorithm's goal is to find a cycle C where $l = L$. We are guaranteed to find at least one such cycle given our baseline assumptions in Section III-A. Our implementation builds upon the cycle-finding method proposed by Hawick and James [23], with complexity $O((V+E) \times (C+1))$, where

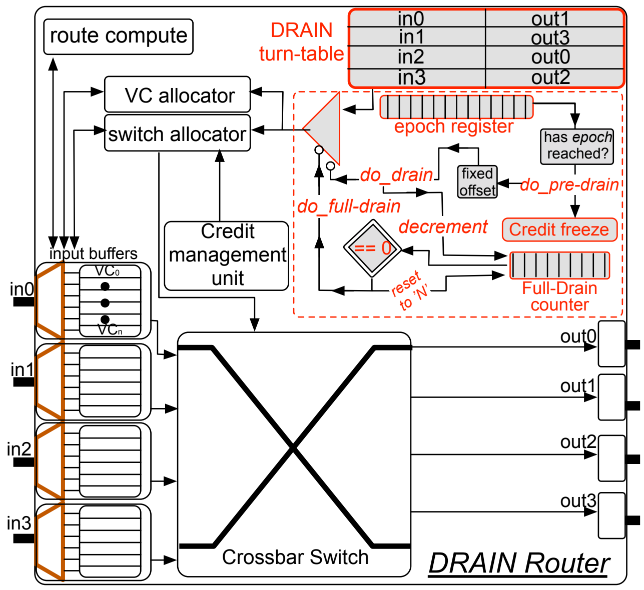


Figure 7: DRAIN router microarchitecture. The red modules are unique to DRAIN.

V, E and C are the number of vertices, edges and cycles in G, respectively. This method employs a recursive tree search with efficient structures for tracking vertex adjacency lists. We augment this to terminate early as soon as a single cycle is found that covers all links in L. Since the algorithm only needs to be computed whenever the topology changes (i.e., when a fault occurs, upon a system reboot), we expect its runtime to be negligible to overall system performance.

C. Router Microarchitecture

Three changes are necessary to the router microarchitecture, highlighted in grey in Figure 7, each described in the following sections:

1) The *epoch register* for determining when it is time to pre-drain and drain.
2) The *credit freeze* for preventing new packets from allocating a VC during each pre-drain.
3) The *turn-table* for determining where each input port turns during each drain.

1) When to Drain and Pre-Drain: We denote the *pre-drain window* and *drain window* as the time periods (i.e., clock cycles) reserved for pre-draining and draining, respectively. Every drain window is preceded by a short pre-drain window. When to pre-drain and drain is established ahead of time and known by all routers in the network; these values are loaded at boot time and require no subsequent global coordination. We add an epoch register per router that counts down until the next pre-drain and drain window. As is common in chip multiprocessors, all routers operate on the same clock; thus all epoch registers are always in sync. This means that routers *do not* need to communicate and coordinate their draining with each other. This is an important advantage of DRAIN over reactive deadlock resolution mechanisms (e.g., SPIN [5]) that must perform additional synchronization between routers to detect and eliminate deadlocks. Such synchronization limits scalability and incurs significant hardware complexity despite the fact that deadlocks very rarely arise.

Epoch. The time between drain windows (the *epoch*) is parameterizable and statically chosen at design time. On one hand, draining more frequently incurs higher energy and performance overhead since it drains (and potentially misroutes) packets more often even if no deadlocks exist. On the other hand, draining too infrequently runs the risk of allowing deadlocks to persist for longer periods and impeding system performance. We explore these trade-offs in our evaluation.

2) How to Drain and Pre-Drain: We describe how the network operates and what architectural changes are necessary for draining.

Pre-Drain Window. When the epoch counter reaches zero, each router initiates a pre-drain. To implement pre-draining, credit allocation is frozen for all packets that are currently not in-flight (i.e., not traversing a link). This ensures that when the drain window begins, no packets are in motion. The length of the pre-drain window is statically determined by the maximum packet size supported in the network (in our evaluation, this is 5 cycles).

Drain Window. At every drain window, the drain path specified by our offline algorithm (Section III-B) is drained. As shown in Figure 7, draining employs a turn-table per router that overrides the VC and switch allocators. The turn-table stores the output port for which each input port is bound, corresponding to the drain path. Since only one entry is needed per input port, the turn-table is small and scales with the number of ports per router; the table size does not increase as more routers are added to the network. Turn-tables can be configured at boot time, which will permit a new drain path to be computed by our offline algorithm in the event of a link fault.

When draining, the turn-table overrides the allocators and grants the packet exclusive priority to turn onto the specified output port. Every packet currently occupying an escape VC buffer is forced to move; they must follow the drain path by turning onto the next output port in the cycle, specified by the turn-table. Draining moves each packet by one hop.[3] If packet arrives at its destination router during draining, it may immediately eject if there is a free slot in the ejection queue.

Full Drain. To address livelock, DRAIN performs a *full drain* once every N drain windows, for very large N (full drain counter in Figure 7). A full drain involves draining the entire path such that each packet in an escape VC can visit all routers and may eject upon arriving at its destination. This incurs a high performance overhead but only needs to be

[3]While it is possible to perform more than one hop, we find this to always perform worse than the single hop case

done very rarely, since the likelihood of livelock is extremely low under typical operating environments.

3) Discussion: Here we discuss how DRAIN eliminates the need for virtual networks and supports flit-based flow control.

Virtual Networks. As discussed previously, the convention is to use virtual networks to avoid protocol-level deadlocks, regardless of what mechanism is used to resolve routing-level deadlocks. The number of virtual networks is dependent on the number of message classes in the system's communication protocol. Each virtual network requires a distinct set of VCs across all routers, so that the packets of one message class never block the packets of any other message classes. This imposes a significant area and power overhead. Unlike prior solutions for routing-level deadlocks, DRAIN provides protocol-level deadlock freedom implicitly and does not require any virtual networks. This is due to two properties of DRAIN: 1) the act of draining guarantees the movement of all packets along the drain path, and 2) the drain path passes through all routers in the network by design. As a result, DRAIN ensures that any packet of any message class will eventually have the opportunity to reach its destination router, regardless of what other packets are in its way. A detailed proof is provided in Section III-D2.

Flit-Based Flow Control. DRAIN is straightforward to implement on networks that use packet-based flow control; we opt for virtual cut-through in our implementation. To support flit-based flow control (e.g., wormhole), DRAIN leverages packet truncation mechanisms from prior work [24], [25]. Since DRAIN forces flits to turn obliviously, packets may be truncated by the draining; i.e., some flits are forced to turn in one direction while others turn elsewhere. Routers are augmented with additional logic for generating a new packet whenever a packet is truncated. Specifically, the router dynamically 1) encodes the downstream flit as a tail flit and 2) embeds header information to the upstream flit. Upon ejection, all flits are buffered at the MSHRs of the cache controllers. When all flits have been ejected, the full packet is reassembled and processed as usual. Unlike in prior work on deflection routing [24], [25], packets are rarely truncated, only once every drain window.

D. Correctness

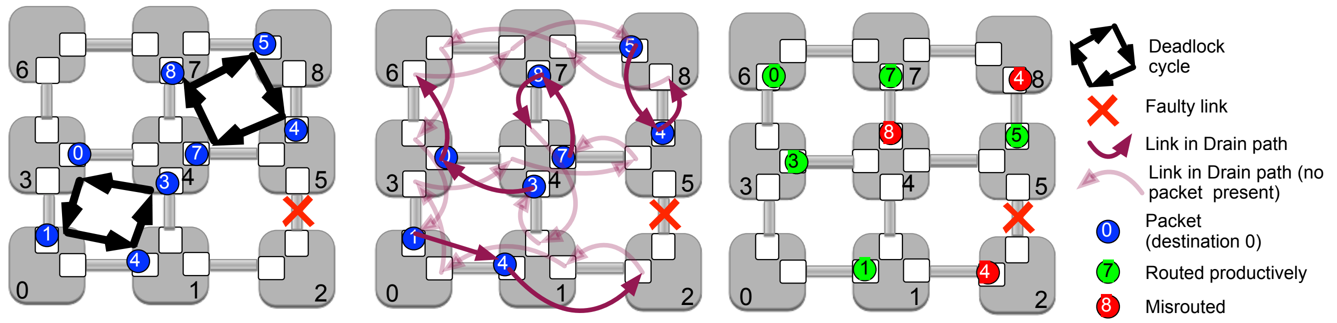
1) Proof of Routing-Level Deadlock Freedom: If a routing-level deadlock is present in a network, there must exist some cyclic dependence among packets in escape VC buffers along a set of links l. Deadlocks cannot exist in non-escape VCs since these packets will always have an opportunity to move to an escape VC. At every drain window, each escape VC in every link in l is drained, forcing any deadlocked packet in it to move one hop in some direction. After moving all packets by one hop, the deadlock will now be in one of two states: 1) the deadlock will either be broken, or 2) the deadlock will remain. The deadlock is broken if at least one of the deadlocked packets now has the opportunity to either eject from the network or turn away from the deadlock cycle onto a minimal path to its destination. If the deadlock remains, then all of the deadlocked packets in l must have moved one hop to a different link in l. Since the network is fully reachable (an assumption we make in Section III-A), for each deadlocked packet, there is always at least one link in l that would arrive at the packet's destination router and offer the opportunity to eject. All deadlocked packets are guaranteed to eventually arrive at such a link, since at each subsequent drain window, each packet will continue to move one hop to the next link in l. Though each individual drain window is not guaranteed to break a deadlock, all deadlocks are guaranteed to be broken eventually in a future drain window or full drain (Section III-C2).

2) Proof of Protocol-Level Deadlock Freedom: If a protocol-level deadlock is present in a network, there must exist some packet p_A of some message class A that is blocking a packet p_B of another message class B, and message class A is dependent on message class B in the coherence protocol. If p_B were to reach its destination router, the deadlock would be broken since all ejection queues are separated by message class, as stated in our assumptions (Section III-A). The problem is that p_A is occupying a VC buffer that p_B needs to reach its destination router. By design, at every drain window, every packet must leave its current VC buffer and move in some direction, as designated by the drain path. After moving all packets by one hop, p_A and p_B will now be in one of two states: 1) p_A and p_B have moved in different directions, breaking the deadlock, or 2) both p_A and p_B have moved in the same direction. Even though the deadlock may still exist in the latter case, p_B is now waiting in the next router of the drain path. Since the drain path is guaranteed to pass through each router in the network at least once, p_B is guaranteed to eventually reach its destination router at some future drain window.

What if ejection queues are full? Though ejection queues may be full sometimes, they will never be full due to deadlock since we assume separate ejection queues per message class (Section III-A). Thus ejection queues are guaranteed to eventually free up. There will always be at least one *sink* message class (e.g., response messages) that corresponds to the end of a coherence transaction; the ejection queue of a sink message class can always be consumed.

What if there are a burst of deadlocks? Though deadlocks occur with low probabilities, there could be scenarios where packet injection leads to a burst of deadlocks one after the other. Each drain will resolve one/more deadlocks as the packets get re-distributed. In some cases, multiple drains may be required for the deadlocks to get resolved. The periodic full-drain (Section III-C2) will guarantee that no deadlock will be persistent.

3) Livelock and Starvation Avoidance: If ejection ports are busy, blocked packets may require multiple drains before



(a) 2 deadlocks present in network (b) DRAIN window (c) Deadlocks resolved

Figure 8: Step-by-step process of how DRAIN resolves deadlocks. (a) Packets have routed into two deadlock cycles in a faulty network. (b) During the drain window, all packets follow the predefined drain path in unison. (c) After draining for one hop, both deadlocks are broken.

they eventually exit the network. Though highly unlikely, this has the risk of continuously misrouting packets to the point where they never reach their destination. Both livelock and starvation are avoided by full draining, as discussed in Section III-C2. Though a full drain incurs a performance overhead, it is very infrequent; for the vast majority of applications, a full drain is never needed. To further reduce the likelihood of livelock and starvation, we set the epoch (i.e., time between drain windows) to be no less than the expected worst-case latency of a packet in the network, which is proportional to the network diameter (i.e., the largest number of hops between any pair of routers). This ensures that if a packet is misrouted, it will have sufficient time to reach its final destination before the next drain window where it may be misrouted again. For most cache-coherent multiprocessors, the worst-case packet latency can be statically estimated because 1) the routers use well-known VC and switch arbiters that are proven to be fair, and 2) the caches and directories have finite queues and MSHRs, bounding the total number of in-flight transactions in the system.

E. *Walk-Through Example*

Figure 8 presents a walk-through example showing how DRAIN eliminates deadlocks. The X indicates a faulty link in the network between routers 2 and 5. Figure 8a shows two deadlock cycles. Packets are indicated by blue circles with their destinations specified; e.g., Packet 0 at Router 3 needs to travel south to Router 0 but is stalled waiting for Packet 1. During the drain window (Figure 8b), all packets follow the drain path for one hop, as highlighted by the magenta arrows. The complete drain path, as computed by our offline algorithm (Section III-B), is shown; the bolded arrows indicate the turns taken by the deadlocked packets. Figure 8c shows the resulting location of the packets after one hop along the drain path. For example, Packet 4 follows the drain path to Router 2; this is a misroute. When the drain window ends, Packet 4 will need to travel back towards its destination. Similarly, Packet 0 is also misrouted away from its destination to Router 6. Packets 1, 3, 5 and 7 are routed closer to their destination. Draining for one hop successfully breaks both deadlocks. In some cases, more than one drain window may be required to clear all deadlocks.

IV. Methodology

DRAIN is evaluated using *gem5* [26] with the *Garnet2.0* [27] network model and the *Ruby* memory model. We use *DSENT* [28] to model power and area for a 11 nm process. We simulate 16 and 64-core processors with a 2-level cache hierarchy. The cache architecture uses 32KB and 64KB L1 instruction and data caches respectively and 2MB last level cache (LLC) with MESI directory coherence protocol.

DRAIN is compared against escape VCs (with routing restrictions) [3] and SPIN [5]. While DRAIN only needs one virtual network, the baseline designs need multiple virtual networks to prevent protocol-level deadlocks. In our evaluations, we provision each virtual network with two VCs. Unless otherwise specified, our default implementation of DRAIN uses 64K-cycle epochs and a single virtual network with two VCs; we refer to this configuration as *VN-1, VC-2*. For completeness, we also evaluate DRAIN with 1) the same number of virtual networks as the baselines (*VN-3, VC-2*) and 2) one virtual network with the same number of total VCs as the baselines across all virtual networks (*VN-1, VC-6*). Note that only the escape-VC baseline requires two VCs per virtual network; SPIN can operate with a single VC per virtual network. However, for a fair performance comparison, we evaluate our baselines with two VCs per virtual network.

DRAIN's performance is evaluated on a fault-free 2D mesh and faulty irregular networks. Faults are injected randomly as link failures in the network topology while ensuring connectivity is maintained. For the 4×4 network, we model 0 and 8 faulty links; for the 8×8 network, we consider a range of faulty links up to 12. For each fault case, 10 different simulations with 10 randomly selected fault patterns of the given link failures are chosen. These different patterns result in a wide range of irregular topologies. The results presented are averaged across all 10 cases. In our results graphs, latency is shown in cycles and saturation throughput is shown in packets received/node/cycle.

Table II: Key Simulation Parameters.

Real application simulation parameters	
Core	64 cores and RISCV ISA (LIGRA), 1GHz 16 cores and x86 ISA (PARSEC, SPLASH-2), 1 GHz
L1 Cache	Private, 32KB Instruction + 64KB Data 4-way set associative
Last Level Cache (LLC)	Shared, distributed, 2MB 8-way set associative
Cache Coherence	MESI (LIGRA, PARSEC, SPLASH-2); VNet=3
Network parameters	
Topology	Irregular 8x8 Mesh (LIGRA and synthetic workloads) Irregular 4x4 Mesh (PARSEC and SPLASH-2)
Routing Algorithm	DoR (Regular Mesh, Escape VC) Up*/Down* (Irregular topologies, Escape VC) Fully adaptive random (SPIN) Fully adaptive random (DRAIN)
Router Latency	1-cycle
Virtual Network	3-VNet (Escape VC, SPIN) 1-VNet (DRAIN) 2 VCs/VNet
Buffer Organization	Virtual Cut Through. Single packet per VC
Link Bandwidth	128 bits/cycle
Number of faults	0, 8 (LIGRA, PARSEC, SPLASH-2) 0, 1, 4, 8, 12 (Synthetic traffic)

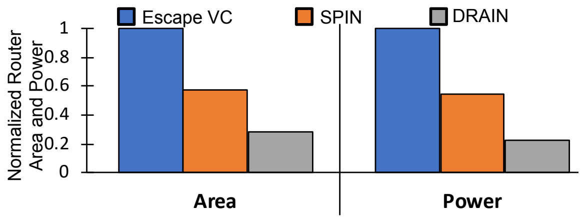


Figure 9: Router area and static power comparison.

A. *Workloads*

DRAIN is evaluated on both real-world applications and synthetic traffic. Applications are drawn from the PARSEC, SPLASH-2 and Ligra benchmark suites [8], [29], [30]. For synthetic traffic, we focus on *uniform random* and *transpose* traffic with a mix of 1-flit and 5-flit packet sizes; results for other traffic patterns are qualitatively similar. The simulator is warmed up for 1000 cycles; thereafter network statistics are collected by injecting a fixed number of tagged packets at each node in the system. Simulation completes when all the tagged packets are ejected from the network. We use an 8×8 network for Ligra and synthetic traffic and a 4×4 network for PARSEC and SPLASH-2.

V. Evaluation

We first compare DRAIN against prior proactive (escape VCs with up*/down* routing and virtual networks) and reactive (SPIN) solutions in terms of area and power. We then evaluate performance, under both synthetic traffic and real application execution. Finally we sweep the key design-space parameters of DRAIN and quantify the impact on packet tail latency, compared against the baselines.

A. *Area and Power*

In this section, we highlight the advantage of DRAIN in terms of area and power consumption. Figure 9 shows both the router area and static power normalized to the baseline escape VCs. The total router power includes that of the baseline hardware resources (i.e., buffers and allocators) and the power consumption of the additional resources that are required to handle deadlock. Escape VCs require an extra VC to proactively avoid deadlocks, which leads to significant overhead. Both baseline systems (escape VCs and SPIN) require multiple virtual networks to ensure protocol-level deadlock freedom. DRAIN, on the other hand, inherently eliminates protocol-level deadlocks and thus improves router power by about 77% compared to the baselines.

As shown in Figure 9, the simplified design of DRAIN yields almost 72% reduction in area compared to escape VCs. The majority of the area reduction comes from the elimination of extra virtual networks; however, it is worth noting that SPIN imposes a ∼15% overhead compared to a basic router design with dimension-order routing (DoR) to handle the extra control complexity for global coordination. In our comparison, both escape VC and SPIN have an equal number of virtual networks, but escape requires at least two virtual channels per virtual network, while SPIN can work with a single virtual channel per virtual network. DRAIN works with a single virtual network (as it is protocol level deadlock-free) and a single virtual channel within the single virtual network. This yields significant savings in router area and power with DRAIN.

Though our MESI protocol requires only three virtual networks, other coherence protocols may require even more; e.g., MOESI requires six virtual networks. In these cases, the area and power savings of DRAIN would be even greater.

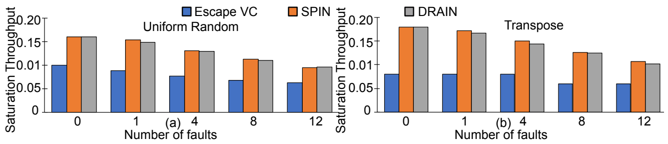


Figure 10: Saturation throughput for synthetic traffic patterns with increasing number of faults in an irregular 8×8 mesh.

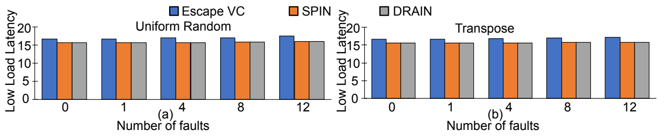


Figure 11: Low-load latency for synthetic traffic patterns with increasing number of faults in an irregular 8×8 mesh.

B. *Performance*

This section evaluates DRAIN's performance compared to prior deadlock-freedom solutions.

Synthetic Traffic. Figure 10 shows the saturation throughput for DRAIN and the baseline designs with increasing faults. Escape VCs yield the lowest throughput of the three techniques. The routing restrictions on the escape VC significantly reduce performance for all packets regardless of the low probability of deadlock. SPIN increases throughput by reacting to the rare case of deadlock. DRAIN achieves

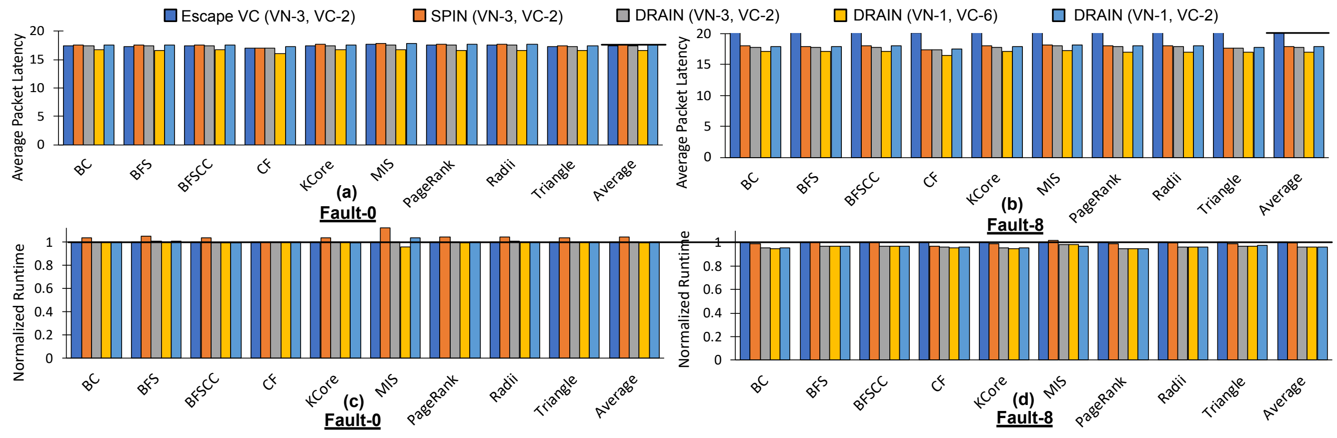


Figure 12: Packet latency and runtime of LIGRA applications on an 8×8 mesh with 0 and 8 faults.

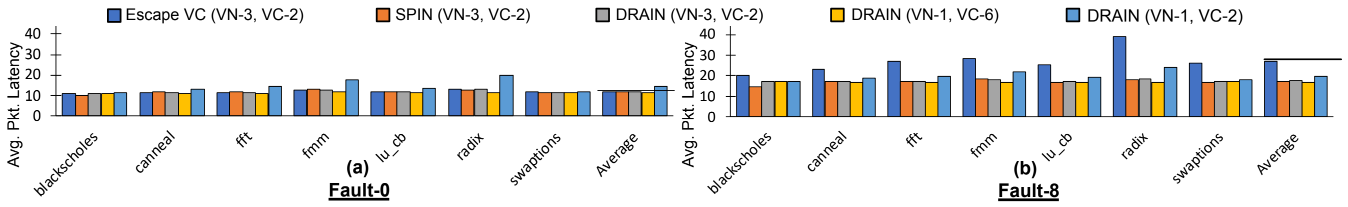


Figure 13: Packet latency of PARSEC and SPLASH-2 applications on a 4×4 mesh with 0 and 8 faults.

the same throughput as SPIN for uniform random traffic and slightly lower throughput for transpose traffic. In the event of a deadlock, DRAIN may wait longer than SPIN to resolve the deadlock, since DRAIN is an oblivious approach (i.e., it has no detection mechanism). SPIN uses a time-out of 1024 cycles, while we evaluate DRAIN with an epoch of 64K cycles. With substantially lower complexity than SPIN (Section V-A), we achieve nearly equivalent performance.

Figure 11 compares the low-load packet latency for the three designs. DRAIN achieves the same latency as SPIN and both techniques achieve better latency than escape VCs. The escape VC with up*/down* routing forces non-minimal paths on the majority of packets leading to higher hop counts. At low loads, we expect deadlock to be extremely rare; thus DRAIN achieves equivalent low-load latency to SPIN. For all techniques, low-load latency increases with increase in number of faults; faults reduce path diversity and some packets must route non-minimally around faults for all three techniques.

Application Results. Figures 12 and 13 show the performance results of SPIN and DRAIN, normalized against escape VCs on the Ligra and PARSEC workloads, respectively. We evaluate on mesh networks with 0 and 8 faults. For the topology with 0 faults, we configured the escape VCs to use minimal dimension-order routing while all other non-escape VCs use fully adaptive routing. For 8 faults, we configured the escape VCs to use non-minimal up*/down* routing, since DoR is not possible in the presence of failed links.

We evaluate three configurations of DRAIN: same number of virtual networks as the baselines (VN-3, VC-2), one virtual network with the same number of total VCs as the baselines (VN-1, VC-6) and the default configuration (VN-1, VC-2). In general, DRAIN and SPIN show similar average performance on our applications. The average packet latencies across these workloads, as shown in Figures 12a and 12b for Ligra and Figures 13a and 13b for PARSEC, are fairly close between DRAIN and SPIN. In our default DRAIN configuration (VN-1, VC-2), packet latency is higher since there are $1/3$ less total VCs than the baselines. Despite this, the application runtimes are not harmed, as shown in Figures 12c and 12d. Thus, DRAIN achieves the same performance as SPIN at $\sim 1/3$ the hardware cost (Figure 9).

C. *Sensitivity Studies*

This section explores DRAIN in greater detail.

Epoch. Figures 14a and 14b show the impact of varying the drain epoch from 16 to 64K cycles on low-load packet latency and saturation throughput, respectively. These experiments are performed on uniform random traffic. In the extreme case of 16 cycles, the network is continuously flushing the drain path, leading to poor throughput and latency due to frequent misrouting. As the epoch is increased, latency is reduced and saturation throughput is increased. Draining is best done very infrequently due to the low likelihood of deadlocks.

Tail Latency. Figure 15 shows the effect of DRAIN on the 99th-percentile network packet latency compared to escape VCs and SPIN. Since DRAIN is oblivious to deadlocks, there is a risk of allowing deadlocks to clog the network

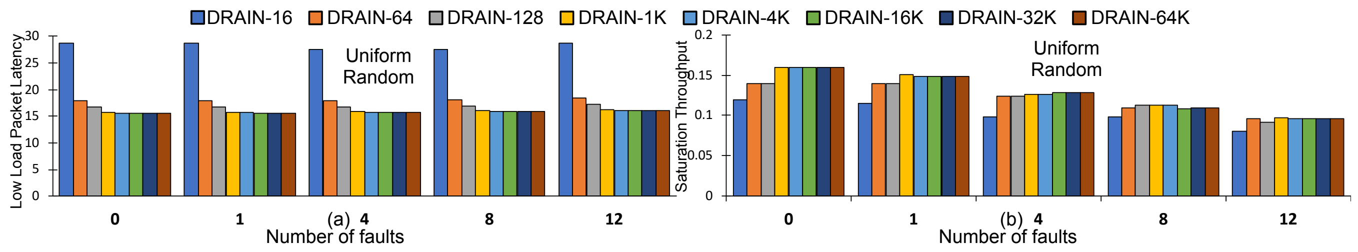


Figure 14: Low-load latency and saturation throughput of DRAIN as a function of the epoch, with increasing number of faults.

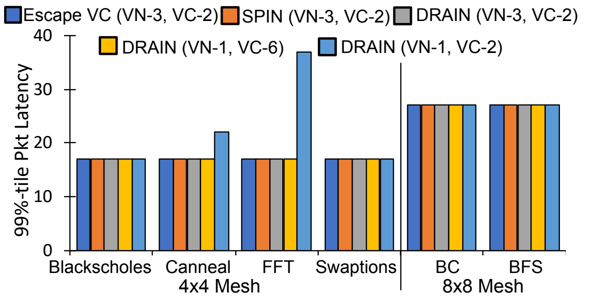


Figure 15: 99th-percentile latency comparison.

and degrade performance for a long period of time. In our experiments, we find that despite infrequent draining (i.e., large 64K epochs), the impact on tail latency is small. We observe a modest increase in 99th-percentile latency only when DRAIN is configured with less total VCs than the baselines (VN-1, VC-2), running the most memory-intensive applications.

VI. Discussion

In this section, we briefly discuss how DRAIN can be useful in fault-free networks that are challenging to make deadlock-free using existing proactive and reactive methods.

Heterogeneous Systems. Designing routing algorithms for chiplet-based architectures [17] is challenging. Multiple independently designed and verified networks must be connected through an interposer network while maintaining deadlock freedom. Just because each individual network is deadlock-free does not guarantee that the composed network will be deadlock-free. Recent work provides deadlock freedom through turn restrictions when entering and leaving chiplet-level networks. An alternative and potentially higher performance solution would be to enable deadlock recovery through DRAIN. DRAIN would allow arbitrary vendor topologies to be connected together in a deadlock-free manner without requiring any costly hardware overheads.

Random Topologies. Random topologies [31], [18] offer reduced diameter and lower average hop count making them attractive from a performance standpoint; however, high-performance, deadlock-free routing on random topologies is challenging. For example, Dodec [18] uses turn elimination to achieve deadlock freedom; however, given that its low radix routers lead to fewer turn options, some routing schemes result in non-minimal paths which can hurt performance. Maintaining all available turns would result in better performance for these networks and easier design flow; to achieve high performance, Dodec uses fully adaptive routing coupled with an escape virtual channel that uses a deadlock-free routing scheme. Similarly, Koibuchi et al. [31] use an adaptive routing algorithm combined with up*/down* routing on the escape virtual channel for deadlock freedom. Additional virtual channels increase the cost of interconnection networks; lower-cost solutions that mitigate deadlock, such as DRAIN, could be a significant improvement.

VII. Related Work

This section highlights some key related work in proactive and reactive solutions for deadlock freedom. It also briefly touches upon routing challenges in fault-tolerant NoCs.

Proactive Deadlock Freedom. Many NoC designs use Dally's theory [32] as the underlying mechanism to provide deadlock freedom. The commonly used turn model [33] is a restrictive implementation of Dally's theory. Escape VC-based solutions [3] are also used extensively to improve path diversity while ensuring deadlock freedom. EbDa [34] proposes a new approach that improves the scalability of analysis for designing deadlock-free routing algorithms. Bubble Flow Control (BFC) [35], [36], [37], [38] is a proactive mechanism that avoids deadlocks in ring and tori networks by ensuring that there is at least one free buffer in any ring via careful packet injection. Bubble coloring [39] has been proposed for deadlock freedom in irregular topologies. However, it incurs frequent misroutes and is less robust to dynamic faults, requiring complex router logic to track virtual bubble rings. **Reactive Deadlock Freedom.** DISHA [6], Ping-Bubble [40] and Static Bubble [7] introduce additional buffers at design-time that remain off, and are turned on upon detection of a deadlock via timeout counters. These buffers can be used to make forward progress. DISHA uses a circulating token to control access to these buffers, while Ping-Bubble and Static Bubble rely on probes to detect a deadlock and allow packets that are part of the deadlock to get access to these extra buffers.

Fault-Tolerant NoCs. Vicis [19] uses a heuristic to determine routing turn restrictions for deadlock avoidance. However, this heuristic fails to *guarantee* deadlock freedom [41]. Immunet [42] uses local BFC [35] in a ring constructed using the spanning tree of remaining nodes in the network. Ariadne [41] adapts the topology-agnostic off-chip up*/down* [9] routing algorithm to find deadlock-free paths in a faulty topology. uDIREC [43] extends this work to cover unidirectional link failures by modifying the methodology of spanning tree construction. BLINC [44] divides the network into segments, each with a different turn restriction, to enhance path diversity.

VIII. Conclusion

In this paper, we propose an entirely new class of deadlock-freedom technique for interconnection networks: *subactive* deadlock freedom. Traditional, well-studied approaches are either proactive, i.e., deadlock is avoided by design through turn restrictions, or reactive, i.e., deadlock is detected and recovered from. In contrast, a subactive approach periodically sweeps away potential deadlocks. We propose DRAIN, a low-cost mechanism to flush potentially deadlocked packets from their current location, which is unique in its ability to eliminate both routing-level and protocol-level deadlocks simultaneously. This new theory and approach leverages the critical observation that deadlocks are extremely rare in practice. While it is imperative that we handle deadlocks, we need not devote extra resources (VCs and virtual networks), extra complexity (detection and recovery mechanisms) nor reduce nominal operating performance (turn restrictions). DRAIN solves this problem with lower complexity and similar performance to the state-of-the-art reactive technique (SPIN) and can be implemented with minimal VCs and virtual networks. Finally, DRAIN can be reconfigured to handle random hard faults, thus increasing the usable lifetime of interconnected many-core architectures.

Acknowledgments

We thank the anonymous reviewers for their helpful comments to improve this work. We thank the SPIN authors for sharing their gem5 implementations. This work was supported by the National Science Foundation through grants FoMR-1823403, I/UCRC-1439722, and OAC-1909900 and by the Percy Edward Hart Early Career Professorship, the Natural Sciences and Engineering Research Council of Canada and the Canadian Foundation for Innovation.

References

[1] International Technology Roadmap for Semiconductors (ITRS), "More moore." http://www.itrs2.net/, 2014.

[2] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[3] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 1320–1331, December 1993.

[4] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, "The Alpha 21364 network architecture," *IEEE Micro*, vol. 22, pp. 26–35, Jan 2002.

[5] A. Ramrakhyani, P. Gratz, and T. Krishna, "Synchronized progress in interconnection networks (SPIN): A new theory for deadlock freedom," in *Proceedings of the International Symposium on Computer Architecture*, 2018.

[6] K. V. Anjan and T. M. Pinkston, "An efficient, fully adaptive deadlock recovery scheme: DISHA," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 201–210, June 1995.

[7] A. Ramrakhyani and T. Krishna, "Static bubble: A framework for deadlock-free irregular on-chip topologies," in *International Symposium on High Performance Computer Architecture*, 2017.

[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] M. D. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker, "Autonet: A high-speed, self-configuring local area network using point-to-point links," *J-SAC*, vol. 9, no. 8, 1991.

[10] T. M. Pinkston and S. Warnakulasuriya, "On deadlocks in interconnection networks," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 38–49, 1997.

[11] S. Ma, N. Enright Jerger, and Z. Wang, "Whole packet forwarding: Efficient design of fully-adaptive routing algorithms for networks-on-chip," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2012.

[12] D. K. Schroder and J. A. Babcock, "Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing," *Journal of applied Physics*, vol. 94, no. 1, pp. 1–18, 2003.

[13] J. McPherson and H. Mogul, "Underlying physics of the thermochemical e model in describing low-field time-dependent dielectric breakdown in sio 2 thin films," *Journal of Applied Physics*, vol. 84, no. 3, pp. 1513–1523, 1998.

[14] E. Takeda and N. Suzuki, "An empirical model for device degradation due to hot-carrier injection," *IEEE electron device letters*, vol. 4, no. 4, pp. 111–113, 1983.

[15] J. R. Black, "Electromigration: A brief survey and some recent results," *IEEE Transactions on Electron Devices*, vol. 16, no. 4, pp. 338–347, 1969.

[16] U. R. Karpuzcu, B. Greskamp, and J. Torrellas, "The bubblewrap many-core: popping cores for sequential acceleration," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 447–458, ACM, 2009.

[17] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. S. B. Altaf, N. Enright Jerger, and G. H. Loh, "Modular routing design for chiplet-based systems," in *Proceedings of the International Symposium on Computer Architecture*, 2018.

[18] H. Yang, J. Tripathi, N. Enright Jerger, and D. Gibson, "Dodec: Random-link, low-radix on-chip networks," in *Proceedings of the International Symposium on Microarchitecture*, 2014.

[19] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester, "Vicis: a reliable network for unreliable silicon," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 812–817, ACM, 2009.

[20] C. Iordanou, V. Soteriou, and K. Aisopos, "Hermes: Architecting a top-performing fault-tolerant routing algorithm for networks-on-chips," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 424–431, IEEE, 2014.

[21] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. Chen, "A reliable routing architecture and algorithm for NoCs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 5, pp. 726–739, 2012.

[22] K. Bhardwaj, K. Chakraborty, and S. Roy, "Towards graceful aging degradation in NoCs through an adaptive routing algorithm," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 382–391, ACM, 2012.

[23] K. A. Hawick and H. A. James, "Enumerating Circuits and Loops in Graphs with Self-Arcs and Multiple-Arcs," Computational Science Technical Note CSTN-013, Massey University, 2008.

[24] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 196–207, ACM, 2009.

[25] C. Fallin, C. Craik, and O. Mutlu, "CHIPPER: A low-complexity bufferless deflection router," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 144–155, IEEE Computer Society, 2011.

[26] 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*, vol. 39, no. 2, pp. 1–7, 2011.

[27] N. Agarwal, T. Krishna, L. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 33–42, 2009.

[28] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic, "Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pp. 201–210, 2012.

[29] 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*, pp. 24–36, 1995.

[31] M. Koibuchi, H. Matsutani, H. Amano, D. F. Hsu, and H. Casanova, "A case for random shortcut topologies for hpc interconnects," in *Proceedings of the International Symposium on Computer Architecture*, 2012.

[30] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, 2013.

[32] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, vol. 36, pp. 547–553, May 1987.

[33] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," *Journal of ACM*, vol. 41, pp. 874–902, September 1994.

[34] M. Ebrahimi and M. Daneshtalab, "EbDa: A new theory on design and verification of deadlock-free interconnection networks," in *Proceedings of the International Symposium on Computer Architecture*, 2017.

[35] C. Carrion, R. Beivide, J. A. Gregorio, and F. Vallejo, "A flow control mechanism to avoid message deadlock in k-ary n-cube networks," in *Proceedings of the Fourth International Conference on High-Performance Computing*, 1997.

[36] V. Puente, C. Izu, R. Beivide, J. Gregorio, F. Vallejo, and J. Prellezo, "The adaptive bubble router," *Journal on Parallel and Distributed Computing*, vol. 61, pp. 1180–1208, September 2001.

[37] L. Chen, R. Wang, and T. M. Pinkston, "Critical bubble scheme: An efficient implementation of globally aware network flow control," in *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 592–603, IEEE, 2011.

[38] L. Chen and T. M. Pinkston, "Worm-bubble flow control," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 366–377, IEEE, 2013.

[39] R. Wang, L. Chen, and T. Pinkston, "Bubble coloring: Avoiding routing- and protocol-induced deadlocks with minimal virtual channel requirement," in *ICS '13*, 2013.

[40] Y. H. Song and T. M. Pinkston, "A new mechanism for congestion and deadlock resolution," in *ICPP*, 2002.

[41] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco, "ARIADNE: agnostic reconfiguration in a disconnected network environment," in *PACT*, 2011.

[42] V. Puente, J. Gregorio, F. Vallejo, and R. Beivide, "Immunet: A cheap and robust fault-tolerant packet routing mechanism," in *ISCA*, 2004.

[43] R. Parikh and V. Bertacco, "uDIREC: Unified diagnosis and reconfiguration for frugal bypass of noc faults," in *MICRO*, 2013.

[44] D. Lee, R. Parikh, and V. Bertacco, "Brisk and limited-impact NoC routing reconfiguration," in *DATE*, 2014.