DRAIN: Distributed Recovery Architecture for Inaccessible Nodes in Multi-core Chips

Andrew DeOrio†, Kostantinos Aisopos‡§, Valeria Bertacco† and Li-Shiuan Peh§
†University of Michigan, Ann Arbor, MI
‡Princeton University, Princeton, NJ
§Massachusetts Institute of Technology, Cambridge, MA

ABSTRACT

As transistor dimensions continue to scale deep into the nanometer regime, silicon reliability is becoming a chief concern. At the same time, transistor counts are scaling up, enabling the design of highly integrated chips with many cores and a complex interconnect fabric, often a network on chip (NoC). Particularly problematic is the case when the accumulation of permanent hardware faults leads to disconnected cores in the system. In order to maintain correct system operation, it is necessary to salvage the data from these isolated nodes.

In this work, we introduce a recovery mechanism targeting precisely this issue: DRAIN (Distributed Recovery Architecture for Inaccessible Nodes) provides system-level recovery from permanent failures. When an error disconnects a node from the network, DRAIN uses emergency links to transfer architectural state and cached data from disconnected nodes to nearby connected caches. DRAIN incurs zero performance penalty during normal operation, and is compatible with any cache coherence protocol, interconnect topology or routing protocol. Experimental results show that DRAIN is able to provide complete state recovery within several milliseconds, on average, for a 1GHz 64-node CMP at an area overhead of only a few thousand gates.

Categories and Subject Descriptors

B.4.5 [**Hardware**]: Input/Output and Data Communications—*Reliability, Testing, and Fault-Tolerance*

General Terms

Reliability, Design

Keywords

Network-on-Chip, Fault-Tolerance, Recovery, Resilient Systems

1. INTRODUCTION

In server, embedded and desktop markets multi-core chips have become mainstream, and the trend is towards increasing core counts. To ease the programming of such architectures, most products support shared memory, along with multiple levels of caching and a network-on-chip (NoC) that connects them. Caches enhance system performance as they exploit temporal locality; however, they may contain dirty copies of data, thus becoming single points of failure should the cache become inaccessible. As the lifetime of transistors in such complex chips decreases with each technology node, multiple permanent faults are expected to occur in the network during the lifetime of a chip [3]. If such faults result in the loss of a network router, the caches of the corresponding node will become inaccessible, thus the CMP will no longer operate correctly, due to loss of unique data copies. Consequently, it is important to explore resilient architectures that mitigate fault-induced data loss, in order to continue normal operation with a graceful performance degradation.

Data recovery can be a challenge: a reliable network architecture should be able to detect and diagnose a network fault, reconfigure the routing algorithm to reroute around faulty links/routers, and finally recover the data that were affected by the fault. Various recent NoC works have looked into detecting and diagnosing faults, and reconfiguring the routing algorithm [5, 13], but recovering data from disconnected nodes or subnetworks is an area that has hardly been explored. A node becomes disconnected when all its adjacent links permanently fail, or whenever a core does not have connectivity to every other surviving core. Once a network node is disconnected, its cached data becomes inaccessible, since there are no functional links through which the caches can communicate to memory and/or to surviving nodes. The authors of [5] note that even at just 30 permanent faults, one node on average is disconnected, causing the entire chip to fail.

Most solutions assume that a checkpointing mechanism is available, which will roll back to a previous state where the data is safely stored in the memory [12, 15]. However, checkpointing involves a significant runtime performance and storage overhead to pro-actively backup a large amount of data, even if the data that is actually lost consists of just a few cache lines. To overcome such complexity and runtime performance penalties, some techniques propose to save the coherence data in-transit among nodes during reconfiguration [13]. However, these only address data *within the network*, and do not tackle data in the caches or processor cores (architectural state). To the best of our knowledge, no resilient solution has addressed the issue of recovering data in the event that one or more network nodes becomes disconnected. In such situations, no network-level solution suffices.

1.1 Contributions

Our proposed **D**istributed **R**ecovery **A**rchitecture for **I**naccessible **N**odes (DRAIN) addresses the problem of CMP processor nodes that have become disconnected as a result of accumulating permanent faults. It guarantees that processor architectural state and dirty cache data can be safely sent to memory via dedicated cache-to-cache emergency links, tolerating any number of disconnected nodes. This feature, in combination with a resilient NoC, **guarantees full system recovery in the face of unlimited network faults.** Unlike checkpointing approaches, DRAIN does not incur any runtime performance overhead during normal operation, while the additional recovery time upon a network failure is only a few milliseconds (assuming 1GHz clock). Our solution can be implemented with **minimal hardware modifications**, resulting in an area overhead of a few thousand gates. Finally, it is flexible and can **work with any underlying architecture**, including homogeneous and heterogeneous chip multiprocessors, multiple shared or private levels of caching, any network topology, any resilient routing algorithm that guarantees connectivity upon link failures and any cache coherence protocol.

DAC 2011, June 5-10, 2011, San Diego, California, USA.

2. RELATED WORK

We discuss three major areas of related work: cache reliability, interconnect reliability and full system-level reliability solutions.

Cache Reliability. Modern designs already implement a variety of resiliency mechanisms to protect individual caches, for example error correcting codes (ECC). ECC is able to tolerate bit-errors, often a single bit, in datapath elements. Reconfigurable structures [1, 8] are another common solution, where extra cache lines are added at design time. Post fabrication, these extra lines are configured as substitutes for faulty lines. Another reliability solution is triple modular redundancy, typically used to protect control logic by triplicating it and voting among the three outputs. This solution is very expensive in terms of area and power overhead, and furthermore provides only probabilistic reliability guarantees. While current cache reliability approaches preserve the correctness of data, they are localized solutions and can not handle faults due to lack of connectivity between multiple caches in CMPs.

Interconnect Reliability. Reliable NoCs are currently an active research area, with a range of solutions that enable a network to reconfigure around faults. Some solutions are able to reconfigure interconnect routes as long as errors remain within bounds, such as up to five faults [6]. Others allow an unlimited number of faults, but require that the faults lie within a homogeneous region, which may need to be convex [16]. This may require disabling functional routers to satisfy region requirements. Finally, a few solutions exist that do not place bounds on either the number or the configuration of faults [5, 13]. In these works, the network degrades as the number of faults increases: links break and nodes may become disconnected over time. While these solutions provide a mechanism for reconfiguring the interconnect and routing around faults, they are at the network level and do not address the problem of recovering data from nodes that become disconnected.

System-level reliability. By contrast, checkpointing schemes [12, 15] can be leveraged to provide error recovery for the network, attempting to recover system state when an error is detected. A checkpoint-enabled system logs cache data and architectural state, periodically copying these data to main memory. In the event of an error, state can be recovered from the logs in memory and the system is rolled back to an earlier execution point. These mechanisms work in a proactive manner, always preparing for an error. Consequently, they require significant hardware overhead to accommodate buffers, which can be on the order of 512KB in size [15]. Moreover, they incur performance overheads during normal operation, which can exceed 6% [12], an overhead that is incurred even in the absence of errors.

In short, existing cache protection schemes are not able to recover data when a node becomes disconnected, and existing network protection approaches are not able to recover cache or state. Checkpointing mechanisms can recover system state, but at a significant hardware or performance cost. These limitations suggest reactive methods that might provide recovery functionality, while avoiding costly hardware and performance overheads. DRAIN tackles precisely these goals. It is a reactive recovery design that is highly robust, able to recover data from a faulty system, even when multiple nodes have become disconnected by an interconnect error. It incurs zero performance overhead during normal operation, only reacting to the detection of an error, and it has a low area overhead.

3. DRAIN ARCHITECTURE

The DRAIN solution augments a CMP with dedicated "emergency links" operated by distributed hardware controllers and connecting nearby caches. When an error is detected, DRAIN suspends execution, flushes dirty cache data and architectural state to memory, and then allows the OS to re-map the address space. Normal operation can then resume, with all data prior to the error recovered.

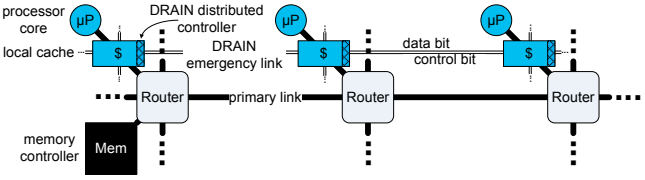

Figure 1: DRAIN-enabled system. An existing CMP is augmented with emergency links and small controllers to transfer the data from caches that have become disconnected.

Figure 1 shows the high-level modifications to a CMP architecture that are required to implement DRAIN. The baseline system consists of nodes, each with a processor core, local cache and router, with some nodes also connected to memory controllers. Nodes communicate through a flexible interconnect, in this case, a network on chip (thick lines in Figure 1). DRAIN adds 2-bit network links that connect neighboring caches together (thin double lines in Figure 1), with one bit for data and one for control. The emergency links are used only during recovery, when a node or a subnetwork becomes disconnected, in order to recover cache data and architectural state that would otherwise be lost.

Recovery Overview. The error recovery process begins with the detection of an error by either hardware or software, typically handled by the underlying reliable network (Figure 2). Error detection has been extensively researched [4, 7] and is not the focus of this work. When an error renders all links to a node inoperable, a special interrupt designed for DRAIN is issued to the processors, causing all state required to resume the running processes to be saved. This information is typically stored in a Process Control Block (PCB). The PCB includes the PID, architectural registers (including the program counter, load/store queue, stack, etc.), address space, I/O port permissions, stack pointers, etc.

Next, the network reconfigures, reestablishing communication among the processing elements that remain connected. A variety of schemes are possible here, for example [5, 10, 13], as long as the interconnect enables the communication of functioning units and avoids the loss of in-transit packets. The newly reconfigured network is reflected in the figure by the disabled primary links, which leads to the occurrence of a newly isolated node, shown by the missing router (dashed) connected to a local cache and processor.

Emergency link transfer. Next, DRAIN transfers data via a combination of primary and emergency links, shown in the bottom row of Figure 2. First, the nodes that remain connected to main memory following network reconfiguration are informed by the reliable network of their connectivity via the emergency link control bit. These nodes drain their state via the main network (step 1 in the bottom row of Figure 2). DRAIN interacts with the system's coherence mechanism to transfer the address and data of only those cache lines that are dirty to main memory. For example, in the write-invalidate cache coherence protocol with `MOESI` states, lines in the `M` (modified) or `O` (owned) state may be dirty, including lines in a transition state. The processor's architectural state is then transferred to a (now evicted) cache line, enabling DRAIN to transfer the state. The Process Control Block executing on the node is included in the transferred state.

Once all the connected nodes have transferred their architectural state and emptied their caches, they advertise to each neighbor in

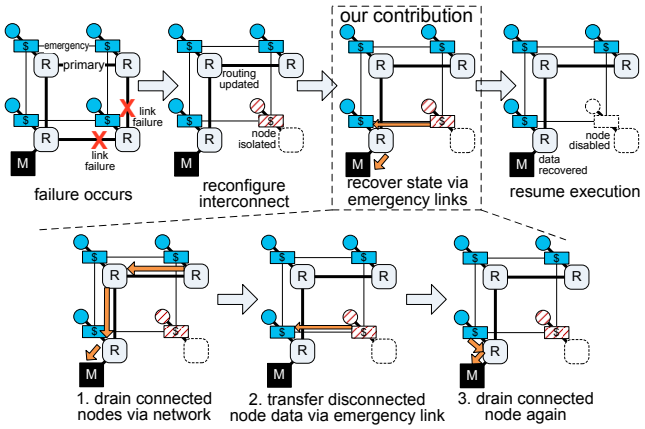

Figure 2: DRAIN system operation recovers state when an error occurs, allowing the system to be reconfigured and resumed without losing information. The DRAIN algorithm operates in three major steps, first draining connected nodes via the existing interconnect. Next, disconnected nodes' data is transferred to a nearby connected node, and from there transferred to main memory.

turn that they are ready to receive data over the emergency link control bit. In step 2, cache data and state from the disconnected node are transferred one bit at a time over the emergency link, facilitated by the distributed DRAIN controllers (shown by the hashed portion of the cache in Figure 1). A target node, which accepts a transfer from the disconnected node, ceases to advertise that it is ready to receive as the transfer begins. When a destination cache receives a cache line, it writes the line to the appropriate address, marking it as dirty in the destination cache. Upon completing the transfer, the target node drains its cache contents again to main memory (step 3 of Figure 2). At this point, all dirty cache data and architectural state in the entire CMP system has been transferred to main memory, allowing the operating system to remap addresses and processes (PCBs) to the surviving processor nodes. The reliable network signals the processors to wake up. A processor waking from a DRAIN interrupt will generate a memory request to retrieve a PCB and resume normal operation. This enables the OS to flexibly re-assign PCBs upon resume. The resume process proceeds in a similar fashion as a processor switching to a new process during a context switch. Finally, normal execution can resume on the newly recovered system.

3.1 Recovery Algorithm

Triggered by a detected failure, DRAIN's distributed recovery algorithm ensures that all cache data and architectural state reaches main memory. Leveraging both the emergency links as well as the correctly functioning portion of the primary interconnect, the algorithm finds the most efficient combination of both to deliver all data and state to memory.

To enable the operating system to remap the workload onto the reduced system resources, all caches in the system must be drained, even those that remain functional and connected. The first step in the DRAIN algorithm is to transfer data in the nodes connected to memory via primary links. Dirty cache lines are first written back, followed by architectural state. For architectural state transfer, each register or state element is first copied to a cache line in the recently drained cache. Finally, the drained cache advertises that it has completed this step via the control bit of the emergency link. Thus, other caches become able to use this cache space as

```
1 drain_via_emergency_link(this_node)
2   while target_cache not found
3     for each neighbor
4       if neighbor is connected and empty
5         target_cache = neighbor; break;
6     if target_cache not found
7       for each neighbor
8         if neighbor is toward boundary and empty
9           target_cache = neighbor; break;

10  for each dirty cache_line
11    copy_line to target_cache
12  for each register/state_element
13    copy_register to target_cache
```

Figure 3: DRAIN uses an emergency link for disconnected nodes, scanning its neighbors for a connected node to which it transfers dirty cache lines and architectural state.

target node in transferring the contents of a disconnected node.

If the node in question is not connected to main memory via primary links, emergency links are used to recover its data. The emergency links operate by copying the lines of the disconnected cache to a nearby neighbor, as detailed in Figure 3. First, the disconnected node scans the control bits of the emergency links to its neighbors (Figure 3 lines 2-5), attempting to find a neighbor that has a route to main memory. If available, it selects the first such neighbor that advertises an empty cache, using it as the transfer target. If no such neighbor exists, as in the case of a large, disconnected subnetwork, then the emergency links transfer through intermediate nodes to reach one that is connected. In this case, a node will try to find a target cache that is connected to main memory and fail. The fallback procedure is to do a cache-to-cache transfer towards the outer boundary of the disconnected subnetwork, which is ascertained from the routing logic in the reliable network (Figure 3 lines 6-9). The transfer is described by lines 10-11 of Figure 3, where the emergency link controller iterates over each dirty cache line, transferring the dirty data, along with its address. Then, architectural state is transferred to the target cache as shown in lines 12-13. Finally, the target node, upon receiving the data, will be analyzed again to find the next best step towards main memory. If the target node has no direct route to memory, data will traverse several other nodes before reaching memory via primary and emergency links.

3.2 Discussion

Emergency links. In order to ensure their correct operation, emergency links are used only during a DRAIN recovery, and are otherwise disabled with power-gating. Due to their infrequent use and power-gating, emergency links realize a significantly lower risk of wearout-induced permanent faults, as well as negligible power consumption during normal operation. The low area profile of the emergency links further reduces their exposure to failures. Reliability can be enhanced even further by adding ECC or TMR to the emergency links, a reasonably cheap solution for a two-bit link (3 additional bits for fully correctable ECC or 4 bits for TMR).

DRAIN is a flexible system, and can support a variety of architectures, as long as caches can be connected together with emergency links. While our experimental results focus on an NoC architecture with private L1 and L2 caches at each node, other configurations are possible. The key component is for the caches to be connected via emergency links, and for the existing interconnect to be fault-tolerant. The existing on-chip fabric does not weigh on the emergency link design, thus any CMP architecture is supported, as long as it can reconfigure around faults, and it remains connected to main memory, enabling correct operation once DRAIN has suc-

cessfully recovered the data from disconnected nodes. As faults increase, the possibility of a disconnected memory controller becomes a more realistic scenario, rendering the system inoperable even after reconfiguration. In a safety critical system, a variety of enhancements are possible to ensure that main memory remains connected to the network. The addition of redundant links to main memory is a high-performance solution. Alternatively, emergency links can be added from connected memory controllers to nearby caches, thus creating more routes for draining from caches to main memory.

Simultaneous faults. Failures in any configuration are supported by DRAIN. System failures accumulate as wearout-related faults occur throughout the lifetime of the chip. The frequency of fault occurrence is reflected by the Mean Time Between Failures (MTBI), typically measured in the order of months or years [11], consequently multiple faults occurring at the exact same time are unlikely. Thus, our solution supports any total number of faults occurring through the lifetime of a chip, but one at a time. Since fault rates remain an active research area, we present our results in terms of absolute numbers of faults.

Heterogeneous multi-cores. Nodes in the system can be either homogeneous or heterogeneous, and caches can be of any size. While different processing elements do not affect the DRAIN system, the emergency link controllers must be adapted for non-uniform cache sizes. In case of different cache sizes at different nodes, the DRAIN controllers are augmented with a small amount of additional logic to allow the transfer of partial cache contents, supporting the case where a large cache must be drained to a smaller one. The larger cache transfers dirty lines until the smaller target cache is full, upon which the target cache flushes its contents to main memory via the emergency links. This process is repeated until the entirety of the larger cache has been drained. Systems with uniform cache size can be optimized so that entire caches can be transferred without additional logic.

Cache configuration. Whether caches are inclusive or exclusive is a consideration in a DRAIN implementation. In the case of inclusive caches, only the higher level (farthest from core) inclusive caches must be transferred, with the lower level caches first being written back to the higher level cache. Exclusive caches must be transferred separately, copying the contents to a similar cache. In this case, a single emergency link can be shared by multiple levels of exclusive caches, with the addition of logic to support sharing the link. Another architecture to consider is the case of write-through caches. Here, the data from caches is already present in main memory, thus, when a node becomes disconnected, no cache data is truly lost. However, architectural state must still be transferred via the emergency links.

4. HARDWARE IMPLEMENTATION

Implementing DRAIN in a chip multiprocessor requires minimal hardware modifications, as it maximizes the reuse of existing cache controller logic (Figure 4). In order to transfer a line in our setup, it is necessary to transfer both data (512 bits) and address (32 bits), a total of 544 bits per transferred cache line, corresponding to 544 cycles of communication delay. Architectural state is transferred in a similar fashion, since it is first copied to a recently drained cache line. By increasing the number of data bits, it is possible to facilitate faster communication but, as we found in our experimental results, transfer time over the emergency links is not a performance-limiting factor. On average, about 4% of the total recovery time was spent in emergency link transfers: while adding bits to the emergency link would reduce this value, the remaining 96% of the time spent flushing data via primary links would not be improved. Furthermore, since reconfiguration is a rare event, the design trades off time for area by serializing communication.

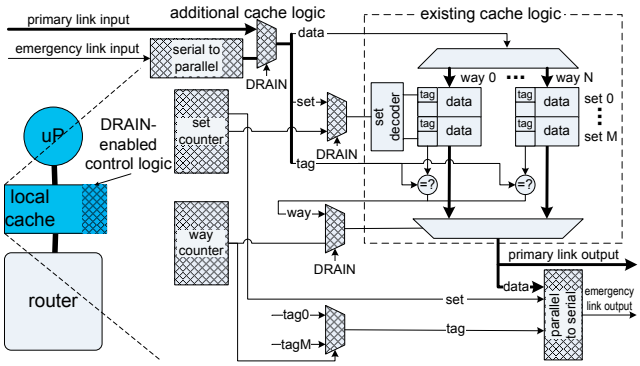

Figure 4: DRAIN-enabled cache controller with hash marks indicating hardware additions. During emergency link operation, cache control is taken over by small counters which iterate over the cache lines, serializing the data for transmission over the emergency links. When operating as a target cache, the received data is de-serialized and written to the cache.

The emergency links are controlled by small hardware units augmenting each cache controller, shown in Figure 4 with hash marks. Muxes are added to the control and data inputs of the existing cache controller (marked by the dashed box in Figure 4), allowing the emergency links to take control of the cache. During DRAIN operation, that is, recovery, a cache can either send data or receive it. When a cache is operating as a target cache receiving data, data enters via the emergency link, and it is first de-serialized by a register the size of a cache line's data and address at the input. Next, data is written to the cache using the existing cache controller hardware.

Sending data over the emergency links requires iterating over each line in the cache, which is accomplished by the set and way counters of Figure 4. These counters determine which cache line to read, and the data is then stored in the parallel to serial converter, which sends it over the emergency link. The address is sent together with the cache data, reconstructed from set, tag and block offset.

The DRAIN controller is designed to minimize hardware overhead, maximizing reuse of existing hardware. It is a simple mechanism, and since the cache is not being used in its normal capacity during a transfer, it is not necessary to add an additional cache port. As Figure 4 shows, the DRAIN logic is switched on via a mux controller during recovery. After recovery is complete, control is returned to the normal cache controller logic.

4.1 Area Overhead

As detailed in Section 4, DRAIN requires minimal hardware modifications. Figure 4 demonstrates the basic hardware structures that need to be added to each cache to implement the emergency links, which are a few counters and multiplexers. Table 1 shows the logic to implement these structures, on a sample L1-L2 private cache system, together with their 2-input AND equivalent gate count. The total overhead per node adds to 4,952 gates (2,466 gates for L1 and 2,486 gates for L2), negligible compared to a core's gate count, which is on the order of hundreds of millions of gates.

5. EXPERIMENTAL RESULTS

We evaluated an implementation of DRAIN on a 64-node architectural simulator, injecting a variety of faults in the underlying

	L1 cache (64KB 4-way)	**L2 cache** (1MB 4-way)
row counter	40 gates	60 gates
way counter	10 gates	10 gates
data multiplex	1,632 gates	1,632 gates
index multiplex	24 gates	36 gates
way multiplex	6 gates	6 gates
tag multiplex	54 gates	42 gates
serial-to-parallel	350 gates	350 gates
parallel-to-serial	350 gates	350 gates
TOTAL	2,466 gates	2,486 gates
	4,952 gates grand total	

Table 1: Additional gates required at each node to implement DRAIN. Each line in the table corresponds to a hashed component in Figure 4. Each node requires a total of 4,952 additional gates.

network-on-chip. We used the SPLASH2 benchmarks as our workloads to evaluate performance during a system recovery.

5.1 Experimental Setup

We implemented DRAIN in the memory model (Ruby) of the Wisconsin Multifacet GEMS simulator [9]. We simulated full system recovery by combining DRAIN with an on-chip implementation of the up*/down* resilient routing algorithm, which was implemented within GEMS' Garnet network model [2]. The parameters of our experimental setup are shown in Table 2. To evaluate our scheme under real workloads, we used SPLASH2 benchmark [14] traces, injecting an additional fault and triggering a recovery after one million instructions. We evaluated full system recovery for both fully connected and partially connected networks.

network topology	8x8 2D mesh, 5-stage wormhole routers
L1 configuration	4-way, 64KB, 64-byte block, 2-3 cycle latency
L1 functionality	private, unified, write-back
L2 configuration	4-way, 1MB, 64-byte block, 4-6 cycle latency
L2 functionality	private, unified, inclusive, write-back
memory	4GB, 160-cycle access latency
coherence protocol	MOESI states, directory

Table 2: Experimental setup for a 64-node system.

Fault Model. To evaluate our solution with a variable number of disconnected nodes, we generated 100 faulty topologies, corresponding to 100 different random seeds, for various numbers of network faults (0, 10, 20, ..., 100) for a total of 1,100 topologies. We examine fault-tolerance in terms of absolute fault counts, since the fault rates needed for MTTF calculations remain an open research area. Our fault model uniformly injects gate-level faults in the router logic, similar to the fault model of Vicis [5]. Faults were not injected in the emergency links, since these are power-gated off during normal operation, allowing them to avoid wear-out related failures. We then identified the subnetwork connected to the main memory and marked all of its nodes as connected; the remaining nodes were marked as disconnected. Next, we mapped the address space to the connected nodes and simulated our benchmarks. After 1 million instructions, when the system is warmed up, we injected an additional fault, resulting in a DRAIN invocation. This single fault models the expected real-world scenario, since it is unlikely that more than one permanent fault occurs at the same time: the frequency of fault occurrence is typically measured in the order of days or months. Consequently, our performance measurements correspond to the worst case scenario: applications penalized for the rare occurrence of a permanent fault. We note that some fault-injected topologies result in disconnected memory controllers, making it impossible for the cores to reach main memory.

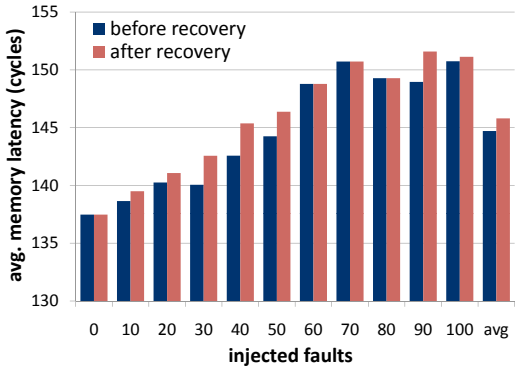

Figure 5: Average memory latency increases as faults increase, reflecting the limited routes in a fault-injected network. Beyond 70 faults, it begins to decrease, as the size of the connected networks shrink. We show the average latency before and after recovery, averaged over all benchmarks.

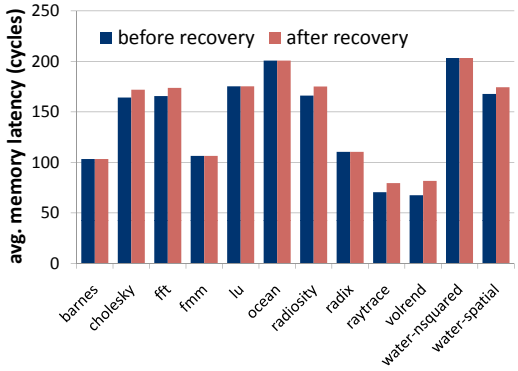

Figure 6: Average memory latency by benchmark for 50 faults, before and after recovery. Latency increases after recovery due to additional network failures: note that it also varies according to the volume of traffic.

The probability of these cases ranged from less than 1% with 10 faults, to almost 88% with 100 faults. Since topologies without a connection to memory are not useful even when reconfigured, we considered only those topologies that remained connected to main memory.

5.2 System Performance with Faults

We first examine the impact of recovery on overall system performance, by measuring average memory latency, which reflects the expected time to retrieve a line from the memory hierarchy. Figure 5 shows the average memory latency as a function of injected faults, with each datapoint reflecting an average over all SPLASH benchmarks with 100 faulty topologies each. The two bars correspond to the average memory latency before and after the recovery is invoked, with the benchmark resuming execution after an additional injected error. On average, we note a small increase in the average memory latency after recovery, less than 5 cycles (or 3%), due to the resulting topology providing fewer paths to connect the surviving nodes. We also observe that while faults increase, average memory latency (before and after recovery) also increases, the result of more disconnected nodes, and thus more lines to drain. This peaks at 70 faults and then decreases again as the network becomes partitioned, resulting in several subnetworks. In Figure 6, we also show the average memory latency for each benchmark, before and after recovery at 50 faults. We observe that for benchmarks that

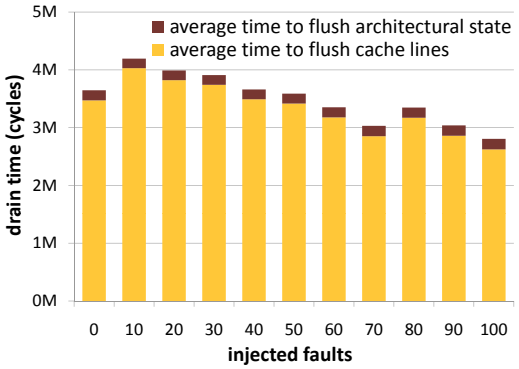

Figure 7: Drain time partitioned by cache and architectural state. Cycles to drain the entire network, averaged over all benchmarks. The time to drain architectural state is a function of the number of cores, which decreases as faults increase.

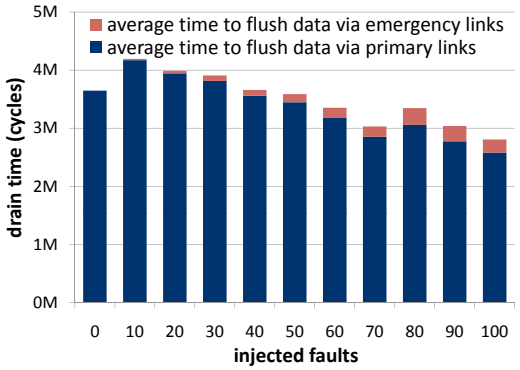

Figure 8: Drain time partitioned by emergency and primary links. Cycles to drain the entire network, averaged over all benchmarks. The drain time decreases as the number of faults increases, reflecting the decreasing surviving network size due to faults.

do not need much global coordination (thus do not generate high traffic), such as *lu*, the average memory latency only slightly increases as faults increase. On the other hand, benchmarks with high traffic, such as *volrend*, are penalized at increasingly faulty topologies (average memory latency increases significantly), since fewer available paths are available for their threads to communicate.

5.3 Recovery Time

Figures 7 and 8 show the time required for DRAIN to flush all connected caches as the number of faults increases. Each datapoint is an average over all SPLASH benchmarks, where each benchmark was evaluated over 100 randomly generated faulty topologies. We observe that for most topologies, the time to recover ranges from 3 to 4 million cycles (just a few milliseconds at 1GHz clock), a reasonable penalty for a rare event. The drain time can be partitioned based on the data that is drained (cache lines or architectural state, Figure 7), or the communication medium (emergency links or on-chip network, Figure 8). Figure 8 shows that the majority of the time is consumed in the on-chip network, because upon recovery, all caches flush their data concurrently, resulting in high network traffic and congestion. In the same figure, we observe that, although the total time to drain data decreases with increasing faults, the time to flush data via the emergency links increases. Upon further investigation, we found that for networks with a large number of faults, an additional fault is likely to disconnect multiple nodes, or break the network into several partitions; consequently, more lines have to be copied from disconnected nodes using the emergency links.

Figures 7 and 8 both reflect an overall trend of decreasing total drain time. This is the result of the decreasing number of connected nodes prior to a drain and therefore a decreasing amount of data to be drained. During the drain, both connected and newly disconnected caches are drained concurrently.

6. CONCLUSIONS

In this work we have introduced DRAIN, a recovery mechanism targeting large scale CMPs. DRAIN augments a CMP interconnect architecture with emergency links that facilitate the recovery of dirty cache data and architectural state in the event that a node or subnetwork becomes disconnected. In our experimental results, we show that DRAIN is able to recover data from disconnected nodes in any faulty network configuration, even those where aggressive failures cause network partitioning. It is able to provide complete state recovery for an entire 64-node CMP within several milliseconds and it incurs very low area overhead of 4,952 gates at each node. Thus, we demonstrate that DRAIN is an effective recovery solution for large scale CMP systems.

Acknowledgments

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. We are also grateful for the helpful input from Andrea Pellegrini in the development of this work.

7. REFERENCES

[1] A. Agarwal, B. Paul, H. Mahmoodi, A. Datta, and K. Roy. *IEEE Trans. VLSI Systems*, 13(1), 2005.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. *Proc. ISPASS*, 2009.

[3] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the era of terascale integration. In *Proc. DATE*, 2007.

[4] S. Constantinides, K.and Plaza, J. Blome, V. Zhang, B.and Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: a defect-tolerant cmp switch architecture. In *Proc. HPCA*, 2006.

[5] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester. Vicis: a reliable network for unreliable silicon. In *Proc. DAC*, 2009.

[6] M. E. Gomez, J. Duato, J. Flich, P. Lopez, A. Robles, N. A. Nordbotten, O. Lysne, and T. Skeie. An efficient fault-tolerant routing methodology for meshes and tori. *IEEE Computer Architecture Letters*, 3(1), 2004.

[7] M. Hosseinabady, A. Banaiyan, M. N. Bojnordi, and Z. Navabi. A concurrent testing method for NoC switches. In *Proc. DATE*, 2006.

[8] H. Lee, S. Cho, and B. R. Childers. Performance of graceful degradation for cache faults. *Proc. IEEE VLSI Symposium*, 0, 2007.

[9] 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. *ACM SIGARCH Computer Architecture News*, 33(4), 2005.

[10] A. Mejia, J. Flich, J. Duato, S.-A. Reinemo, and T. Skeie. Segment-based routing: An efficient fault-tolerant routing algorithm for meshes and tori. In *Proc. IPDPS*, 2006.

[11] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. MICRO*, 2003.

[12] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. ISCA*, 2002.

[13] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. *ACM SIGARCH Computer Architecture News*, 32(2), 2004.

[14] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1), 1992.

[15] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. ISCA*, 2002.

[16] J. Wu. A fault-tolerant and deadlock-free routing protocol in 2D meshes based on odd-even turn model. *IEEE Trans. Computers*, 52(9), 2003.