Topology Agnostic Dynamic Quick Reconfiguration for Large-Scale Interconnection Networks

Frank Olaf Sem-Jacobsen
Simula Research Laboratory
frankose@ifi.uio.no

Olav Lysne
Simula Research Laboratory, University of Oslo
olavly@simula.no

***Abstract*—Toleration of faults in the interconnection networks is of vital importance in todays huge computer installations. Still, the existing solutions are short of being satisfactory. They require that the system defaults into a routing algorithm that is inferior to the original, either in terms of performance, or in terms of the need for virtual channels, or both. Furthermore, since support for dynamic reconfiguration is not supported in current hardware, existing methods require the system to be halted while reconfiguration takes place in order to avoid deadlocks. In this paper we present a method that efficiently generates a new routing function in the presence of faults. The new routing function only reroutes the traffic that is affected by the fault, so that the performance of the original routing function is preserved to the extent possible. No specific functionality in the switches is required, we only require exactly the same number of virtual channels in the presence of faults as the original routing algorithm did. Finally, the new routing function is compatible with the old one, so that deadlock free dynamic transition between the old and the new routing function is immediately available. This means that our solution can easily be implemented on current InfiniBand platforms, e.g. through the OFED software stack. We demonstrate that the method is workable for meshes, tori and fat-trees, and that it is able to guarantee one-fault tolerance for all of these topologies.**

I. Introduction

The performance of high-performance and cluster computing systems relies heavily on the efficiency of the interconnection network. In latter years, the sizes of such systems have become so big that the network needs to be able to function also in the presence of faulty components. This has led to the study and implementation of various methods for routing around faults that appear while the system is running.

Such *fault tolerant routing* consists of two elements. The first is a method for finding a routing function that is efficient for semi-regular topologies, i.e. topologies such as meshes, tori, and fat-trees where some components have been removed due to malfunctions. Ideally this method should be fast - so that the system can commence normal operation as soon as possible after the fault. Furthermore, it should be efficient, so that the degradation of performance in the presence of the fault is minimal. Finally, it should not require more virtual channels for deadlock freedom than the routing algorithm needs for the fault free case. The second element of fault tolerant routing is a method for transitioning between the old and the new routing function without causing deadlock. It is well known that even if the old and the new routing functions are deadlock free by themselves, an uncontrolled transition between the two can cause deadlocks [1].

Unfortunately, there is a huge gap between the ideal described above and the current state of the art. Computing a new routing function when a fault has occurred is not at all fast. For systems such as Ranger, Atlas, and JuRoPa that are based on InfiniBand and use the OFED OpenSM subnet manager, the execution time for the routing algorithms (minhop, Up*/Down*) is in the range of hundreds of seconds to 15 min [2]. Furthermore, the routing functions that come out of the recalculation are based on Topology Agnostic methods, that disregard the carefully planned routing strategies that have been made for the fault free case. For this reason, they either require additional virtual channels, or they lead to a severe drop in performance, or both. Regarding reconfiguration between the old and the new routing function, the picture is equally bleak. Even though several mechanisms for deadlock free dynamic reconfiguration have been proposed, none of them are implemented in current hardware. Runtime reconfiguration in Infiniband simply updates the forwarding tables in the switches in the network with the values calculated by the routing function and makes no provisions for guaranteeing a deadlock free transition. A common solution to this problem is to use *static reconfiguration*. This requires the entire fabric to be drained of all traffic and shut down before the reconfiguration commences. A far more efficient solution is to change the routing tables in the network on-the-fly. This requires careful handling by the routing algorithm of the transient dependencies that occur when the routing tables are updated.

In this paper we present a novel mechanism for fault tolerant routing. The mechanism is in essence topology agnostic, and it is designed with a plug-in architecture to allow topology specific additions that increase the fault tolerance for the specific topology. For meshes, tori, and fat-trees it is able to guarantee toleration of one link fault, and it has a good probability distribution for the toleration

of multiple faults. The mechanism is able to quickly react and reconfigure the network after a topology change, and it only changes the paths for the flows that are directly disconnected by the change. Finally, it does not require any additional virtual channels, and the new paths for the disconnected flows are compatible with the existing paths in the network in such a way that deadlock free dynamic reconfiguration is guaranteed. Our reconfiguration mechanism is compatible with existing technology such as InfiniBand as it requires no specific functionality in the network elements. The algorithm is completely contained in the node responsible for configuring the network (subnet manager), and it can therefore easily be implemented and be put into production. The rest of the paper is organized as follows. In Section II we review previous attempts at creating fault tolerant mechanisms for large-scale networks. We introduce the theory that guarantees the deadlock freedom of our new routing mechanism in Section III, and the mechanism itself in Section IV. We evaluate the mechanism in Section V and the paper is concluded in Section VI.

II. Related Work

There has been a substantial amount of work presented on network fault tolerance in general and dynamic reconfiguration in particular. Almost every interconnect topology have been subject to extensive research in order to create fault tolerance routing algorithms. For multistage topologies, this has been quite successful. E.g. for the fat tree, fault tolerance is easily achieved simply by choosing a different upward path towards a different root in the network. Even a dynamic rerouting mechanism to handle faults locally has been developed [3].

The mesh and the torus is more difficult to handle in terms of fault tolerance, and several proposals exist for different baseline routing algorithms. Lysne et al [4] show that one link fault can be tolerated when using XY routing in a mesh simply by creating a path around a link fault where the first turn that is towards the centre of the mesh. Lots of proposals for meshes and tori rely on block faults [5][6][7][8] which require multiple healthy nodes to be disabled. Single fault tolerance and concave regions has also been considered [9] [10][11]. In [12], the suggestion was to route through intermediate destinations in order to avoid faulty nodes. Several of the solutions rely on virtual channels or adaptive routing to maintain deadlock freedom, and the solutions using deterministic routing without virtual channels only support static reconfiguration, not dynamic reconfiguration. Furthermore, for meshes and tori, no dynamic solution is easily implementable in current systems.

A dynamic reconfiguration mechanism is in general a mechanism that can take an arbitrary set of deadlock free paths and reconfigure the routing tables in the network to support these paths. The role of the reconfiguration mechanism is to ensure that this reconfiguration remains deadlock free. This requires that the paths before and after reconfiguration are separated in the network in some manner. Pinkston et al [13] present the double-scheme where the old and new routing functions are separated into two sets of virtual channels. Casado et Al. [14] present a mechanism for dynamic reconfiguration where the network uses a reconfiguration protocol to apply successive changes to the routing tables to converge towards the new routing function. This only works between two instances of Up*/Down*routing. Additionally, for Up*/Down*routing, Lysne et al [15] show how only the upper part of the Up*/Down* graph needs to be reconfigured. A generic mechanism for providing deadlock free dynamic reconfiguration between any two routing functions is presented in [16]. Based on some key properties of the two routing functions, a step-by-step reconfiguration process is described. In [17] another dynamic reconfiguration method is presented where the old and the new routing functions are separated by a token injected into the packet header. This token identifies whether the packet is following the old or the new routing scheme. A second token-based reconfiguration scheme is presented in [18]. The full complexity of dynamic network reconfiguration is illustrated in [16].

In common for the dynamic reconfiguration mechanisms we have reviewed is that they either rely on a specific topology agnostic routing algorithm (e.g. Up*/Down*) or they require complex and specialised fault tolerance algorithms tailored towards the specific topology or a topology agnostic routing algorithm. From our review of fault tolerance algorithms for the mesh and torus topologies it is clear that most topology specific algorithms rely on certain resources to be available (e.g. virtual channels, adaptive routing) or switch of healthy resources.

The novelty of the approach we present in this paper is that not only does it not require virtual channels or any complex reconfiguration protocol to enforce a progressive update of routing tables or using tokens, it does not require any topology specific or topology agnostic routing algorithm to generate the new routing function. This is baked into our dynamic reconfiguration mechanism in an efficient manner, with a plug-in architecture to easily support specific topologies.

III. Channel List

The key to the efficient and deadlock free implementation of the dynamic quick reconfiguration (DQR) mechanism we present in this paper is the novel data structure we use for handling channel dependencies in the network. We therefore present this in some detail in this section before we move on to presenting the DQR mechanism itself in the next section.

In order to ensure that new paths that are created to reconnect the network after a topology change (fault) the reconfiguration mechanism must beware of all existing dependencies in the network. The usual method for handling channel

dependencies is using a Channel Dependency Graph (CDG) which is a direct graph where the vertices V represent channels and the directed edges E represent dependencies from one vertex to another. A deadlock is identified as a cycle in the CDG [19]. Using this data structure to create new paths in a network involves checking if any of the new dependencies introduced by the path into the CDG leads to a cycle. The search for a cycle in the CDG has complexity $O(|E|)$. In order to find a new deadlock free path for some source destination pair, it is necessary to explore a number of possible paths and check for cycles in the CDG for every possibility. For a large topology the number of possible paths from a single source destination pair is very high. For instance, in a 20 x 20 mesh the largest number of shortest paths for a single source/destination pair (where source and destination are located at diagonal corners) is $\binom{38}{19} = 3.53 \times 10^{10}$. The size of the problem is further increased by the fact that for the existing channel dependencies and set of topology changes the only possible deadlock free paths might be non-minimal. This is clearly not a scalable algorithm.

Motivated by this we introduce the *channel list*. The channel list represents one possible arrangement of all channels in the topology such that any dependency from one channel to another only moves upwards in the list. Hence, there are no dependencies to the bottom channel, and there are no dependencies from the top channel. An example channel list for a simple 4-node ring is presented in Figure 1. It is clear that if it is possible to arrange the channels in a topology in such a list, the routing function that built the dependencies is deadlock free. There cannot be a cycle in a linear sequence of dependencies. The task of arranging the channels initially can be achieved using a linear programming solver. Every dependency in the CDG can be represented as an inequality, where the first channel of a dependency must have an index less than the second channel of the dependency. The linear programming solution to this sets of inequalities is a valid channel list given the channel dependencies and routing algorithm.

Constructing the channel list is not more efficient than searching for cycles in a CDG. However, once the channel list has been constructed, it serves as a valuable tool for creating new deadlock free paths. In fact, any new path that can be created by moving only upwards in the channel list (i.e. the next hop channel is always above the current channel in the channel list), is by design deadlock free. By combining this property with a path searching algorithm such as Dijkstra's shortest path algorithm as we do in the next section, it is possible to find a deadlock free shortest path given the existing dependencies in the channel list in an efficient manner.

Unfortunately, a channel list represents only a single possibility out of a large number of possible channel list arrangements. This means that for a given channel list

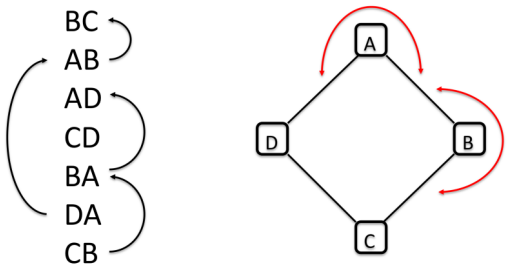

Figure 1. The channel list formed based on the dependencies from the 4-node ring. Each channel is denoted by the pair source node destination node (AB is the channel from A to B).

there might not exist a deadlock free path for a specific source/destination pair, even though the path introduces no cyclic dependencies into the CDG. In this case it is necessary to check whether certain illegal turns (going downwards in the channel list) may be permitted by rearranging the channel list. To permit a new turn the target channel of the new dependency must be moved upwards in the channel list. Furthermore, every channel to which the target channel has dependencies that are now below it must also be moved upwards. This process continues until the channel list again is valid. We present two algorithms in the next section for checking whether a turn can be permitted in the channel list, and for updating the channel list with the new turn. These are important parts of the DQR mechanism we now present.

IV. Dynamic Quick Reconfiguration

The reconfiguration mechanism we present in this paper is called Dynamic Quick Configuration (DQR) because it is high-speed and the new paths are compatible with the existing routing function. These properties are guaranteed by the channel list we presented in the previous section, which ensures that any new dependency introduced by the new paths are compatible with the already existing dependencies, i.e. there can be no deadlock. Furthermore, the high-speed of the reconfiguration mechanism comes from the fact that it will only reconfigure paths that have become disconnected. All other paths will remain the same, minimising the impact on the network. The mechanism does not require virtual channels for deadlock freedom, but where these are used by the routing algorithm (e.g. LASH [20]), these can be utilised to increase the search space for a valid path by allowing to move a path between different virtual channels. In its current incarnation the algorithm assumes that the virtual channels are divided into virtual layers such that the entire path is in

one layer or another and does not cross between them. This will be expanded in the future.

The reconfiguration mechanism is in its simplest form a topology agnostic mechanism that can be applied to any topology and routing function. The degree of fault tolerance depends on the topology and routing function used, and it can be enhanced by adding a topology specific "plug-in" to the mechanism. This plug-in will use topology specific information to identify a set of turns/dependencies that can be safely added to the existing routing function to guarantee connectivity without introducing deadlocks. If, for some reason, this mechanism fails (for instance if there are more failures than it is designed to tolerate), the core topology agnostic functionality in DQR takes over with some probability of success.

The DQR mechanism consists of two parts. The first part is the mechanism responsible for building the channel list structure based on the dependencies caused by the paths that were set up by the routing algorithm. This task can be quite time consuming in large networks. Fortunately the task can be run in the background after network configuration has completed, in advance of any topology changes/faults. The second part is the reconfiguration mechanism itself. This is invoked whenever the subnet manager detects a change in the network topology that requires some paths to be rerouted. The general view of the algorithm is as follows:

1) Construct the channel list as soon as initial routing is complete
2) Wait for topology change/failure
3) Identify all disconnected paths
4) Calculate new paths using the preconfigured channel list.

In this section we first give a general overview over the dynamic quick reconfiguration algorithm. Then, in the next two sections we describe the reconfiguration algorithm and evaluate the complexity of the algorithm to determine its efficiency. For more details and listing of the pseudo algorithms we refer the interested reader to [21].

A. Overview

The first part of the DQR algorithm, generating the channel list, is quite straightforward. The first step is to identify all the flows in the network and the dependencies these cause between channels. We use the term *channels* to encompass all the virtual channels that might be utilised in the network. The result of this step is a list of (virtual) channels and which channels these have dependencies to. Based on this list we can construct the channel list such that every channel is below (has lower index than) all channels to which it has dependencies. As we indicated in Section III this is quite easily done using a linear programming solver where each dependency is directly translated to an inequality in the linear problem.

The second part of the algorithm is the reconfiguration mechanism itself. The first objective is to identify all the flows (source/destination pairs) that are disconnected because of the fault. Once this is done the next step is to generate new paths for the disconnected flows, and herein we have three options.

1) Create a topology agnostic local reroute around the failed element and have all flows (if possible) use this reroute for connectivity.
2) Enable topology specific turns in the channel list to guarantee connectivity through a topology specific plug-in
3) Reroute all disconnected flows end-to-end

The performance of these options depends on the topology. As we will see in the evaluation section, the local reroute option works well for mesh and torus topologies, while the end-to-end reroute works better for fat trees. For guaranteed fault tolerance in the mesh, however, the topology specific plug-in is required.

When finding a new path for a disconnected flow it is usually necessary to introduce illegal turns into the channel list to create connectivity. For instance, if a link fails in a mesh that relies on XY routing, an illegal YX turn must be used to create a new path. Similarly, for fat trees that require switch to switch connectivity, a single link fault can lead to the need for introducing new downward to upward turns at different places in the topology which are illegal for the original routing algorithm. Consequently, the function that finds new paths for the disconnected flows must contain a mechanism to check whether a turn can be made legal by rearranging the channel list or not. We use Dijkstra's shortest path algorithm to generate the shortest possible paths for the disconnected flows given the constraints of the channel list and existing paths. For this step there are several design choices which we evaluate in Section V.

1) Should the path be in the same virtual layer as the original one, or can it be moved to a different layer
2) Should each illegal turn be considered separately, or in combination with other illegal turns existing on the same path

Both options represent trade-offs between run-time complexity and fault tolerance probability. Searching for paths in all virtual layers increases the runtime, but also increases the probability of finding a connected, deadlock free path. Similarly, checking the legality of a single illegal turn against the channel list is a relatively simple operation, while checking the sequence of legal turns required for a specific path is more complex. However, this added complexity leads to an increased probability that the resulting path is deadlock free. Let us review the algorithm in more detail.

B. Rerouting the Affected Paths

The main algorithm for quickly reconfiguring the network is as follows:

1) Identify all disconnected flows F
2) If a plug-in is enabled, execute it to enable the required turns by reordering the channel list
3) For each disconnected flow, find a valid path with the current channel list using Dijkstra's shortest path algorithm with the following modifications:
   - Ensure that the next channel to be tested for a path is (or can be made) valid in the channel list
   - Include the number of turns that have to be enabled in the cost function to prefer paths with fewer turns that must be enabled
   - Include the specific new turns in the cost function to preference turns that have already been enabled by other paths

   a) Once a path has been selected, rearrange the channel list to enable the required turns
4) Finally, update routing tables

The core of the algorithm is the search for the new paths using Dijkstra shortest path algorithm. If a topology specific plug-in is utilised, this is a straightforward effort since there will always be a path that requires no additional turns to be enabled in the channel list (this has been taken care of by the plug-in).

If a topology specific plug-in is not available, or the current topology is not supported by the plug-in (e.g. too many faults), the path search performed by the shortest path algorithm must include paths that enable one or more illegal turns in the channel list. For the sake of speed, there is only done if it is reasonable to believe that the resulting path will be shorter than any previously tested paths.

Let us discuss the other important component of the algorithm, namely checking whether an illegal turn can be enabled and the subsequent reordering of the channel list. An illegal turn creates a downward dependency in the channel list. However, it might be possible to reorder the channel list so that all dependencies again moved upwards. An example of this is presented in Figure 2, where one link in the previous example of our 4-node ring has been removed. To restore connectivity previously non-existent turns have to be used which lead to a downward dependency in the channel list. By moving the target channel of this dependency upwards in the channel list, and repeating this for the following dependencies, a new valid channel list is created.

The design choices we reviewed earlier are realised in the algorithm throughout the path search is performed. The local reroute option can be viewed as a topology agnostic plug-in that tries to enable the necessary turns to create a legal path around the failed element. Similarly, whether to search for a path in the same virtual layer, or in other virtual layers can easily be implemented by considering the channel list for the different virtual layers separately and testing each one through a separate run of the shortest path algorithm.

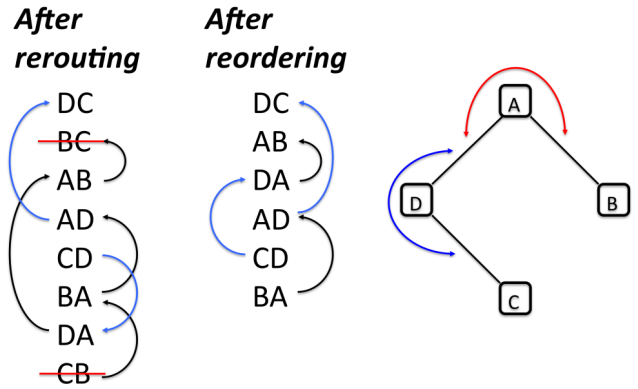

Figure 2. Reordering the channel is to allow the turns necessary to connect the topology after a failure.

Finally, whether to consider each illegal turn separately, or combined with the others is implemented in the shortest path algorithm by keeping a local copy of the channel list which is continuously updated with the enable turns. For more details we refer the reader to [21].

Complexity of the resulting reconfiguration function is $O(|F||T||V|log|V|)$, where F is the number of flows that must be rerouted, T is the small subset of channels that must be moved in the channel list, and V is the number of switches/nodes in the network. The exact values of the number of flows that must be rerouted, F, are very low, and we present these in the evaluation section. For a more detailed discussion we again refer to [21].

C. An Example Topology Specific Plug-In for the Mesh Topology

The main idea for this plug-in is to always ensure that the local paths around the disconnected link goes towards the centre of the mesh as described in [4]. So, instead of creating an arbitrary local reroute around a link, we add a specific rule for how to select this path. Furthermore, we enable the turns required to enter and exit this path at either end of the failed link. In this manner once the local reroute has been successfully established together with the additional turns, the subsequent search for paths for the disconnected flows can proceed without having to enable any new illegal turns.

V. Evaluation

We have evaluated DQR for several of the most common topologies, mesh, torus, and fat tree. The purpose of the evaluation is to see how efficient the mechanism is at re-configuring the network with a connected routing algorithm after a fault event. We also evaluated several of the options we have outlined through the description of the mechanism such as adding a local reroute path, adding topology specific

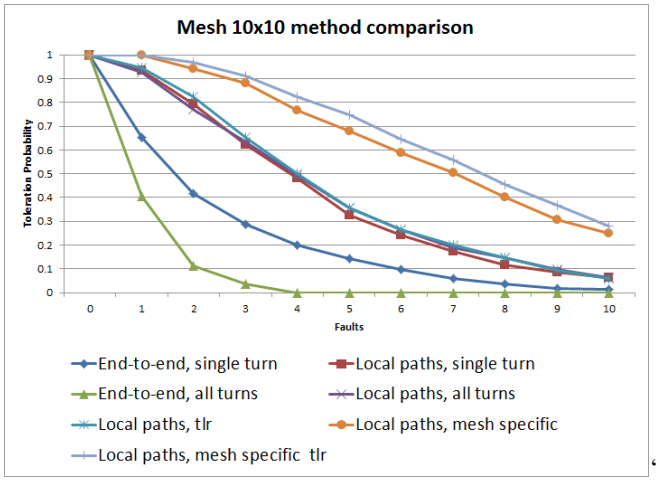

Figure 3. Mesh fault tolerance comparison.

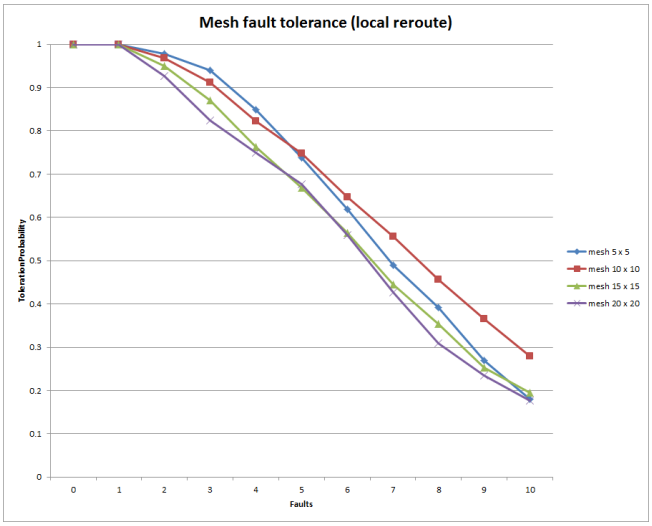

Figure 4. Mesh fault tolerance capability.

dependencies, checking the validity of all illegal turns on the path or just each single turn, and doing this for all paths or just the local reroute. The algorithm has been implemented in Python and it is run on topologies with routing tables dumped by the latest version of OpenSM (OFED 1.5.3.2). Every data point in all the figures represents 500 separate runs which consists of introducing a random faults, reconfiguring, and if successful introducing a new fault and so on. The results are presented in the next sections for the different topologies.

A. *Mesh*

The mesh is a well-known and simple topology where one link fault tolerance can be guaranteed as we discussed earlier. Let us first discuss Figure 3 where we compare all the different variations of the DQR mechanism in a 10 x 10 mesh. The x-axis is the number of link faults (inserted one after the other with reconfiguration in between) introduced into the topology and the Y axis is the probability of finding a connected and deadlock free solution. The keywords ”end-to-end” and ”local paths” signify whether the new paths are created with or without enforcing a local reroute around the failed link. ”All turns” means that for every path that is created, all illegal turns along that path have been considered together, as opposed ”single turn” where each turn is considered in isolation. Finally, for ”turn local reroute (tlr)” we consider all turns along a path together, but only for the local reroute path, not the full paths for the flows, and ”mesh specific” is the mesh specific plug-in variation.

It is clear from figure that the two mesh specific variations are the only variations that are able to guarantee toleration of one link fault, and they have the overall best probability of tolerating further faults. Of these two, the best solution is to consider the sequence of all illegal turns for the local reroute if the mesh specific plug-in fails. In the middle we find the rest of the variations around the local topology agnostic reroute solutions, and the best of these is also the one where we consider all the illegal turns on the path together rather than individually. We get the worst performance if we do not add the local reroute and consider all the illegal turns on the path together. This is because this solution requires there to be a valid new path for all possible sources for a given destination, not just the ones that are disconnected by the fault.

The conclusion is that with a topology specific plug-in mechanism we can guarantee one link fault tolerance in a mesh and have a reasonable probability of tolerating several more subsequent faults.

To evaluate the scalability of the solution for the mesh we tested a 5 by 5, 10 x 10, 15 x 15, and 20 x 20 mesh using the mesh specific plug-in together with single turn local path. The results are presented in Figure 4. The figure clearly shows that we are able to guarantee connectivity with one fault. Furthermore, the degradation in probability is similar for every topology size which indicates quite good scalability properties.

Finally, to put the complexity of the reconfiguration algorithm into context we review the numbers of flows that have to be reconfigured after a link has failed in the 20 x 20 mesh. This fraction is quite stable, it ranges from 1.7% to 2.2% when increasing the number of subsequent faults in the system from one to 10. Around 2% of all the flows are configured which ensures good runtime and scalability for the reconfiguration algorithm.

B. *Torus*

The torus is a mesh with wraparound links so that it is fully symmetric. This topology cannot be routed without using virtual channels to guarantee deadlock freedom. A possible way routing the torus is using the E-cube algorithm

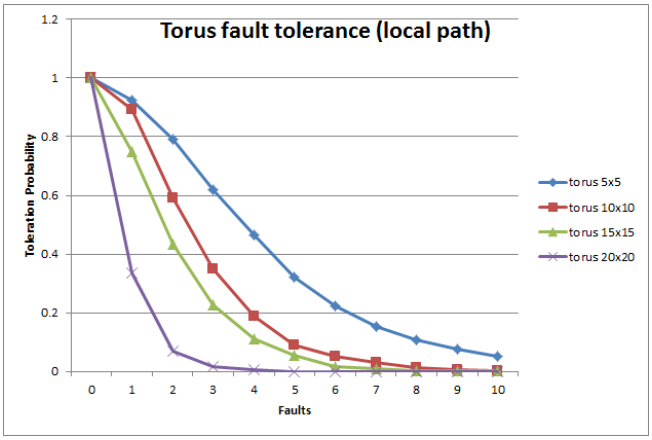

Figure 5. Torus fault-tolerant capability.

which mimics mesh routing and divides the traffic in various portions of the torus into different virtual layers. It is then possible to apply the mesh specific plug-in to every layer in the torus and achieve approximately the same probability of full toleration as for the regular mesh. In other words, using the E-cube routing algorithm DQR can guarantee one fault tolerance with a graceful degradation beyond one fault. Since the results are very similar we have not included the figure in the paper.

Another possible way of routing a torus is using LASH [20]. LASH is a topology agnostic routing algorithm that guarantees shortest path routing and divides conflicting paths (that may cause deadlock) into different virtual layers. It is therefore interesting to see how the topology agnostic DQR behaves together with the topology agnostic LASH. The results for a 5 by 5, 10 x 10, 15 x 15, and 20 x 20 torus are presented in Figure 5 where every layer is searched for a path for a disconnected flow.

There is a striking difference to the mesh probability figure. No fault tolerance is guaranteed, and for the largest torus, 20 x 20, there is only a 33% chance of tolerating a single fault. The reason for this poor performance is that LASH uses arbitrary shortest paths and tries to pack the resulting paths into as few layers as possible. This gives very little room for creating different paths without causing deadlock. The results for not searching in different virtual layers from the original are significantly worse, and have not been included in the paper.

C. Fat Tree

Finally we consider a two-tier fat tree constructed using 36-port switches. This yields a fat tree with 648 ports. The current algorithm in OpenSM for fat tree routing only provides deadlock free node to node and node to switch connectivity. Deadlock free switch to switch connectivity is not supported, although it is required for several management systems that rely on running IP over Infiniband.

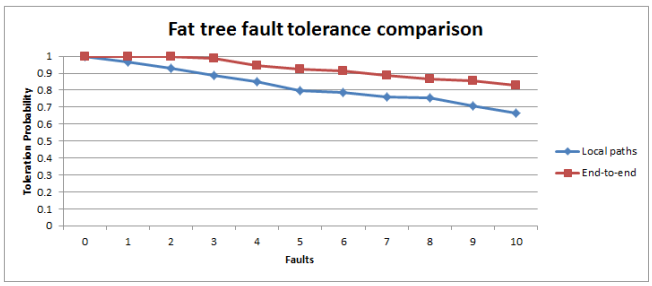

Figure 6. Fat tree fault tolerance comparison.

When evaluating the fat tree we have therefore first treated the switch to switch paths as disconnected and used DQR to reconnect them. Thereafter we introduce faults as for the other topologies. The challenge with switch to switch connectivity in the fat tree is that it involves introducing U-turns in the leaf switches of the tree. Without careful consideration of where these U-turns are placed, deadlocks will occur. Including switch to switch communication makes the evaluation more challenging since link faults in a fat tree without switch to switch connectivity can always be handled without introducing any illegal turns. The fault tolerance is therefore only bounded by the physical connectivity.

The results of the evaluation are presented in Figure 6. For the fat tree we only compare using pure end to end routing and creating local reroutes around the link fault without any topology specific plug-ins (a plug-in is not required to guarantee that the free connectivity). First, we note that DQR was able to successfully create all the necessary switch to switch paths. Second, it is clear from the figure that creating a local reroute around a link fault is not a good solution for the fat tree. In this case single fault tolerance is not guaranteed. The end-to-end algorithm can guarantee connectivity with at least one link fault, and shows a much smaller degradation with increasing number of faults.

To summarise, DQR performs better for some topologies than others, but with topology specific plug-ins connectivity can be guaranteed. The fat tree with switch-to-switch connectivity is supported with the pure topology agnostic solution, while the mesh and torus topologies require a topology specific plug-in to guarantee connectivity. Still, even the topology agnostic solution could tolerate a single link fault with 93% chance and two faults with 80% chance in a mesh. In these cases the best topology agnostic solution is to create a local path around the faulty elements and check that the entire sequence of illegal turns on this path is deadlock free. The LASH routing mechanism is more difficult to work with because the tight packing of paths into as few virtual layers as possible means there is little leeway for creating new paths. However, LASH is a very time-consuming algorithm with a significant deadlock probability when reconfiguring, so it might be worthwhile running DQR before doing a full LASH reconfiguration on the off chance

that it can be avoided.

VI. Conclusion

Having an efficient and deadlock free reconfiguration algorithm for large interconnection networks is important to maintain good utilisation of the computer system. Existing solutions either require virtual channels or have severe performance issues during the reconfiguration. Furthermore, these solutions often rely on topology agnostic routing algorithms to create connectivity. We have presented DQR, a topology agnostic dynamic reconfiguration mechanism that:

- Guarantees a deadlock free reconfiguration if connected paths are available
- Reconfigures only disconnected paths
- Requires no virtual channels or reconfiguration protocol for updating forwarding tables
- Cap has easy support for topology specific functionality through a plug-in architecture to guarantee fault tolerance
- Has low complexity

The evaluations have shown that with this architecture we can guarantee connectivity with single faults in mesh and torus topologies with a graceful degradation beyond this point. Fat trees with switch-to-switch connectivity are supported even with only the topology agnostic solution.

Further work includes implementing the functionality into OpenSM and evaluate it on real systems. More efficient topology specific plug-ins will also be researched to see whether the fault tolerance capabilities of the mechanism can be increased.

References

[1] J. Duato, O. Lysne, R. Pang, and T. M. Pinkston, "Part I: A Theory for Deadlock-Free Dynamic Network Reconfiguration." *IEEE Transactions on Parallel Distributed Systems*, vol. 16, pp. 412–427, 2005.

[2] T. Hoefler, T. Schneider, and A. Lumsdaine, "Optimized Routing for Large-Scale InfiniBand Networks," in *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, no. Lmc. IEEE, Aug. 2009, pp. 103–111.

[3] F. O. Sem-Jacobsen, T. Skeie, O. Lysne, and J. Duato, "Dynamic Fault Tolerance in Multistage Interconnection Networks," *journal*, 2009.

[4] O. Lysne, T. Skeie, and T. Waadeland, "One-fault tolerance arid beyond in wormhole routed meshes 1," *Microprocessors and Microsystems*, vol. 21, no. 7-8, pp. 471–480, 1998.

[5] Chien and J. H. Kim, "Planar-adaptive routing: Low-cost adaptive networks for multiprocessors," *19th Ann*.

[6] S. Chalasani and R. V. Boppana, "Fault-tolerant wormhole routing in tori," in *ICS '94: Proceedings of the 8th international conference on Supercomputing*. New York,, USA: ACM Press, 1994, pp. 146–155.

[7] S. Chalasani and R. Boppana, "Communication in multicomputers with nonconvex faults," *Computers, IEEE Transactions on*, vol. 46, no. 5, pp. 616–622, 1997.

[8] C.-T. Ho and L. Stockmeyer, "A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers," *IEEE Transactions on Computers*, vol. 53, no. 4, pp. 427–438, Apr. 2004.

[9] J. Duato, "A Theory of Fault-Tolerant Routing in Wormhole Networks," in *Proceedings: 1994 International Conference on Parallel and Distributed Systems*. IEEE Computer Society Press, 1994, pp. 600–607.

[10] C. Glass and L. Ni, "The turn model for adaptive routing," in *Proceedings of the 19th annual international symposium on Computer architecture*, vol. pages. ACM, 1992, pp. 278–287.

[11] N. A. Nordbotten and T. Skeie, "A Routing Methodology for Dynamic Fault Tolerance in Meshes and Tori," in *International Conference on High Performance Computing (HiPC)*, ser. LNCS 4873, R. B. V. K. P. Srinivas Aluru Manish Parashar, Ed. Springer-Verlag, 2007, pp. 514–527.

[12] M. E. Gómez, N. A. Nordbotten, J. Flich, P. López, A. Robles, J. Duato, T. Skeie, and O. Lysne, "A Routing Methodology for Achieving Fault Tolerance in Direct Networks," *IEEE Transactions on Computers*, vol. 55, pp. 400–415, 2006.

[13] T. M. Pinkston, R. Pang, and J. Duato, "Deadlock-Free Dynamic Reconfiguration Schemes for Increased Network Dependability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 780–794, 2003.

[14] R. Casado, a. Bermudez, J. Duato, F. Quiles, and J. Sanchez, "A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 2, pp. 115–132, 2001.

[15] O. Lysne and J. Duato, "Fast dynamic reconfiguration in irregular networks," *icpp*, 2000.

[16] O. Lysne, T. M. Pinkston, and J. Duato, "Part II: A Methodology for Developing Deadlock-Free Dynamic Network Reconfiguration Processes." *IEEE Transactions on Parallel Distributed Systems*, vol. 16, pp. 428–443, 2005.

[17] O. Lysne, J. Montañana, T. Pinkston, T, and J. Duato, "Simple deadlock-free dynamic network reconfiguration," *Computing-HiPC 2004*, pp. 504–515, 2005.

[18] Å. G. Solheim, O. Lysne, and T. Skeie, "RecTOR: A New and Efficient Method for Dynamic Network Reconfiguration," in *Euro-Par 2009*, 2009.

[19] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[20] O. Lysne, T. Skeie, S.-A. Reinemo, and I. r. T. Theiss, "Layered Routing in Irregular Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 51–65, 2006.

[21] F. O. Sem-Jacobsen and O. Lysne, "Topology Agnostic Dynamic Quick Reconfiguration for Large-Scale Interconnection Networks," Simula Research Laboratory, Research note, 2011. http://simula.no/publications/Simula.simula.852