A Self-Adaptive Network for HPC Clouds: Architecture, Framework, and Implementation

Feroz Zahid, Amir Taherkordi, Ernst Gunnar Gran, Tor Skeie, and Bjørn Dag Johnsen

Abstract

Clouds offer flexible and economically attractive compute and storage solutions for enterprises. However, the effectiveness of cloud computing for high-performance computing (HPC) systems still remains questionable. When clouds are deployed on lossless interconnection networks, like InfiniBand (IB), challenges related to load-balancing, low-overhead virtualization, and performance isolation hinder full potential utilization of the underlying interconnect. Moreover, cloud data centers incorporate a highly dynamic environment rendering static network reconfigurations, typically used in IB systems, infeasible. In this paper, we present a framework for a self-adaptive network architecture for HPC clouds based on lossless interconnection networks, demonstrated by means of our implemented IB prototype. Our solution, based on a feedback control and optimization loop, enables the lossless HPC network to dynamically adapt to the varying traffic patterns, current resource availability, workload distributions, and also in accordance with the service provider-defined policies. Furthermore, we present IBAdapt, a simplified ruled-based language for the service providers to specify adaptation strategies used by the framework. Our developed self-adaptive IB network prototype is demonstrated using state-of-the-art industry software. The results obtained on a test cluster demonstrate the feasibility and effectiveness of the framework when it comes to improving Quality-of-Service compliance in HPC clouds.

1. Introduction

Cloud computing is principally bringing a paradigm shift in computing, and the industry is witnessing an accelerated transition from small-scale, closed computing and data storage architectures to large, open and service oriented infrastructures. Cloud architectures offer significant advantages over traditional cluster computing architectures including flexibility, ease of setup and deployment, high-availability, and on-demand resource allocation - all packed up in an economically attractive pay-as-you-go business model for its users. Many high-performance computing (HPC) users would also like to benefit from feature-rich cloud offerings, potentially saving them substantial upfront costs while providing *instant* and *pseudo-unlimited* resource capacity for their applications. However, the effective use of cloud computing for HPC systems still remains questionable [1], [2]. Applications running on shared clouds are vulnerable to performance unpredictability and violations of service level guarantees usually required for HPC applications [3], [4]. The performance unpredictability in a multi-tenant cloud computing system typically arises from server virtualization and network sharing. While the former can be addressed by allocating only a single tenant per physical machine, the sharing of network resources still remains a major performance variability issue.

Over the last decade, InfiniBand (IB) [5] has become the most popular lossless network interconnect standard for HPC systems. The recent *Top 500* supercomputer list [6], released in November 2017, reports that about 41 of the top 100 most powerful supercomputers in the world use IB as their interconnect. It is worth to note that, even though only one system in the top 10 supercomputers uses IB, IB is the most popular standardized interconnect (as opposed to custom interconnects) in the top 100 systems affirming its mainstream use. Furthermore, IB connects 77% of the new systems added since the previous list released in June 2017 reflecting its continued popularity in HPC systems.

Thanks to the high-throughput and low-latency communication offered by IB, cloud systems built on top of an IB interconnect promise high potential of bringing HPC and other performance-demanding applications to clouds. However, current IB-based clouds are typically designed oblivious to the underlying network topology and installed routing algorithm. As a result, full utilization of the interconnect is not achieved. Typical challenges in a cloud environment, such as elastic load-balancing, efficient virtualization, and tenant performance isolation, can only be addressed in an IB system when routing is done by taking cloud specific information, for example location of the tenant nodes, into consideration [7].

- *F. Zahid, E. G. Gran, and T. Skeie are with Section of Communication Systems, Simula Research Laboratory, 1364 Fornebu, Norway. Corresponding Author: F. Zahid (feroz@simula.no)*
- *A. Taherkordi is with the Department of Informatics, University of Oslo.*
- *B. D. Johnsen is with Oracle Corporation, Norway.*
- *E. G. Gran is also affiliated with the Norwegian University of Science and Technology.*
- *T. Skeie is also affiliated with the Department of Informatics, UiO.*

In this paper, we propose a framework for a self-adaptive network architecture for HPC clouds, using IB as prototyping technology. The proposed framework is based on an external software adaptation mechanism consisting of a feedback control loop for self-optimization. The core adaptation logic is provided by an adaptation engine. The adaptation engine uses network and cloud metrics, gathered through a subnet monitoring service, together with provider-defined rules and adaptation policies, to keep the network optimized for the cloud. We also present IBAdapt, a simplified condition-action based language for defining adaptation strategies used by the optimization engine. Furthermore, based on the proposed framework, we present a working self-adaptive IB network prototype demonstrated using the OFED[1] software stack and the fat-tree network topology. Results obtained on a test cluster show the feasibility and effectiveness of our framework implementation. We show that a self-adaptive network system improves quality-of-service (QoS) compliance and reduces service level agreement (SLA) violations by proactive monitoring and optimization without any management interaction.

The rest of this paper is structured as follows. In the next section, Section 2, we motivate the need of our work. Section 3, we provide some technical background about the IB architecture, the fat-tree topologies, and self-adaptive systems. The proposed architecture of a self-adaptive HPC cloud is given in Section 4. In Section 5, we present our adaptation framework detailing the adaptation engine and the associated components. The IBAdapt language is presented in Section 6. We discuss our prototype implementation and evaluate results in Section 7 and Section 8, respectively. Related work is discussed in Section 9. We mention some opportunities of future research directions in Section 10 and conclude.

2. Motivation

From untangling complicated sequences in the DNA to simulating intricate meteorological phenomena, HPC has proved to be crucial in solving complex problems that require very high compute and network performance, unavailable on conventional computing platforms. Traditionally, the computational power provided by HPC installations has been mainly used by the scientific community. However, over the last few years, a growing interest in high-performance data analytics (HPDA) and machine learning at scale have fueled the need of HPC for the enterprises. Arguably, through HPC clouds, a large number of enterprises, as well as research institutes and academic organizations, could benefit from feature-rich cloud offerings, potentially saving them substantial capital expenditure. Moreover, studies conducted on public Infrastructure-as-a-Service (IaaS) cloud offerings like Amazon EC2, identify the network as a major performance bottleneck for efficient execution of HPC applications in the cloud [8]. The shortcomings of shared clouds due to their performance unpredictability are also reflected in the market uptake of cloud computing for HPC workloads. A recent market study published by Intersect360 Research, despite mentioning machine learning as a key new trend, shows a lack of market growth for HPC in the public clouds [9]. The report suggests that the market remains selective with respect to the jobs it offloads to the cloud platforms.

A scalable and efficient data center network is essential for a performance capable cloud computing infrastructure. HPC and HPDA applications, in particular, demand high-throughput network connectivity and low latency communication due to the abundant use of parallelization. Applications, such as those used in large-scale simulations, big data analytics, and machine learning, require frequent, irregular, and data-intensive communication between processing nodes, making the network an important determinant of the overall application performance. The performance disparity between HPC systems and cloud infrastructures is chiefly because many current cloud systems use low-cost commodity Ethernet networks providing relatively low bandwidth and high latency between nodes. HPC interconnect technologies, on the other hand, use specialized hardware to provide robust high-throughput low-latency network interconnects in HPC installations.

Recently, the use of IB in cloud computing has also gained interest in the HPC community [10], [11]. Thanks to the high-throughput and low-latency communication offered by IB, cloud systems built on top of an IB interconnect promise high potential of bringing HPC and other performance-demanding applications to the cloud. Furthermore, IB provides sufficient security mechanisms to complement in typical non-trusted data center environments. However, when clouds are deployed on IB interconnects, challenges related to load-balancing, low-overhead virtualization, efficient network reconfiguration, performance isolation, and dynamic self-adaptation obstruct full potential utilization of the underlying interconnect. This is mainly due to the lack of flexibility stemming from the very fundamentals of how IB works. To gain performance in IB, most of the communication-related work is offloaded to the hardware. Moreover, for the same reason, routes are generally static based on *linear forwarding tables* (LFTs) stored in the switches. Contrary to the traditional HPC systems, clouds exhibit a very dynamic environment, where new tenant machines are allocated, migrated, freed, and re-allocated often. The non-flexibility of IB to quickly adhere to varying configurations make IB less suitable for the clouds, and other non-traditional-HPC workloads. Even though some cloud providers, for instance Microsoft Azure, OrionVM, and ProfitBricks, provides compute instances connected using an IB interconnect (mainly through dedicated hardware resources), the aforementioned challenges still needs to be addressed to enable IB for a broader cloud adaptation. In this work, we propose a self-adaptive network architecture to better adapt IB for HPC cloud computing and make it suitable to run non-traditional and emerging HPC workloads, as well as traditional HPC applications, in the cloud.

1. The Open Fabric Enterprise Disribution (OFED) is the de facto standard software stack for building and deploying IB based application. http://openfabrics.org/.

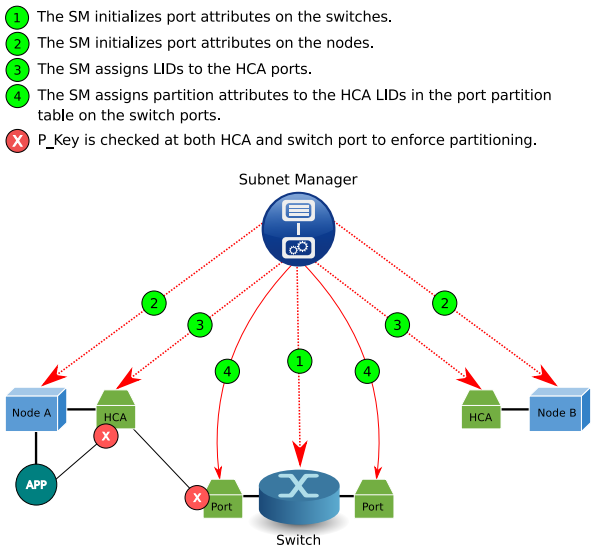


Figure 1: Partition validation process in IB networks.

3. Technical Background

In the following, we provide a brief introduction to the IB interconnect technology and the fat-tree topologies.

3.1. InfiniBand Architecture

An IB [5] network consists of one or more *subnets* interconnected using routers. Within a subnet, hosts are connected using switches and point-to-point links. The point at which an IB end node connects to the IB network is called a Host Channel Adapter (HCA). There is one *master* management entity, the *subnet manager* (SM) - residing on any designated subnet device - that configures, activates, and maintains the IB subnet. There can be several SMs in the subnet for providing fault-tolerance. Except for the master SM, all other SMs are in standby mode for fault-tolerance. The SM also performs periodic light sweeps of the subnet to detect any topology changes, node addition/deletion or link failures, and reconfigures the network accordingly.

3.1.1. Routing. Routing plays a crucial role in HPC systems, and optimized routing strategies are required to achieve and maintain optimal throughput and low latencies between nodes. A large number of routing algorithms have been proposed in the literature, which can be fundamentally classified based on the method the routing algorithm employs to select a particular path, from a source to a destination node, within a set of available alternative paths in the topology. A *deterministic* routing algorithm always selects the same path between a given pair of source and destination node. An *oblivious* routing algorithm, on the contrary, may supply one of several different available paths each time a packet is routed. However, the routing is done regardless of, or *oblivious* to, the current network conditions. *Adaptive* routing alters or *adapts* the routing decisions according to the network state, and hence, tends to avoid congested links in the network [12]. However, it is the deterministic routing which is most commonly found in HPC systems because of its simplicity offering both speed and inexpensive switch implementation as well as in-order delivery of packets, which is crucial for some applications. Furthermore, for large-scale HPC systems with multiple tiers of compute nodes, it is important to exploit the physical topology and the availability of path diversity between nodes to achieve optimal network performance. Spanning tree based protocols [13], typically employed in commodity Ethernet networks to avoid loops, are unable to exploit the topology characteristics in fat-tree networks. The main reason to choose deterministic routing over adaptive routing is that, in IB networks, adaptive routing is not readily available. Even though there are some academic and proprietary implementations of adaptive routing [14], all of the nine available routing engines in OpenSM use a deterministic routing approach. Another important reason to choose deterministic routing is the flexibility required for our implementations including criteria other than network conditions.

In an IB subnet, all HCA ports on the end nodes and all switches are addressed using local identifiers (LIDs). Each entry in an LFT consists of a destination LID (DLID) and an output port. Only one entry per LID in the table is supported. When a packet arrives at a switch, its output port is determined by looking up the DLID in the forwarding table of the switch.

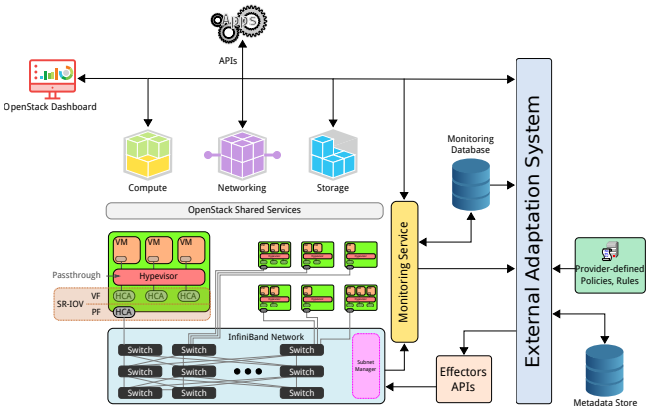


Figure 2: Architecture overview of a self-adaptive HPC cloud based on OpenStack and InfiniBand interconnect.

3.1.2. Partitioning. Partitioning is a security mechanism provided by IB to enforce isolation of logical groups of systems sharing a network fabric. The IB partitions provide similar isolation features as Ethernet 802.1Q VLANs [15]. Each HCA port on a node in the fabric can be a member of one or more partitions. Partition memberships are managed by a centralized partition manager, which is part of the SM. The SM configures partition membership information on each port as a table of 16-bit partition keys (*P_Keys*). The SM also configures switches and routers with the partition enforcement tables containing *P_Key* information associated with the LIDs. The *P_Key* information is added to every IB transport packet sent. When a packet arrives at an HCA port or a switch, its *P_Key* value is validated against the table configured by the SM. If an invalid *P_Key* value is found, the packet is discarded immediately. Packet filtering is done at both HCAs and switches to ensure that a compromised node cannot violate the security of the system. In this way, communication is allowed only between ports sharing a partition. The IB partitioning process is described in Figure 1.

3.1.3. Network Reconfiguration. After the subnet is configured as a result of the initial discovery process (*heavy sweep*), the SM performs periodic *light sweeps* of the subnet to detect any topology changes (for instance node additions/removals, and link failures). If a subnet management agent (SMA), which resides on each IB device, detects a change, it forwards a message to the SM with information about the change. As part of the reconfiguration process, the SM calls the routing algorithm to regenerate LFTs for maintaining connectivity and performance.

3.2. Fat-Tree Topologies and Routing

Many of the IB based HPC systems employ a fat-tree topology [16] to take advantage of the useful properties fat-trees offer. These properties include full bisection-bandwidth and inherent fault-tolerance due to the availability of multiple paths. Moreover, fat-trees are easy to build using commodity switches [17]. The fat-tree is a class of general-purpose network topologies characterized by multi-stage tree like structure commonly with multiple *root* nodes. The initial idea behind fat-trees was to employ *fatter* links between nodes, with more available bandwidth, as we move towards the roots of the topology. The fatter links help to avoid congestion in the upper-level switches and the bisection-bandwidth is maintained. Different variations of fat-trees are later presented in the literature, including *k*-ary-*n*-trees [18], Extended Generalized Fat-Trees (XGFTs) [19], and Parallel Ports Generalized Fat-Trees (PGFTs).

4. Architecture

The overview of our proposed architecture for a self-adaptive HPC cloud is presented in Figure 2. The cloud uses OpenStack [20] as the cloud orchestration software and IB as the underlying interconnection technology. OpenStack is a popular cloud platform for enterprises providing integrated control over a large pool of compute, storage, and networking resources throughout a data center. Core OpenStack services include *Nova* for managing the lifecycle of compute instances; *Neutron* for providing network connectivity as a service; and object, block, and image storage services (*Swift*, *Cinder*, and *Keystone*). On top of the core OpenStack services, several shared services are provided supporting features such as key management, orchestration, bare-metal provisioning, and containers support, among others. Administrative management is provided through OpenStack Application Programming Interface (API) and an interactive web-based dashboard (*Horizon*).

As shown in the figure, we propose an external adaptation system that takes provider-defined policies, rules, and constraints as an input and periodically evaluates the running system to detect violations or any potential for optimization. Provider-defined policies may include configurable parameters such as adaptation period, routing algorithm to be used, and

various thresholds for the adaptation strategies. In addition, provider-defined policies also include a condition-action list, where both system parameters and custom parameters are evaluated and a particular user-defined action performed when given conditions are found true in the next adaptation cycle. In this way, the adaptation system keeps provider-defined constraints satisfied, while at the same time it is ensured that the cloud network is optimized according to the current configurations and provider-defined adaptation strategies. Furthermore, any SLA violations, also defined in the provider-defined rules, are caught and fixed instantly, if possible, by the adaptation system.

Discounting provider-defined policies, the adaptation engine interacts with four main components: *Monitoring Service*, *Monitoring Database*, *Effectors APIs*, and *Metadata Store*. The monitoring service monitors both the OpenStack cloud system and the underlying IB subnet. For OpenStack, the monitoring service uses the OpenStack APIs to collect information about running cloud services, tenant data, and other higher level cloud information. The cloud information is then stored, and periodically updated, in a monitoring database and passed to the adaptation system. Generally speaking, such cloud information can be fetched from any cloud platform, and in particular, the framework is not limited to OpenStack. Any specific implementation, however, will need to provide support for the particular cloud API in use. Information about the IB subnet, like topology, location of HCAs, LIDs, port GUIDs, LFTs, and partitioning is obtained through interaction with the SM. As mentioned in Section 3.1, the SM is a centralized management entity in an IB network. Through the subnet management interface, the SM exchanges control packets, called subnet management packets (SMPs), with the subnet management agents (SMAs) that reside on every IB device. Using SMPs, the SM is able to discover the fabric, configure end nodes and switches, and receive notifications from SMAs. Note that the SM implementation based on the concept of centralized administration in IB networks achieves functionality similar to that of the more recent concept of software-defined networking. One difference, however, is that the SM does not allow per-flow based networking as the routing tables implemented, in the form of LFTs in switches, govern which path is taken on a per packet basis in the network. The monitoring service queries information from the SM. In addition, in case of failure detection during a light sweep, notifications can also be pushed to the monitoring service. The monitoring service keeps historical data of the port data counters queried for each HCA port in a monitoring database. The adaptation system has access to the monitoring database and it uses port counters to calculate node traffic profiles using a provider-defined traffic analysis and prediction algorithm. More details about traffic profiling are given in Section 5.1.

The adaptation system has access to Effectors APIs, again provided via an interface through the SM. The effectors APIs are used to change the routing algorithm in use, calculate and install new LFTs, and update partitioning information in an IB subnet. When adaptation system analysis asks for a change in the IB subnet to keep the HPC cloud optimized, it uses effectors APIs to apply the required changes to the IB subnet. The adaptation engine also maintains a metadata store with extensive information about the HPC cloud, the IB subnet, and historical adaptation results. We foresee that, as future work, the metadata store can be used to apply machine learning techniques for making adaptation better over time.

5. Adaptation Framework

The adaptation framework defines various components necessary to implement the functionality required to realize a self-adaptive lossless network infrastructure for the HPC clouds. Our solution is inspired by the Rainbow framework [21]. The rainbow framework uses a control loop based architectural model to monitor and continuously evaluate runtime properties of a managed system for constraint violations, and perform global or module-level adaptations when a problem occurs.

The overview of the proposed framework is given in Figure 3. The framework consists of two layers: a system layer and an adaptation layer, shown in the figure as the lower and the upper parts, respectively. The system layer concerns the underlying managed system that consists of an IB fabric, the SM, the Monitoring Service, and the Effectors APIs. The monitoring service, as described in Section 4, gathers *monitors* with information about both the upper-layer cloud system and the IB subnet. The effectors APIs expose *effectors* to the adaptation system that define mechanisms to execute a required change in the IB subnet, such as a routing or partitioning update. Both the monitoring service and the effectors APIs interact with the SM, which is responsible to configure and maintain the IB subnet. The monitors and the effectors, in combination, define the interaction between the system layer and the adaptation layer through a *Subnet Interaction API*.

The adaptation layer consists of five main components that interact with each other to provide self-adaptation capabilities to the network. A *Subnet Model* is maintained based on the information obtained from the monitors using the subnet interaction API. The core of the adaptation logic lies in the *Adaptation Engine*. The adaptation engine receives provider-defined rules and policies as an input, and evaluates if the current subnet model satisfies the rules set by the provider, and if any further optimization is possible. If a new configuration, in the form of new routing tables, has been deduced by the adaptation engine according to the employed adaptation logic, it is evaluated by the *Routing Evaluator* component of the adaptation layer. The routing evaluator estimates the overhead involved in reconfiguring the network according to the new set of routing tables. As part of the estimation, a *simulation engine* will *dry-run* the new reconfiguration to evaluate if there are enough potential benefits of installing the new configuration over the current one. If it is decided that the network is to be reconfigured, the *Adaptation Planner* plans and schedules the reconfiguration process, including provisioning the downtime required. The reconfiguration process itself is performed by the *Adaptation Executor*.

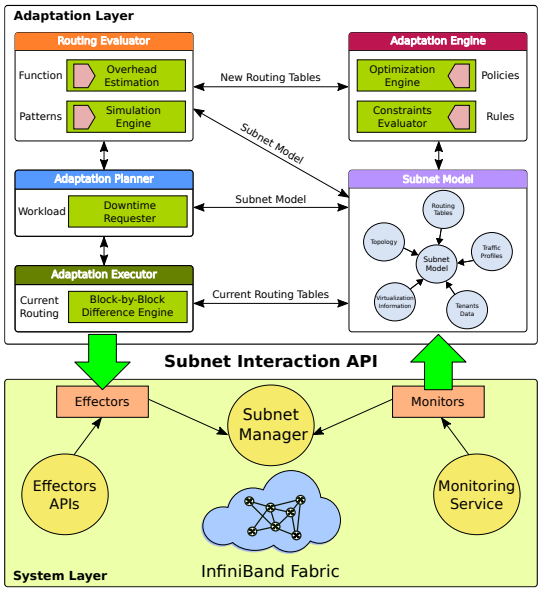


Figure 3: A self-adaptation framework for IB subnets.

Metric Set	Impact
Topology	The topology determines which routing algorithm should be used for optimal routing in the IB fabric.
Routing Tables	Current routing tables specify the paths taken by the packets in the network.
Traffic Profiles	Traffic profiles specify node traffic characteristics used for the routing optimizations.
Tenant Data	Tenant data specifies required partitioning in the IB subnet. Partitioning is used for both network isolation and optimized routing.
Virtualization Information	Routing algorithms make use of the virtualization information to route virtualized nodes efficiently.

TABLE 1: Subnet model metrics and their impact.

Each component of the adaptation framework is now described in detail giving specifics about their application in IB-based HPC clouds.

5.1. Subnet Model

The subnet model includes information about the topology of the subnet, routing tables installed on IB switches, and tenant data including information about virtual machines (VMs) running in the cloud. In addition, the subnet model also keeps traffic profiles for each compute node in the network. As part of the feedback control loop, the subnet model is kept continuously updated according to the metrics available through the monitoring service. Table 1 summarizes the impact of each of the metric sets on IB network optimization in our proposed framework. The use of a subnet model for network optimization, with respect to each of the defined metric sets, is discussed in Section 5.2.

Traffic Profiling. Network traffic profiling is a vast topic with applications in many areas including network traffic management, anomaly detection, energy efficiency, and QoS. In our framework implementation, the traffic profile of a compute node defines the amount of incoming/outgoing traffic it is assumed to receive/send over the next adaptation period. The framework defines three simple models for traffic profiling based on linear prediction, *Equal Weights*, *Linear Weights*, and *Real-time* prediction methods. However, as calculating traffic profiles is entirely defined by the provider-defined policies, more advanced methods such as those based on autoregressive models, can easily be supported. Similarly, other parameters affecting traffic predictions, like the number of historical data counters to maintain and the adaptation period, are also configurable through provider-defined policies.

If t historical data counters, $T = \{T_t, T_{t-1}, ..., T_1\}$, have been observed over time $\{1, 2, .., t\}$ with the most recent being T_t, the traffic data counters at time $t+1$, $\hat{T}(t+1)$, can be predicted by Equation 1.

$$\hat{T}(t+1) = \sum_{i=1}^{t} a_i T(t+1-i) \tag{1}$$

The predictor coefficients, a_i, represents weights historical data counters receive in predicting network traffic according to their place in the time series. In equal weights network profiling, each historical counter has the same weight in predicting future traffic, whereas, the linear network profiling uses a linear function to decrease the weight of each counter further we go in the time series. The third method we support is based on real-time data counters. In real-time network prediction, the traffic profiles are calculated based on the most recent data counter recorded (Equation 2).

$$a_i = \begin{cases} n & \text{if equal weights for data counters} \\ i & \text{if linear weights for data counters} \\ 1 & \text{if real-time prediction is used and i = t} \\ 0 & \text{if real-time prediction is used and i != t} \end{cases} \tag{2}$$

In [22], we demonstrated that for fat-tree routing, network load-balancing can be improved by using traffic profiles of the nodes represented by their incoming data traffic. This is so because, for deterministic routing, the network path selection is based on destination LID at the IB switches. More details are provided in Section 7.1.

5.2. Adaptation Engine

The adaptation engine is the crux of the self-adaptive network architecture for HPC clouds. The adaptation engine has two main roles, each represented by a different logical component acting on the current subnet model. First, the *Constraints Evaluator* ensures that the constraints provided by the cloud administrator, in terms of *rules*, are kept satisfied in the dynamic cloud. If provider-defined rules are found no longer satisfied by the current cloud configuration, the adaptation logic is triggered to find a new suitable configuration. Second, the *Optimization Engine* contains adaptation logic to find the most suitable configuration given the rule set and policies defined by the administrator. The optimization engine may find a new configuration to optimize the network, as per provider-defined policies, even when the current configuration satisfies all rules defined as constraints. Both the rules and the policies are provided to the adaptation engine using the IBAdapt language discussed in Section 6. The rules related to faults and failures in the network are system-defined; Whenever a fault is detected by the SM, for example a node is disconnected, adaptation logic is executed by default to find a new configuration. Other rules, for instance whether tenant nodes are allowed to share intermediate links in the cloud or not, are defined by the provider as needed. Similarly, policies are also user-defined and describe strategies and tactics to meet optimization goals. In particular, policies are sets of condition-action rules that specify adaptation logic, in the form of actions, to be executed by the adaptation engine when a specific condition is met. For example, a policy may define that if the average receive bandwidth per node is decreased below a certain threshold, the routing algorithm should be re-run with new node traffic profiles to calculate new routing tables.

Adaptation Logic. Policy actions can be used to write adaptation logic for the network and the upper-layer cloud system for optimization. The optimization actions are usually developed using utility functions. Utility functions express the rationale for adaptation decisions using a utility-based approach, and are therefore appropriate when adaptation conditions and effects interfere, or when defined policies are in conflict [23]. A utility function describes a real-valued scalar desirability to system states. Optimization actions compute the utilities of different possible configurations and select an appropriate configuration with the maximum utility. In the case of our self-adaptive HPC network, the system states represent different parameters indicating routing quality. The combined goal of the adaptation actions is to come up with a new set of routing tables for the network that satisfy provider-defined constraints and provide the best cumulative utility as per the provider policies and current resource availability. In our framework implementation for the IB fat-trees, all user-defined policies are evaluated and they together generate a set of parameters to be given as an input to our fat-tree routing algorithm for network optimization. Our implementation is discussed in detail in Section 7.

5.3. Routing Evaluator

After a new set of routing tables have been calculated by the adaptation engine, the Routing Evaluator evaluates two important aspects of the recommended change: the overhead of installing new routing tables (in terms of updating the LFTs), and the potential gain of the new routing configuration over the existing one. Overhead estimation is performed by taking into account currently installed LFTs and the new LFTs. The potential benefit of the new routing, calculated using utility functions, is further confirmed by running different communication patterns using a simulation engine.

5.3.1. Overhead Estimation. After the LFTs have been calculated, the reconfiguration cost depends on the time and bandwidth required to send SMPs to the switches containing LFT block updates, and the number of dropped packets during the reconfiguration [24]. The actual number of LFT block updates depends on the specific LIDs that are modified [2]. A block-by-block comparison mechanism is employed to find updated blocks. This is further discussed in Section 5.5. Considering an average case where modified LIDs are considered distributed evenly in the used LFT blocks, if n is the number of total switches in the network and ΔL is the number of LIDs updated with an average of q modifications per LFT block, the total reconfiguration time can be estimated using Equation 3. The average time for an SMP packet from the SM to reach the switch node is denoted as t_{smp}.

$$\text{Reconfiguration Time} = \sum_{i=1}^{n} \frac{\Delta L_i}{q} \times t_{smp} \tag{3}$$

5.3.2. Simulation Engine. The simulation engine is used to dry-run the new routing tables to find if there is enough potential benefit for the recommended change in the routing. Two different simulation tools can be used: A flit-level simulation engine and the ORCS Simulator. For both simulation engines, routing tables are evaluated on emulated physical topologies using the OFED utility called *ibsim*.

For the flit-level simulation, we use an extended IB flit-level simulation model [25], originally contributed by *Mellanox Technologies* [26]. The model is implemented using the OMNeT++ network simulation framework and has been used extensively in the literature to evaluate IB networks.

The Oblivious Routing Congestion Simulator (ORCS) [27] is capable of simulating a variety of communication patterns on statically routed networks.

5.4. Adaptation Planner

The purpose of the adaptation planner is to schedule the reconfiguration process at a specific time. According to the administrative policy in place, the adaptation planner can proceed with an update immediately or defer it until a later time. In critical cloud systems, the adaptation planner can use an already planned maintenance window for the reconfiguration. Alternatively, the adaptation can also be delayed until re-routing is induced by an external factor such as a change in the topology or a component failure.

5.5. Adaptation Executor

The adaptation executor is the final component of the adaptation layer in the feedback control loop. At the time when the installation of a new configuration is scheduled by the adaptation planner, the adaptation executor uses the effectors APIs to propagate actual LFT changes to the network switches. The adaptation executor uses effectors to update LFTs on the switches using a differential update mechanism. The mechanism uses a block-by-block comparison between existing and new LFTs, to make sure that only modified blocks are sent to the switches. Figure **??** shows an example update; only an updated *Block 2* with two modified entries is sent to the switch in this example. Note that non-existing paths are marked as directed to port *255*.

6. IBAdapt: A Rule-based Language

In this section we present IBAdapt, which is a condition-action driven rule-based language for defining adaptation rules and policies for the cloud service providers. The IBAdapt language is aimed to implement adaptation policies based on condition-action pairs. At each adaptation interval, the IBAdapt interpreter evaluates a policies file and executes actions when corresponding conditions are met. The main goal of the IBAdapt is to keep the language definition as simple as possible while complex actions can be provided by the system and implemented as part of the adaptation engine.

The overview of the IBAdapt interpreter and its implementation is provided in Figure 4. The language compiler is implemented using *flex* and *bison*. Flex is used to generate scanner source code for the Lexical analyzer. The lexical analyzer returns a stream of tokens that are handled by the parser, generated using bison. The parser generates a parse tree which is used by the Abstract Syntax Tree (AST) generator to generate an abstract syntactic structure of the source code given in the policies definition file. After the semantic analysis, the policies are executed. The interpreter maintained a symbol table to store variables defined by the system and the user. The minimal IBAdapt grammar in Backus–Naur Form (BNF) is given in Listing 1.

As defined by $\langle statement \rangle$ class in Listing 1, the IBAdapt language defines two basic kinds of statements: assignments and policy statements. The assignments can be used to define or assign values to the variables. There are two kind of

2. An LFT is divided in up to *768* blocks, each representing *64* LIDs and their corresponding output ports

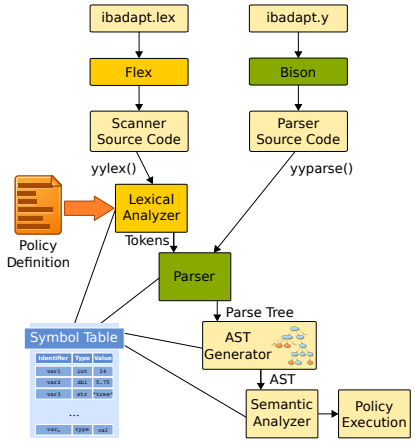


Figure 4: Overview of IBAdapt implementation.

$\langle program \rangle$ ::= $\langle statementList \rangle$;

$\langle statementList \rangle$ ::= $\langle statementList \rangle$ $\langle statement \rangle$
| $\langle statement \rangle$;

$\langle statement \rangle$ ::= $\langle policy \rangle$
| $\langle assignment \rangle$;

$\langle policy \rangle$ ::= **policy** *identifier* **(** $\langle conditionList \rangle$ **)** { $\langle actionList \rangle$ }; ;

$\langle conditionList \rangle$ ::= $\langle conditionList \rangle$ '&&' $\langle condition \rangle$
| $\langle conditionList \rangle$ '||' $\langle condition \rangle$
| $\langle condition \rangle$;

$\langle condition \rangle$ ::= $\langle factor \rangle$ '<' $\langle factor \rangle$
| $\langle factor \rangle$ '>' $\langle factor \rangle$
| $\langle factor \rangle$ '<=' $\langle factor \rangle$
| $\langle factor \rangle$ '>=' $\langle factor \rangle$
| $\langle factor \rangle$ '==' $\langle factor \rangle$
| $\langle factor \rangle$ '!=' $\langle factor \rangle$;

$\langle factor \rangle$ ::= **int** *number*
| **double** *number*
| **string** *literal*
| *identifier* ;

$\langle actionList \rangle$ ::= $\langle actionList \rangle$ $\langle action \rangle$
| $\langle action \rangle$;

$\langle action \rangle$ ::= *identifier*(); ;

$\langle assignment \rangle$ ::= **int** *identifier* ':=' $\langle expression \rangle$;
| **double** *identifier* ':=' $\langle expression \rangle$;
| **string** *identifier* ':=' $\langle expression \rangle$; ;

$\langle expression \rangle$ ::= $\langle term \rangle$
| $\langle expression \rangle$ '+' $\langle term \rangle$
| $\langle expression \rangle$ '−' $\langle term \rangle$;

$\langle term \rangle$ ::= $\langle factor \rangle$
| $\langle term \rangle$ '*' $\langle factor \rangle$
| $\langle term \rangle$ '/' $\langle factor \rangle$;

Listing 1: IBAdapt Grammar.

variables IBAdapt supports, user-defined variables and system-defined variables. The system-defined variables can be read and modified within the scope of their policy definition. Both kind of variables are of three types: integers, floating point numbers, and string literals, as defined by $\langle assignment \rangle$ class. Each policy statement, $\langle policy \rangle$, includes a condition (or set of nested conditions) to be evaluated, and the actions that are executed once the condition is found true. A policy definition file may define several policy statements with each policy having multiple actions. A list of actions found executable according to the conditions is maintained by the interpreter in the sorting order of their definition in the policy file. An $\langle action \rangle$ is defined by an identifier denoting a function to be executed supplied by a user-provided script. Each action is executed only once even if it is referenced in several policies.

7. Implementation

We now describe our prototype implementation of a self-adaptive network architecture for IB subnets using the fat-tree network topology. The target of the prototype framework implementation is to enable self-adaptation of the fat-tree routing algorithm in a multi-tenant cloud environment configured on top of the IB interconnect.

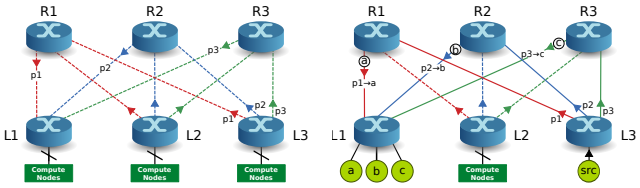
The implementation is based on the OFED IB stack and uses OpenSM as the subnet manager. We have implemented the adaptation framework using the C programming language and currently the framework is tightly coupled with the OpenSM implementation itself. However, the adaptation framework can easily be moved as an independent external system calling subnet interaction APIs, in the form of monitors and effectors, to integrate with the OpenSM. Our prototype system is built by extending OpenSM to include an optimization loop at fixed, but configurable, time periods. As mentioned in Section 3.1, the OpenSM periodically runs a light sweep to detect any changes in the topology. If such a change is detected that requires a network reconfiguration, OpenSM calls the routing algorithm to regenerate LFTs for the switches. We extended OpenSM's light-sweep functionality to handle optimizations, together with the faults, through our adaptation framework. At each adaptation interval, the adaptation engine of the framework evaluates the current subnet model, which is kept updated in a MariaDB database by the monitoring service, for any constraints violations or potential for further optimizations. The adaptation engine processes administrator defined policies and rules given as an input (using the IBAdapt interpretor), and corresponding actions are executed if given conditions are met. In our prototype implementation, each action generates a set of parameters, such as new traffic weights for the nodes and partitioning information. All generated parameters are then collectively provided as an input to our specially-designed fat-tree routing algorithm. The fat-tree routing algorithm makes use of the provided parameters, and generate a new set of routing tables to be potentially installed on the switches. The new routing is then evaluated by the routing evaluator for the potential benefits of installing the new routing, compared to the overhead involved in network reconfiguration. If adaptation is confirmed worthy, the adaptation planner and the executor plans and executes the actual routing table reconfiguration on the IB switches, respectively.

The parameter generation for the fat-tree routing is based on optimization criteria stemming from different challenges related to the realization of effective cloud computing using HPC interconnects. These challenges include efficient load balancing, tenant performance isolation, and fast network reconfiguration. Our prototype optimization engine uses some of our previous results addressing those challenges in the fat-tree topology [7], [22], [28]. However, note that the use of our optimization methods is just for demonstration purposes. The proposed self-optimization framework is generic, based on condition-action based policies, and providers are free to define any kind of rules, constraints, and actions as they deem suited using the IBAdapt language. In the same way, the self-adaptation framework is usable for any network topology, even though our prototype implementation is focusing on the fat-trees.

We now summarizes the optimization actions and generated parameters supported in our fat-tree routing algorithm.

7.1. Node Weights for Load Balancing

For an efficient HPC cloud, it is highly important that the load on network links is balanced and network saturation avoided. Network saturation can lead to low and unpredictable application performance, and a potential loss of profit for the cloud service providers. Furthermore, due to the dynamic workload admission, the network should be able to reconfigure itself according to current node traffic profiles. To tackle load-balancing issues, we use the notion of *weights* associated with each compute node, previously presented in [22]. These weights are used to take traffic profiles of the nodes into account when calculating routes in the fat-tree routing algorithm. The weight of a node reflects the degree of priority the flows towards a node receive when calculating routing tables. For example, a possible configuration could assign weights to the nodes in the range $[1, 100]$ depending on how much traffic a node is known to receive in the network. Such a scheme could assign $weight = 1$ for nodes that receive very little traffic (primarily traffic generators, for example), and $weight = 100$ for nodes receiving traffic near the link capacity. The values in between, $1 < x < 100$, will then reflect the proportion of traffic a node is expected to receive in the network. Weights are calculated employing a simple port data counter based scheme using observed incoming traffic saved in the historical data counters described in Section 5.1. When the adaptation system is booted, equal initial weights are assigned to all nodes. Later, previous weights are overridden by freshly calculated weights



(a) Phase I - Path Calculation (b) Phase II - Path Assignment

Figure 5: Two-phase leaf-switch based multipath routing.

at each adaptation interval. If B is the set of receive bandwidths for all the nodes measured over a time period, the weight for each node can be assigned in the range $[a, b]$ by using linear transformation as given by Equation 4.

$$W(x) = (x - a)\frac{b - a}{\max(B) - \min(B)} + a, \forall x \in B \tag{4}$$

The policies definition file defines the average threshold weight among nodes that needs to be crossed before an adaptation is performed. If the average change in weights over the last calculation is above the defined threshold, new weights are generated and provided to the *weighted* fat-tree routing for the routing calculation. Please consult [22] for details about weighted fat-tree routing.

7.2. Partitioning and Isolation Policies

Applications running on shared clouds are vulnerable to performance unpredictability and violations of the service level guarantees usually required for HPC applications. The performance unpredictability in a multi-tenant cloud computing system typically arises from utilization of shared compute and storage resources, server virtualization and network sharing. While the former can easily be addressed by allocating only a single tenant per physical machine, the sharing of network resources still remains a major performance variability issue. Intuitively, the network performance received by the applications of a tenant in a shared cloud is affected by the workload of other tenants in the system. The current IB implementation provides isolation mechanisms to enforce security through partitions, but does not provide those mechanisms at the routing level. This results in interference among tenant clusters. To tackle this problem, we presented partition-aware routing, pFTree, in [28], and later extended it to incorporate both provider-defined tenant-wise isolation policies and weighted load-balancing in [7]. The provider-defined isolation policies govern how the nodes in different partitions are allowed to share network resources with each other. These policies are defined using IBAdapt, and together with the partitioning information obtained through a tenant database, provided to the fat-tree routing.

7.3. Network Reconfiguration in Fat-Trees

The ability to efficiently reconfigure an interconnection network is an important feature that needs to be supported to ensure reliable network services for dynamic HPC clouds. In addition to handling faults, the adaptation engine needs reconfiguration to sustain network performance, and to satisfy runtime constraints defined by the provider policies. For instance, the routing function may need an update to optimize for a changed traffic pattern, or to maintain QoS guarantees. In IB reconfiguration, the original routing function needs to be updated to cope with the change. The main shortcoming of current reconfiguration techniques in IB is the costly re-routing for each reconfiguration event, making reconfigurations very expensive. The adaptation executor uses a metabase-aided network reconfiguration scheme, first presented in [24], to achieve fast reconfigurations for the fat-tree topologies. In the metabase-aided reconfiguration method, routing is divided into two distinct phases: calculation of paths in the topology, and assignment of the calculated paths to the actual destinations. For performance-driven reconfiguration, when reconfiguration is triggered without a topology change, the path calculation phase can be completely eliminated by using a metabase with stored paths from the first phase, hence saving routing time, which in turn substantially reduces overall network reconfiguration time. Two-phase leaf-switch based multipath routing on a small example fat-tree network is shown in Figure 5. In the first phase, multiple set of paths in the form of spanning trees are calculated for the destination leaf switches. However, the path calculation is done without considering the compute nodes attached to the leaf switches. Consider the example shown in Figure 5(a), we calculate three different paths towards leaf-switch *L1*. As there are three root switches in this topology, *R1-R3*, we can calculate three distinct paths towards *L1*, one through each root switch. For each of these paths, *p1*, *p2* and *p3*, a complete spanning tree in the topology can be calculated, rooted at the selected root switch. Each of the spanning trees, shown with differently colored dashed lines in the figure, gives one complete set of routes from the other two leaf switches (*L2* and *L3*) to *L1*. The actual assignments of the calculated paths to the destination compute nodes is done in the second phase, as shown in Figure 5(b).

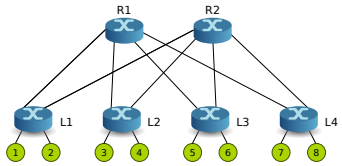
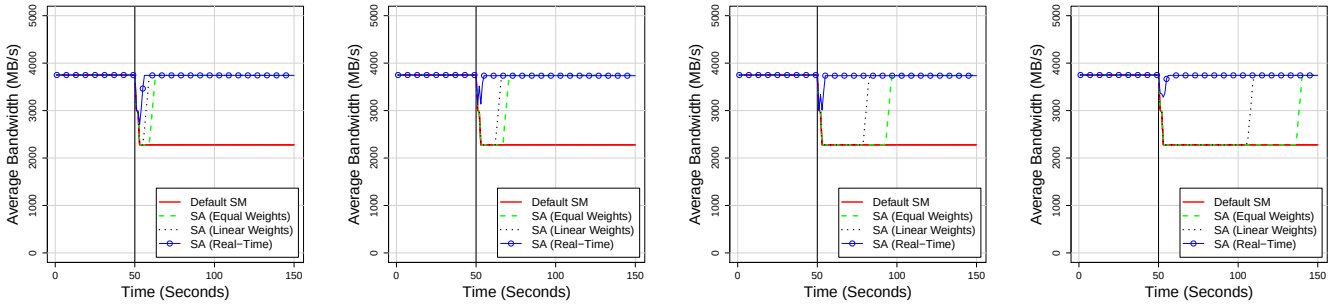


Figure 6: Topology for small-scale experiments.



(a) Counters = 10 (b) Counters = 20 (c) Counters = 50 (d) Counters = 100

Figure 7: Average flow bandwidth for different weight calculation methods and historical counters.

8. Evaluation

We run small-scale real-world experiments to evaluate our implementation of the self-adaptive network architecture on IB-based fat-trees. Our evaluation is to confirm the feasibility of the self-adaptive network architecture and the results are shown to demonstrate dynamic self-adaptation based on provider-defined policies and parameters. In particular, we show the influence of adaptation parameters on the reconfiguration decisions taken by the adaptation engine. However, for brevity and avoiding reiterations, we present only selected results for our implementation. Extensive results, not related to the self-adaptation, demonstrating solutions to several aforementioned HPC cloud challenges have been presented in our previous work on fat-tree networks [7], [24], [28].

8.1. Test Bed

We have conducted our experiments on a test setup with a two-level fat-tree topology consisting of six network switches and eight compute nodes, as shown in Figure 6. We use Sun DCS 36 QDR switches, and a mix of SUN Fire X2270 and HP ProLiant DL360p machines for the compute nodes. Each node runs Oracle Linux and is connected to the IB network using a Mellanox ConnectX-3 VPI adapter. We use the Mellanox OFED software stack on top of Oracle Linux to enable IB communication. All links are QDR capable with 40 Gb/s signaling rate using $4x$ link widths. The effective theoretical bandwidth per link is 32 Gb/s due to $8/10$ bit encoding employed by IB QDR technology.

We run each of the experiments with both the default OpenSM and the OpenSM equipped with our self-adaptation framework. For demonstration, we specifically use the fat-tree routing with the weighted routing capabilities (and corresponding action in the supplied policy definition file) so the traffic profiles obtained through monitoring can influence the routing in the network. Moreover, for each adaptation experiment, different policy parameters are used to evaluate the effect of user-defined policies on influencing how fast and how often the network reconfiguration takes place. In particular, we use different numbers of historical data counters for calculating network profiles, different weight prediction methods, and various weight thresholds in the adaptation policy definition files. The OFED stack's *ib_write_bw* utility is used to run traffic flows in the IB subnet and record their throughputs over time. Note that the time is divided into adaptation units, configurable in the policy definition file. In our case one unit equals one second.

8.2. Adaptation Time

In this experiment, we start with several non-interfering traffic flows in the topology using *ib_write_bw*. After a certain time (50 units), we start more flows so that contention is created between flows in the network reducing average bandwidth per node. However, as the self-adaptation framework is equipped with the weighted fat-tree routing, the sharing of intermediate

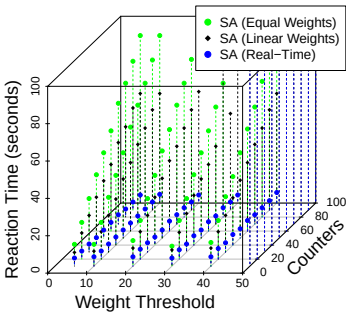
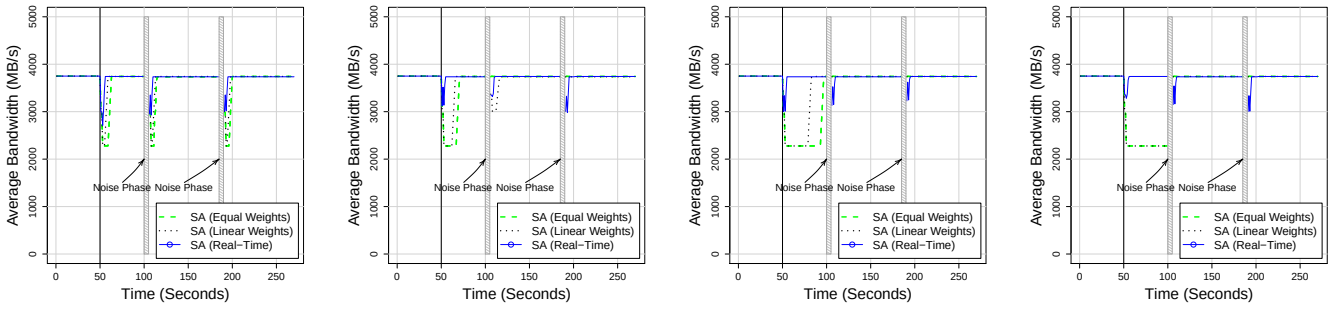


Figure 8: Reaction time using weighted fat-tree routing.



(a) Counters = 10 (b) Counters = 20 (c) Counters = 50 (d) Counters = 100

Figure 9: Average flow bandwidth in the presence of noise phases.

links between flows can be removed by re-routing triggered by the adaptation system. The adaptation times for different weight calculation methods and four different numbers of historical counters, configured through the policy file, are shown in Figure 7. The default SM does not reconfigure the routing so average network bandwidth, shown as a solid red line, is dropped to almost 50% of the original bandwidth when the new flows are added (Figure 7(a)-7(d)). For the self-adaptive framework implementation, we use three different methods for weight prediction, as described in Section 5.1. When only 10 historical counters are used, all three weight prediction methods, Equal Weights (dashed green line), Linear Weights (dotted black line), and Real-Time (circled blue line), react quite quickly when new flows are added, as shown in Figure 7(a). However, when we use 100 historical counters, as given in Figure 7(d), only the real-time weight prediction method, which work on the last counter only, restores the original bandwidth expeditiously. Equal weights prediction takes about 90 time units before the average bandwidth improves. This is because, the higher the number of historical data counters, the lesser is the effect of the last counter on the predicted weight, as all counters have the same preference in calculation.

Figure 8 summarizes the reaction time of our self-adaptation framework with different weight thresholds and a varying number of data counters used in the weight prediction. As shown in the figure, the reaction times for all prediction methods are not very much affected by the weight thresholds until it is increased to 50. This is because in our small-scale test setup, the calculation of weights is based on flows that are either running or stopped, giving discrete weight values to the nodes. In a bigger real-world setups, where counters are accumulated over time, the weight calculation may result in more disperse weights influencing the self-adaptation with the weight thresholds as well. However, note that when we use a threshold equal to 50, none of our methods are able to converge in 100 time units showing the influence of weight thresholds on the self-adaptation. Note that in our system the time for detecting a change in the network is based on the configuration of time IB subnet manager do a *light-sweep* of the subnet. The light-sweep period is configurable in the IB subnet manager. The lower values however brings more configuration overhead and the risk of very high values is slow reaction. The sweet spot of granularity of the two is an interesting area to explore. However, this is very application-specific and would not be suitable to tune in a generic architecture, as we have presented in this paper.

8.3. Intermittent Noises

As demonstrated in the last experiment, the self-adaptation system reacts fast when the real-time weight prediction method is used. However, the real-time weight prediction method results in quick influence in case of intermittent noises and may also trigger far too many network reconfigurations. To demonstrate this property, we extend the experiments used in Section 8.2 to include two noise phases at predefined intervals. In each of the noise phases (5 time units in length each), we temporarily reverse the network communication roles of nodes so the receivers become traffic generators and vice versa. The results are shown in Figure 9. Figure 9(a) shows average bandwidth for flows when 10 historical counters are used. The noise phase inductions are shown as dashed gray areas in the figure. As the number of counters are low, we see that each of our weight prediction methods is affected by both the noise phases, and overhead network reconfigurations are triggered, influenced by the noise. However, when the number of counters are increased to 20, as given in Figure 9(b), the equal weight prediction method is not influenced. Moreover, the linear weight prediction is also affected only by the first noise case. With more historical data counters in use (Figure 9(c) and Figure 9(d)), only real-time weight prediction is affected. The desired effect in those cases is that the adaptation system should not get affected by the intermittent noises and traffic profiles are calculated on the basis of sustained traffic patterns. This experiment further shows the importance of the self-adaptation system and also validates that the configured adaptation policies can greatly influence the adaptation system.

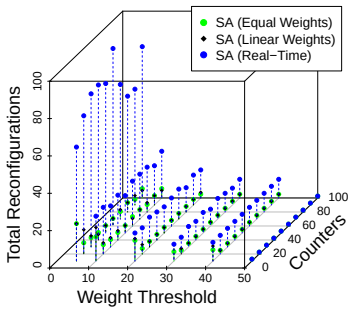


Figure 10: Total number of reconfigurations triggered by the adaptation engine.

Figure 10 shows the total number of reconfiguration events triggered by the adaptation engine with different policy parameters. As shown in the figure using blue dots, the total number of reconfigurations triggered by the adaptation engine using the real-time prediction method is very high with weight threshold equals to 5 (in the range of 61-95). Except for a very high weight threshold, 50, when only a few reconfigurations are triggered, the other two weight prediction methods equal weights (green dots) and linear weights (black diamonds) trigger a substantially low number of reconfigurations (as few as 1 or 2 for higher numbers of counters). This, however, does not show the total number of reconfigurations actually performed. The routing evaluator component of the adaptation layer may turn down many such reconfiguration triggers depending on the estimated overhead and results from the optional simulation engine.

8.4. Real world applications

In order to demonstrate benefits of a self-adaptive network architecture for real-world high-performance computing applications, we have created small-scale experiments using NAS parallel benchmark (NPB) suite [29]. For each of these experiments, we use a script to submit jobs to the runtime. Each job is run 100 times and the cumulative job completion time is reported. In order to demonstrate routing effectiveness on our small cluster, we added one additional node connected to each of the leaf-switches in our test topology, as shown in Figure 6, and start with a sub-optimal routing. The jobs are run on four selected nodes, one on each leaf switch. The routing algorithm is configured to optimize routing based on physical isolation of the partition running the job, once this metric is triggered through another script. The NPB benchmarks are derived from computational fluid dynamics (CFD) applications and consist of several kernels. We use Conjugate Gradient (CG), Integer Sort (IS), and Multi-Grid (MG) benchmark kernels. Metric triggering script is configured with different time configurations ranging from 10 seconds to submit the node numbering of the job partition to the routing engine to the real-time submission as soon as the first job is started. As shown in Figure 11, improvements in cumulative job completion times using self-adaptive routing system depends on the application and its communication patterns as well as the configuration of metric triggering script. For instance, when metric reconfiguration script is configured for 10 second reaction time, the

CG job completion takes 22.8 seconds as compared to the real-time reaction configuration which brings the job completion times close to 14.88 seconds.

8.5. System and Reconfiguration Overhead

The computational overhead brought up by the self-adaptive system does not affect network or application performance in the Cloud as it is contained only in the subnet manager node. However, when the self-adaptive system decides to do a routing reconfiguration, reconfiguration cost is incurred. In general, the total system overhead of our self-adaptive system depends on two factors. The re-routing time and the reconfiguration cost in the network. Re-routing is needed to setup the network paths according to the updated subnet model represented by, for instance, traffic profiles of the nodes or by modified topology. Once the new routing tables (in the form of LFTs) have been calculated, the reconfiguration cost depends on the time to send subnet management packets (SMPs) to the switches containing LFT block updates, and the number of dropped packets during the reconfiguration. The actual number of LFT block updates depends on the specific LIDs that are modified, which can not be known in advance. Please note that in a small network with just 6 switches and 8 nodes, all updates will be contained in a single LFT block and hence the reconfiguration time will be in the order of milliseconds. The routing time, however, can be a significant factor. For performance based reconfigurations, routing time can be significantly reduced employing a metabase-aided routing method as described in Section 7.3. Having pointed out that, reconfiguration overhead depends on the policies of the adaptation engine and the frequency of network adaptations. The architecture presented in this work targets providing framework for such implementations but does not limit the scope to a particular implementation of the adaptation engine.

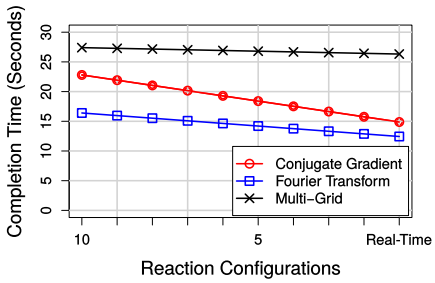


Figure 11: Job completion times for NAS Parallel benchmarks using a configurable self-adaptive routing system.

9. Related Work

The use of an IB interconnect for building HPC clouds has recently gained attention in the HPC community [10], [30]. In particular, advancements in hardware virtualization techniques, such as *single root I/O virtualization* (SR-IOV), have opened doors to efficient execution of HPC applications in the cloud [11], [31], [32]. More recently, containerized execution of HPC applications has also been studied [33]. The current approaches, however, are limited to the realization of HPC clouds without support for dynamic cloud optimizations, which makes the target of our work. The non-flexibility of IB to quickly adhere to varying configurations make IB less suitable for the commercial cloud solutions as well. Even though some cloud providers, for instance Microsoft Azure, OrionVM, and ProfitBricks, provides compute instances connected using an IB interconnect, they too do it through dedicated hardware instances and broader IB adaption is missing.

Dynamic load-balancing in IB, that is to enable the routing to react to changing network conditions, relies on adaptive routing mechanisms [14]. Adaptive routing typically utilizes global routing information [34] or some kind of feedback notifications [35], [36] to dynamically select routes for packets at the switches so that congested parts of the network can be avoided. Mellanox Traffic-Aware Routing (TARA) [37] is a proprietary routing algorithm which optimizes routing by taking into consideration the current topology and various services and active applications. Adaptive routing, although promising higher degree of network utilization and load balancing, increases routing overhead, and may introduce out-of-order packet deliveries, as well as degraded performance for window-based protocols [38]. Moreover, such techniques are reactive so preemptive network optimizations using historical traffic data are not possible. Other load balancing techniques using traffic-oblivious routing are optimized assuming all-to-all communication. When nodes exhibit distinct data communication patterns, however, optimizing all-to-all communication results in sub-optimal routing.

Recently, Schedule-aware Routing (SAR) [39] is proposed for dynamic routing optimization. The SAR performs path reconfigurations based on concurrently running applications. The approach, though very interesting, is specifically designed for batch systems and is tightly integrated with the DFSSSP routing algorithm. The DFSSSP routing has been extended

to include information about batch job locations to optimize the path calculation for intra-job paths. Our approach is independent of the routing algorithm and is not limited to optimizations based on traffic profiles or node locations. The provider-defined policies make it possible to include self-optimization of the network based on any criteria suitable for a system. In addition, our self-optimization framework can also use a specialized routing technique like SAR for a particular implementation.

A common approach to organize self-adaptive systems is through a *feedback loop* [40], sometimes also referred to as *autonomic control loop* in the context of *autonomic computing* [41]. The main objective of self-adaptation is to empower computing systems with autonomic properties, often denoted as the *self-** properties. These properties include *self-configuration*, *self-healing*, and *self-optimization*, among others. In addition, a prerequisite of implementing the autonomic behavior in a system is that it should have *self-awareness* and *context-awareness*. That is, the system should have adequate mechanisms to gain knowledge about its internal states and the external operating environment. Feedback loops are often implemented by a *Monitor-Analyze-Plan-Execute* based adaptation sequence. In MAPE-based systems, the system being adapted, also called a *managed system*, is continuously monitored and analyzed, and an adaptation strategy is planned and executed, when a symptom stopping the system to achieve the desired behavior is detected [42], [43], [44]. The rainbow framework [21] is an architecture based self-adaptation framework using a feedback-control loop to support adaptation strategies. Our work is inspired by the Rainbow framework but does not use any of its implementations because the framework is not well-maintained, is specific to Java-based applications, and is found tightly coupled with the Stitch language [45]. Furthermore, the only available implementations target web-based applications specifically. Note that a straight-forward implementation of self-adaptive systems to the HPC clouds is not possible due to the challenges we mention in Section 2. Hence, the target of this work is not improving the state-of-the-art in self-adaptive systems, albeit, we enables the the use of self-adaptive systems in high-performance lossless interconnection networks that are traditionally statically routed.

10. Conclusions and Future Directions

In this work, we identify that unavailability of a self-adaptive network impedes realization of efficient HPC clouds. We propose a framework for a self-adaptive network architecture for HPC clouds based on lossless high performance interconnection networks. Our solution consists of a feedback control loop that dynamically adapts to the varying traffic patterns, current resource availability, workload distributions, and also in accordance with the service provider-defined policies.

The self-adaptive network framework presented in this paper can be thought of as a *proof-of-concept* with concrete research opportunities available for its application with a variety of workloads. For instance, communication efficiency plays an important role in the performance of the big data applications, in particular for the phases where data needs to be *shuffled* between worker nodes in bursts [46]. Using a self-adaptive network architecture, a *routing-aware* big data framwork can be implemented that can work in coordination with the networking system to improve job scheduling decisions. In this connection, we are exploring an extension proposal to the remote direct memory access (RDMA)-based Apache Hadoop implementation, maintained by the Ohio State University [47]. In addition, emerging deep learning workloads can also significantly benefit from a self-adaptive network architecture as network performance significantly impacts the performance of distributed deep learning [48].

References

[1] T. Sterling and D. Stark, "A high-performance computing forecast: partly cloudy," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 42–49, 2009.

[2] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *Cloud computing*. Springer, 2009, pp. 115–131.

[3] A. Gupta and D. Milojicic, "Evaluation of HPC Applications on Cloud," in *6th Open Cirrus Summit (OCS), 2011.*, 2011.

[4] P. Bientinesi, R. Iakymchuk, and J. Napper, "HPC on competitive cloud resources," in *Handbook of Cloud Computing*. Springer, 2010, pp. 493–516.

[5] (2015) InfiniBand Architecture Specification: Release 1.3. Http://www.infinibandta.com/.

[6] Top 500 Super Computer Sites. Accessed January 5, 2018. [Online]. Available: http://www.top500.org/

[7] F. Zahid, E. G. Gran, B. Bogdański, B. D. Johnsen, and T. Skeie, "Efficient Network Isolation and Load Balancing in Multi-Tenant HPC Clusters," *Future Generation Computer Systems*, 2016.

[8] A. Gupta and D. Milojicic, "Evaluation of hpc applications on cloud," in *Sixth Open Cirrus Summit (OCS), 2011.* IEEE, 2011, pp. 22–26.

[9] (2016) HPC Market Model and Forecast: 2016 to 2021 Snapshot Analysis. Http://www.intersect360.com/industry/reports.php?id=148, accessed April 15, 2018. [Online]. Available: http://www.intersect360.com/industry/reports.php?id=148

[10] V. Mauch, M. Kunze, and M. Hillenbrand, "High performance cloud computing," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1408–1416, 2013.

[11] J. Zhang, X. Lu, S. Chakraborty, and D. K. D. Panda, "Slurm-V: Extending Slurm for Building Efficient HPC Cloud with SR-IOV and IVShmem," in *European Conference on Parallel Processing (EuroPar '16)*. Springer, 2016, pp. 349–362.

[12] F. Zahid, "Network Optimization for High Performance Cloud Computing," Ph.D. dissertation, Faculty of Mathematics and Natural Sciences, University of Oslo, 2017.

[13] R. Perlman, "An algorithm for distributed computation of a spanning tree in an extended LAN," in *ACM SIGCOMM Computer Communication Review*, vol. 15, no. 4. ACM, 1985, pp. 44–53.

[14] J. C. Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato, "Supporting fully adaptive routing in infiniband networks," in *International Parallel and Distributed Processing Symposium (IPDPS '03)*. IEEE, 2003, pp. 10–pp.

[15] I. of Electrical and E. Engineers, "802.1 Q/D10, IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks." 1997.

[16] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[18] F. Petrini and M. Vanneschi, "k-ary n-trees: High performance networks for massively parallel architectures," in *Proceedings of the 11th International Parallel Processing Symposium, 1997.* IEEE, 1997, pp. 87–93.

[19] S. R. Öhring, M. Ibel, S. K. Das, and M. J. Kumar, "On generalized fat trees," in *Proceedings of the 9th International Parallel Processing Symposium, 1995.* IEEE, 1995, pp. 37–44.

[20] OpenStack: Open Source Cloud Computing Software. Accessed August 1, 2017. [Online]. Available: http://www.openstack.org/

[21] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[22] F. Zahid, E. G. Gran, B. Bogdański, B. D. Johnsen, and T. Skeie, "A Weighted Fat-Tree Routing Algorithm for Efficient Load-Balancing in InfiniBand Enterprise Clusters," in *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2015.*, 2015, pp. 35–42.

[23] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, no. 1, 2007.

[24] F. Zahid, E. G. Gran, B. Bogdański, B. D. Johnsen, T. Skeie, and E. Tasoulas, "Compact Network Reconfiguration in Fat-Trees," *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4438–4467, 2016.

[25] E. G. Gran and S.-A. Reinemo, "InfiniBand congestion control: modelling and validation," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, 2011, pp. 390–397.

[26] OMNeT++ InfiniBand Flit Level Simulation Model. Accessed August 1, 2017. [Online]. Available: http://ch.mellanox.com/page/omnet

[27] T. Schneider, T. Hoefler, and A. Lumsdaine, "ORCS: An oblivious routing congestion simulator," *Indiana University, Computer Science Department, Tech. Rep*, 2009.

[28] F. Zahid, E. G. Gran, B. Bogdański, B. D. Johnsen, and T. Skeie, "Partition-Aware Routing to Improve Network Isolation in InfiniBand Based Multi-tenant Clusters," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015.*, 2015, pp. 189–198.

[29] NAS Parallel Benchmarks. Accessed August 1, 2017. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html/

[30] M. Hillenbrand, V. Mauch, J. Stoess, K. Miller, and F. Bellosa, "Virtual InfiniBand clusters for HPC clouds," in *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*. ACM, 2012, p. 9.

[31] M. Musleh, V. Pai, J. P. Walters, A. Younge, and S. Crago, "Bridging the virtualization performance gap for HPC using SR-IOV for InfiniBand," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 627–635.

[32] J. Zhang, X. Lu, M. Arnold, and D. K. Panda, "MVAPICH2 over OpenStack with SR-IOV: An Efficient Approach to Build HPC Clouds," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015.* IEEE, 2015, pp. 71–80.

[33] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 74–81.

[34] Z. Ding, R. R. Hoare, A. K. Jones, and R. Melhem, "Level-wise scheduling algorithm for fat tree interconnection networks," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 96.

[35] P. Geoffray and T. Hoefler, "Adaptive routing strategies for modern high performance networks," in *16th IEEE Symposium on High Performance Interconnects (HOTI '08)*. IEEE, 2008, pp. 165–172.

[36] E. Zahavi, I. Keslassy, and A. Kolodny, "Distributed adaptive routing convergence to non-blocking DCN routing assignments," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 1, pp. 88–101, 2014.

[37] Unified Fabric Manager (UFM®) Software for Data Center Management. Accessed August 1, 2017. [Online]. Available: http://www.mellanox.com/related-docs/prod_management_software/PB_UFM_Software.pdf

[38] C. Gomez, F. Gilabert, M. E. Gomez, P. Lopez, and J. Duato, "Deterministic versus adaptive routing in fat-trees," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2007.* IEEE, 2007, pp. 1–8.

[39] J. Domke and T. Hoefler, "Scheduling-aware routing for supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, 2016, p. 13.

[40] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[41] P. Horn, "Autonomic computing: IBMś Perspective on the State of Information Technology," 2001.

[42] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70.

[43] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, 2006.

[44] P. Vromant, D. Weyns, S. Malek, and J. Andersson, "On interacting control loops in self-adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 202–207.

[45] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[46] H. Herodotou, "Hadoop performance models," *arXiv preprint arXiv:1106.0940*, 2011.

[47] High-Performance Big Data (HiBD). Accessed May 21, 2018. [Online]. Available: http://hibd.cse.ohio-state.edu/

[48] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *arXiv preprint arXiv:1802.09941*, 2018.