Surviving Failures in Bandwidth-Constrained Datacenters

Peter Bodík
Microsoft Research
peterb@microsoft.com

Ishai Menache
Microsoft Research
ishai@microsoft.com

Mosharaf Chowdhury
UC Berkeley
mosharaf@cs.berkeley.edu

Pradeepkumar Mani
Microsoft
prmani@microsoft.com

David A. Maltz
Microsoft
dmaltz@microsoft.com

Ion Stoica
UC Berkeley
istoica@cs.berkeley.edu

Abstract— Datacenter networks have been designed to tolerate failures of network equipment and provide sufficient bandwidth. In practice, however, failures and maintenance of networking and power equipment often make tens to thousands of servers unavailable, and network congestion can increase service latency. Unfortunately, there exists an inherent tradeoff between achieving high fault tolerance and reducing bandwidth usage in network core; spreading servers across fault domains improves fault tolerance, but requires additional bandwidth, while deploying servers together reduces bandwidth usage, but also decreases fault tolerance. We present a detailed analysis of a large-scale Web application and its communication patterns. Based on that, we propose and evaluate a novel optimization framework that achieves both high fault tolerance and significantly reduces bandwidth usage in the network core by exploiting the skewness in the observed communication patterns.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

Keywords

datacenter networks, fault tolerance, bandwidth

1. INTRODUCTION

Users increasingly rely on online services to manage their computation, storage, and communication requirements. Downtimes hurt them dearly. In 2010, North American businesses collectively lost an estimated $26.5 billion in revenue due to partial or complete outage of services [1]. According to [2], unplanned outages cost $5,000 per minute, on average.

Previous work on designing network topologies [16, 29] and resource allocation mechanisms [6, 17] focused on tolerating failures of network components by providing multiple paths between server pairs and on ensuring predictable network behavior through better bisection bandwidth and reservation. However, in practice, despite redundancy, failures and maintenance of network and power components cause sections of datacenters with tens to thousands of servers to become unavailable [4, 12, 14, 15]. At the same time, datacenter networks can get congested, causing spikes in network latencies. In both cases, Internet services become unavailable and/or unresponsive. Although fault tolerance and bandwidth could be improved by investing more in power and network infrastructure, such actions can be performed only at a slow time-scale, typically during datacenter construction.

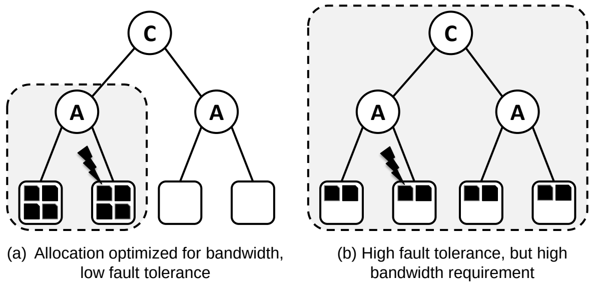

C
A
A
(a) Allocation optimized for bandwidth, low fault tolerance
C
A
A
(b) High fault tolerance, but high bandwidth requirement

Figure 1: Simple network topology with two aggregation switches and four racks illustrating the tradeoff between bandwidth usage (pack servers together, (a)) and fault-tolerance (spread servers across racks, (b)). Grayed boxes indicate parts of the cluster network each allocation is using. Assuming only racks as fault domains, (a) has worst-case survival of 0.5 (four of the eight servers survive a failure of a rack), while (b) has worst-case survival of 0.75 (six of the eight servers survive).

In this paper, we present practical solutions that consider the requirements of the underlying applications and help right away in any network. Given a fixed network topology, our goal is to improve the fault tolerance of the deployed applications *and* reduce the bandwidth usage in the core simply by optimizing the allocation of applications to physical machines.

However, there is conflict between simultaneously improving the fault tolerance and reducing the bandwidth usage of cloud-based services. We can improve the fault tolerance by spreading machines of a particular application across many fault domains, thus reducing the impact of any single failure on the application. However, this allocation requires more bandwidth in the core of the network and thus could be prone to congestion, as illustrated in Figure 1(b). On the other hand, to reduce communication through oversubscribed network elements, we can allocate all machines of the application on the same rack or under one aggregation switch, as proposed in [6, 17]. This allocation, however, reduces the fault tolerance of the application; a failure of the top-of-rack (TOR) or aggregation switch, can make the application unavailable, as illustrated in Figure 1(a).

Figure 2 demonstrates the perils of trying to independently opti-

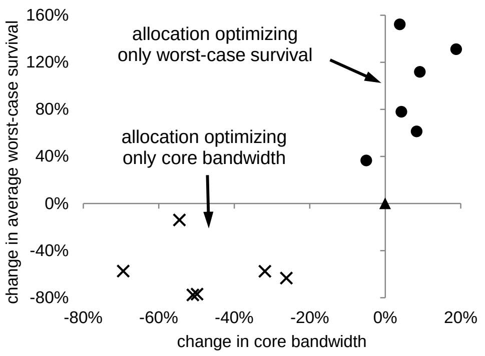


Figure 2: This plot shows the changes in core bandwidth and average worst-case survival for six different datacenters, after applying BW-only optimization (minimum k-way cut, crosses) and FT-only optimization (spreading servers, circles). For both optimizations, one of the metrics improves significantly, but the other one actually degrades.

mize for either fault tolerance or bandwidth usage. Using data from six large-scale production datacenters, we applied both bandwidth-only optimization (using minimum k-way cut of the communication graph, similar to [28][1]) and also fault tolerance-only optimization (described in Section 4). For each service in a datacenter, we measure fault tolerance using *worst-case survival* – fraction of machines of the service that remain available during a single, worst-case failure. Both algorithms significantly improve the metric they optimize for, but actually degrade the other metric. Thus, our goal is to *design algorithms that produce server allocations which achieve both high fault tolerance and reduce bandwidth requirements on the core of the network.*

Our work is motivated and evaluated on actual empirical data from large-scale production datacenters running Bing.com, each running thousands to tens of thousands of servers. The following are crucial observations from our study of the communication patterns of software services in these datacenters:

- The communication matrix between different services is very sparse – only 2% of all pairs of services in the datacenter communicate. In addition, the communication pattern is extremely skewed – the top 1% of all services generate 64% of all the network traffic.
- The fault domains of power equipment do not necessarily match the fault domains of networking equipment, thus complicating the fault-tolerant server allocation.

These observations motivate the formulation of our optimization framework that provides a principled way to explore the tradeoff between reducing bandwidth usage and improving fault tolerance. The problem of improving fault tolerance and the problem of reducing bandwidth usage are both NP-hard and hard to approximate (see Section 4 for details). With that in mind, we formulate a related convex optimization problem that incentivizes spreading machines of individual services across fault domains. In addition, we add a penalty term for machine reallocations that increase bandwidth usage. Intuitively, the combination of these two components exploits the communication skewness, e.g., spreading machines of low-communicating services improves their fault tolerance without significantly affecting bandwidth. Appealingly, the resulting algorithm is easy to integrate and computationally efficient.

Our algorithm achieves $20\% - 50\%$ reduction in bandwidth usage in the core of the network, while at the same time improving

[1]We note that [28] does not consider the joint bandwidth-fault tolerance optimization.

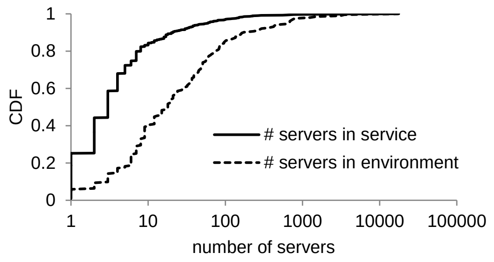


Figure 3: CDF of number of servers in individual services and number of servers in environments.

the average worst-case survival by $40\% - 120\%$. Moreover, because this algorithm minimizes the number of machine moves, it can achieve significant improvements by moving only 20% of machines in the datacenter. Our simulations show that these improvement in fault tolerance can reduce the fraction of services affected by potential hardware failures by up to a factor of 14. In addition, we present a solution based on a minimum k-way cut algorithm, which in many cases achieves reduction in bandwidth by additional $10 - 20$ percentage points.

The contributions of this paper are three-fold:

- **Measurement study.** We present detailed analysis of the structure of a modern, large-scale Web application and the observed communication patterns between the individual services comprising this application (Section 2). While there exist studies on datacenter communication patterns [8, 20], our study concentrates on the traffic at the application level.
- **Algorithms.** To the best of our knowledge, the joint fault tolerance and bandwidth optimization has not been considered in previous studies. Although fault tolerance and bandwidth are typically contradicting objectives, our optimization framework (Sections 3–4) achieves both high fault tolerance and significant reduction in bandwidth usage in the core. Our algorithms exploit the observed skewed communication pattern, scale to large datacenters, and can accommodate arbitrary fault domain structure.
- **Methodology.** We present methodology for stressing server placement algorithms and evaluating their effectiveness. In particular, we introduce a novel framework which utilizes empirical measurements for solving the joint optimization of worst-case survival of software services, machines moves, and bandwidth usage.

2. MOTIVATION AND BACKGROUND

In this section we analyze Bing.com, a large-scale Web application running in multiple datacenters around the world and use it to illustrate the issues described in the Introduction. This analysis is based on data from production clusters with thousands to tens of thousands of servers and also from a large-scale pre-production cluster consisting of thousands of machines. While this analysis is based on a single application, we note that Bing is not simply a "Web index", as it contains many components very similar to typical Web applications. We also highlight the insights we learned and we extract the key parameters, like *skew*, that impact the effectiveness of our algorithms. This enables others to apply our observations and the proposed solutions to other large scale datacenter and cloud computing environments.

We describe the application characteristics in Section 2.2, highlight the sparseness and skewness of communication patterns in 2.3, and characterize the complexities of datacenter fault domains in 2.4. Finally, we describe additional scenarios and implications of this study on our optimization framework.

Term	Definition
Logical machine	Smallest logical component of a web application
Service	Service consists of many logical machines executing the same code
Environment	Environment consists of many services
Physical machine	Physical server that can run a single logical machine
Fault domain	Set of physical machines that share a single point of failure; one machine can belong to multiple domains

Table 1: Definitions

2.1 Definitions

For the purpose of this paper, we assume a datacenter with a known network topology running one or many independent Web applications. A general web application is composed of many *environments* (or frameworks), such as front-end, back-end, or an offline, data-processing framework. Each environment is composed of one or more *services*; for example, the front-end environment might consist of services such as the web servers, caches and other logic that is necessary when processing user requests. Another example of an environment is a MapReduce-like system, with services for job execution or job submission UI. Each service requires a certain number of *logical machines* to support the workload and each logical machine is deployed to a *physical machine* (or a *server*) in the datacenter. We assume only one logical machine per physical machine and describe an extension to a more general scenario in Section 6. Reassigning a logical machine between physical machines is expensive for various reasons. It could take tens of minutes for machines to warm up their caches of in-memory data needed to answer production requests; furthermore, reimaging and making data sets available to the new logical machine takes time and consumes network resources. We assume the network can be modeled as a hierarchical tree[2]; see Section 5 for details. Table 1 summarizes the terms used in this paper.

A *fault domain* is a set of physical machines that share a single point of failure, such as a top-of-rack switch or a circuit breaker. Upon a failure, the servers in the corresponding fault domain become either unavailable or their capacity is reduced. The fault domains can overlap in non-trivial patterns as described in more detail in Section 2.4.

2.2 Service Characteristics of Bing

Bing is a large-scale Web application deployed in many datacenters around the world, serving mostly interactive user traffic with a significant offline, data processing workload. The number of services in Bing's datacenters is in the order of a thousand. Figure 3 shows the distributions of the number of servers for individual services and entire environments. The services are mostly small – 80% of them have less than ten logical machines, while a few services are very large. Similarly, most environments contain less than ten services, but some environments contain a few tens of services.

2.3 Communication Patterns

In this section, we present the analysis of communication patterns of Bing running in a preproduction environment consisting of thousands of servers. We trace communication between all pairs of servers and for each pair of services i and j, we compute the total number of bytes transmitted between this service pair, $R_{i,j}$, and also total traffic generated by individual services, T_i. We refer to (the collection of) $R_{i,j}$ as the *service communication matrix*.

[2]We note that even fat-tree-like networks can be modeled as a logical hierarchical tree with higher bandwidth at the root.

link utilization	>50%	>60%	>70%	>80%
aggregate months above utilization	115.7	47.5	18.3	6.2

Table 2: The aggregate time (during a single month) that all core links spent above certain utilization in one of our production datacenters. We cannot reveal the actual utilization of the links.

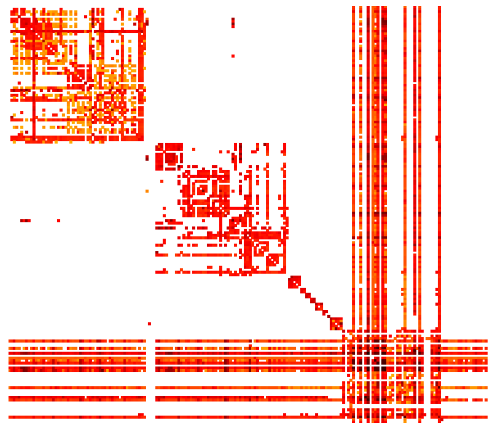


Figure 4: Visualization of communication matrix for a small subset of services using a heat map.

Datacenter network core is highly utilized. As shown in [8], the links in the network core are the most utilized in a datacenter (compared to aggregation and TOR switches) and will thus benefit the most from reduction in usage. We have experienced several incidents in our datacenters where a single application was not aware of its placement in the network topology and started a large data transfer. This overloaded the core of the network and thus impacted the rest of the datacenter. Table 2 shows the amount of time the core links in one of our production datacenters spent at various utilizations during one month. While these statistics are not normalized by the number of links, there are many links that are highly utilized for long time intervals. By reducing the bandwidth usage in the core of the network, our algorithms thus reduce the probability of congestion and packet loss.

Traffic matrix is very sparse. A section of the communication matrix is visualized as a heatmap in Figure 4, where each row and column correspond to a service and color (light to dark) that encodes the amount of communication (white meaning no communication). Notice that the communication matrix $R_{i,j}$ is extremely sparse. Our pre-production environment runs on the order of thousand different services, however, out of all the possible service pairs, only 2% service pairs communicate at all. The remaining 98% of service pairs do not exchange any data.

Communication pattern is very skewed. We observe that the communication pattern of the 2% service pairs that do communicate is extremely skewed. The dashed line in Figure 5 shows the cumulative traffic generated by certain fraction of service pairs. 0.1% of the service pairs *that communicate* (including services talking to themselves) generate 60% of all traffic and 4.8% of service pairs generate 99% of all traffic. The solid line in Figure 5 shows the cumulative traffic generated by individual services. 1% of services generate 64% of traffic and 18% of services generate 99% of traffic. Services that do not require lot of bandwidth can be spread out across the datacenter, thus improving their fault tolerance.

Most bytes stay inside environments. In Figure 4, the services are ordered alphabetically by (environment,service) so that services in

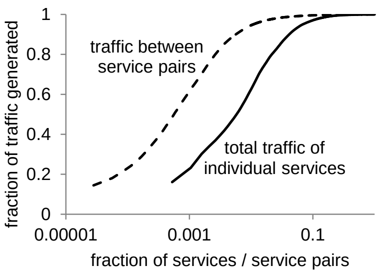


Figure 5: Cumulative fraction of total traffic (y-axis) generated by the top X% of services (x-axis, solid line) and top X% of service pairs (dashed line). The service and service pairs are in decreasing order of T_i, or $R_{i,j}$, respectively.

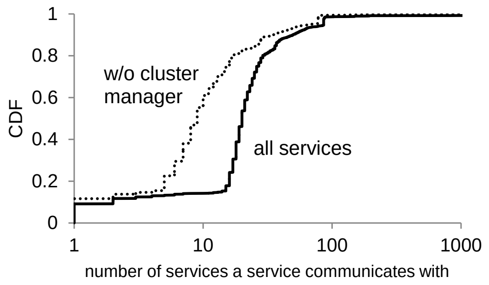


Figure 6: CDF of number of services an individual service communicates with (*i.e.,* degree of vertex in communication graph) for the full graph (solid line) and with services of cluster manager environment excluded (dotted line).

same environment appear next to each other. Most of the communication is along the diagonal (service talking to itself or other services in same environment); the two bigger squares on the diagonal represent two different environments. The vertical and horizontal lines correspond to cluster manager communicating with all machines. The majority of the traffic, 45%, stays within the same service, while 23% leaves the service but stays within the same environment, and 32% crosses environments.

Median service talks to nine other services. The communication matrix can be also represented as a graph, where vertices represent the services and edges represent the communication between service pairs. Figure 6 shows the CDF of the vertex degrees for the full communication graph (solid line). Approximately 15 services of the cluster manager environment communicate with almost all other services (see Figure 4), but this is not application traffic perse. We therefore also show the CDF of degrees with cluster manager services excluded (dotted line); median degree is 9.

Communicating services form small and large components. The complete communication graph has a single connected component, because the services are linked by common management, logging, and data analysis systems. However, after keeping only the edges that together generate 90% of the traffic (representing the most "chatty" service pairs), this graph falls apart into many disconnected components; see Figure 7 which highlights the major communication patterns. The circles correspond to individual services, and the edges to communication between service pairs. Notice that there are two large connected components – one with star-like topology, another with longer dependencies, and many other smaller components.

2.4 Failure Characteristics

In this paper we consider failures of datacenter hardware equipment that can cause unavailability of many physical servers. Examples of devices we consider are server enclosures (containing

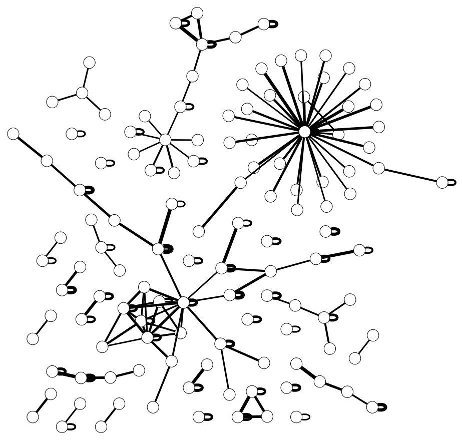


Figure 7: Communication pattern between individual services (represented by circles) after keeping only edges that generate 90% of the traffic. The width of the edges is proportional to the total bytes sent between the corresponding two components.

several servers), TOR and aggregation switches, containers, and power equipment. However, our approach is not limited to these failures and can be extended to any additional fault domains[3]. As demonstrated above, large-scale web applications are composed of many connected services and a failure of a critical service can cause failure of the whole website or significantly impact its functionality and degrade user experience.

Networking hardware failures can cause significant outages. The authors of [15] describe a large study of network failures in several Microsoft datacenters, many of them running Bing. We summarize their findings that motivate the need to improve application fault tolerance. While TOR switches and aggregation switches are fairly reliable (only 5% and 10% failure rates per year, respectively), the failures of TORs were responsible for majority of the observed downtime. While redundancy is often employed in datacenters for aggregation switches or access routers, [15] observes that redundancy reduces impact of failures on lost bytes by only 40% and thus does not prevent impact on applications. Even if these failures are rare, when large fraction of a service is deployed under the same aggregation switch, that service might become unavailable during a failure or have reduced throughput, with potentially high impact on user experience. Notice that the impact of maintenance is similar to a large-scale failure, as many servers might become unavailable. Nonetheless, since maintenance is scheduled in advance, our algorithms can be executed ahead of maintenance in order to automatically move applications out of the affected servers.

Power fault domains create non-trivial patterns. On top of fault domains created by network equipments, servers are also subject to failures in power distribution. While some of the power fault domains match the fault domains of the networking equipment, in general, the wiring of power to servers can create non-trivial patterns with power fault domains that do not match the domains created by switches and routers. For example, servers in multiple racks (but not all racks under an aggregate switch) can be connected to the same circuit, effectively putting them in the same fault domain. Therefore, distributing a service across these racks would not improve its fault tolerance. In some cases, a server and the corre-

[3]While correlated failures happen in practice, their probability is much lower than individual failures. We do not consider them in this paper.

Term	Definition
BW	Aggregate bandwidth usage on core links
WCS	Smallest fraction of machines that remain functional during a single failure in a datacenter
FT	Average WCS across all services
FTC	Fault tolerance cost function
NM	# of server moves to reach target allocation
Cell	Set of physical machines belonging to the same set of fault domains

Table 3: Definitions for the optimization framework.

sponding TOR and aggregate switches might be connected to the same power device, while in other cases they might be connected to three different power devices, again increasing the complexity of the fault domain mapping. These complexities arise from variation and evolution of power chassis architecture, container architecture and datacenter architecture as well as changes in the vendors. Due to their opaqueness, power fault domains are hard to consider for application developers and they have to be handled automatically by the server allocation algorithm.

2.5 Other Practical Considerations

Our core algorithms propose server allocations that reduce bandwidth usage in the datacenter and improve application fault tolerance. As described in Section 2.1, moving a logical machine to a new server is an expensive operation. We therefore consider machine moves as a crucial metric in our optimization framework.

We point out that our solution is orthogonal to previously mentioned solutions, such as a different network topology or bandwidth reservations. In fact, by reducing bandwidth usage we can make bandwidth available to new services that actually need it and could avoid costly and disruptive upgrades of network equipment.

There are several additional issues and scenarios that one may need to consider in practice. For example, services might have additional hard constraints on minimum worst-case survival or hardware configuration of the physical servers. We do not evaluate all such scenarios in this paper, however, our algorithms can be easily extended to handle such cases, as described in Section 6.

2.6 Implications for Optimization Framework

Our analysis of the communication patterns shows that even for a single large-scale website, only a small fraction of services communicate and the communication pattern of the ones that do is very skewed. Most of the traffic is generated by a small number of services and service pairs. *It is therefore feasible to spread a large fraction of services across the datacenter to improve their fault tolerance, without affecting much the bandwidth usage in the core of the network.*

While failures of networking and power equipment are relatively rare, given a bad allocation of services in a datacenter, even a single failure can have significant impact on a website. This motivates the use of the "worst-case survival" metric for fault tolerance (cf. Section 3) that captures the impact of the worst single failure on a service. *Our optimization framework has to consider the complex patterns of the power and networking fault domains, instead of simply spreading the services across several racks to achieve good fault tolerance.*

3. PROBLEM STATEMENT

In this section we provide the description of the problem. See formal mathematical details in Section 4.

3.1 Metrics

In this paper we consider the following three metrics.

Bandwidth. Core links are links from the root of the network topology tree to the second-level nodes. We use the sum of rates on the core links as the overall measure of the bandwidth usage at the core of network; this measure is denoted by **BW**.

Fault Tolerance. For every service, we define the Worst-Case Survival (WCS) to be smallest fraction of machines that remain functional during any *single* failure in the datacenter. For example, if the service is uniformly deployed across 3 racks, and we only consider TOR switch failures, its WCS is $2/3$. While more general survivability models have been proposed [26], we prefer this simpler and more practical metric. We use the average WCS across services as the measure for fault tolerance in the entire datacenter, and denote this average by **FT**. We note that the worst-case survival is directly related to application-level metrics such as throughput or capacity; *e.g.,* a service with WCS of 0.6 will lose at most 40% of its capacity during any single failure.

Number of Moves. The number of servers that have to be reimaged to get from initial datacenter allocation to the proposed allocation. We denote this number by **NM**. Table 3 contains a summary of our terminology.

Remark. Reducing the bandwidth usage in the core might actually lead to reduction of the WCS of a high-communication service by allocating its machines in the same rack. To prevent such situations for important services, we may extend our basic formulation, putting hard constraints on WCS for such services, as described in Section 6. Alternatively, FT of *small*, high-communication services can be improved by fully replicating them. The FT of *large*, high-communication services is inherently high since they are deployed across multiple fault domains anyway.

3.2 Optimization

The input for our optimization framework is a network topology with initial (or current) assignment of machines to services. Unless otherwise specified, in the remainder of the paper, we consider the following optimization problem

$$\begin{aligned} \text{Maximize} \quad & FT - \tilde{\alpha} BW, \\ \text{Subject to} \quad & NM \leq N_0, \end{aligned} \tag{1}$$

where $\tilde{\alpha}$ is a tunable positive parameter[4], and N_0 is an upper limit on the number of moves. This formulation accommodates a flexible tradeoff between BW and FT, manifested through the choice of $\tilde{\alpha}$. While we focus on (1) for concreteness, we note that our solution approach could be adapted to other formulations as well, such as minimizing NM under the constraint of certain improvements of BW and FT. See Section 6 for more details.

4. ALGORITHMIC SOLUTIONS

4.1 Problem Hardness

Even when considering the optimization of FT or BW in isolation, the resulting optimization problems are NP-hard and hard-to-approximate. Variants of the BW optimization problem have been broadly considered, see, e.g., [22] and references therein. These works obtain approximation algorithms with logarithmic guarantees, yet they restrict attention to *balanced* cuts (partitions of approximately equal size). Unfortunately, it is an open problem to find good approximation algorithms for the unbalanced case, which represents our setup (network partitions of unequal size). Already

[4]in inverse units of BW so that the objective function is unit-less.

a simple instance of the FT problem reduces to the maximum independent set problem. The known hardness bounds for that problem imply that no reasonable approximation factors are achievable [19]. On top of that, restricting the number of moves adds another level of complexity. Therefore, our focus in this paper is on designing reasonable heuristics that can consider all three metrics simultaneously. We note that we are not able to compare performance against the optimal solution, because the scale of the problem makes exhaustive search infeasible. Instead, we use other heuristics that do not limit the number of moves as performance benchmarks.

4.2 Overview of Solution Approach

In view of the hardness of our optimization problem, our algorithmic approach relies first on designing methods that individually optimize either BW or FT. We then combine the methods into algorithms that incorporate both objectives. Our solution roadmap is the following:

- We first introduce the notion of *cells*, which serve as the basic physical entities in our optimization framework, and allow for significant reduction in the size of the optimization problem.
- Instead of maximizing the intractable FT objective, we define an alternative cost function in Section 4.4, the *Fault Tolerance Cost* (FTC), which has “nice" convex structure. We show that FTC is in practice negatively correlated to FT, hence the minimization of FTC improves FT. The convexity of FTC allows us to iteratively improve the FTC with greedy moves that reduce the current value of the cost function, until reaching its global minimum (or very close to it)[5].
- Our basic method for optimizing BW is to perform a minimum k-way cut on the communication graph of the logical machines.
- CUT+FT+BW is a hybrid algorithm consisting of two-phases. First, we use a minimum k-way cut algorithm to compute an initial assignment that minimizes bandwidth at the network core. Second, we iteratively move machines to improve the FT. The drawback of this hybrid algorithm is that the graph cut procedure ignores the number of moves, and thus generates allocations that require moving almost all machines in the datacenter, which is undesirable during regular datacenter operation. Therefore, this algorithm serves mainly as a performance benchmark for the actual algorithm that we use.
- Finally, FT+BW also takes into account the NM metric. The algorithm does not perform the graph cut, but rather starts from the current allocation and improves performance through greedy moves that reduce a weighted sum of BW and FTC.

4.3 Formal Definitions

Cells. To reduce the complexity of our optimization problem due to overlapping and hierarchical fault domains, we partition the set of physical machines to *cells* (i.e., each machine belongs to exactly a single cell). A cell is a subset of physical machines that belong to exactly the same fault domains (therefore different cells do *not* overlap). For example, the datacenter in Figure 8 has four different cells, capturing all combinations of power sources and containers. Since all machines within a given cell are indistinguishable in terms of faults, it suffices to describe an assignment of machines by the set of variables $\{x_{n,k}\}$; the variable $x_{n,k}$ indicates the number of machines within cell n allocated to service k. We next define the BW and FT objectives with $\{x_{n,k}\}$ as the optimization variables. We emphasize that our cell-based formulation is general in the sense that it does not require any assumptions on the fault-domain topology.

Formal definition of BW and FT. To formally define BW, let $I(\cdot,\cdot)$ be the indicator function, whose inputs are cell pairs. For each such pair (n_1, n_2), $I(n_1, n_2) = 1$ if traffic from n_1 to n_2 (and vice-versa) traverses through a core link, and $I(n_1, n_2) = 0$ otherwise. Then BW, which is the total bandwidth consumption at the core is given by

$$BW(\mathbf{x}) = \sum_{n_1,n_2} \sum_{k_1,k_2} I(n_1, n_2) x_{n_1,k_1} x_{n_2,k_2} r_{k_1,k_2}, \qquad (2)$$

where $r_{k_1,k_2} = \frac{R_{k_1,k_2}}{s_{k_1} s_{k_2}}$ is the required bandwidth between a pair of machines from services k_1 and k_2 (s_k is the number of machines required by service k), and $\sum_{\cdot,\cdot}$ sums each pair only once.

To formally define FT, let $z_{k,j}(\mathbf{x_k}) \triangleq \sum_{n\in j} x_{n,k}$ be the total number of machines allocated to service k affected by fault j ($\mathbf{x_k}$ is the vector of allocations $\{x_{n,k}\}$ for service k). Our fault tolerance objective, FT, is formally given by

$$FT(\mathbf{x}) = \frac{1}{K} \sum_{k=1}^{K} \frac{s_k - \max_j z_{k,j}(\mathbf{x_k})}{s_k}, \qquad (3)$$

where K is the total number of services.

4.4 Basic Building Blocks

FTC and steepest descent moves for FT. As mentioned earlier, instead of maximizing (3), our fault tolerance optimization is achieved via the *minimization* of a cost function, termed FTC. This function is defined as follows:

$$FTC(\mathbf{x}) = \sum_j w_j \sum_k b_k \left(z_{k,j}(\mathbf{x_k})\right)^2, \qquad (4)$$

where $z_{k,j}$ and $\mathbf{x_k}$ are defined above, and b_k and w_j are positive weights that can be assigned to services and faults, respectively. For example, one could give higher weights to small services, as faults might have a relatively higher impact on them. More generally, these weights can be used to prioritize service placement according to its importance. The advantage of optimizing over (4) instead of optimizing directly over FT is that the former is a *convex* objective function. Convex functions are appealing, as “local" variable changes (e.g., swapping the physical machine allocation of two logical machines) that improve current cost, also get us closer to the *optimal* value (as opposed to arbitrary cost functions, in which such moves would converge to a *local* minimum). Intuitively, a decrease in FTC should lead to an increase in FT, as squaring the $z_{k,j}$ variables incentivizes keeping their values small, which is obtained by spreading the machine assignment across multiple fault domains; see Figure 8 for a simple example. Figure 9 shows the negative correlation between FTC and FT. While other convex functions could be used instead of (4), we use the sum-of-squares function, as it provides efficient and fast calculation of the delta-cost of machine reallocations, and performs very well in practice.

Our basic method for improving the FTC uses machine *swaps*, which simply switch the allocation of two logical machines, thereby preserving the number of machines assigned to each service (while increasing the NM count by two). To improve FT, we choose the swap that most improves FTC (details below). Accordingly, the swaps can informally be regarded as moves in the “steepest descent" direction.

[5]Convergence to global optimum is guaranteed for the continuous optimization problem provided that the step size is not too large. Here we deal with discrete variables, hence theoretically there is no guarantee to converge to the global optimum. Nevertheless, given the large scale of the problem, the “integrality gap" is negligible and the greedy moves are expected to converge to costs which are nearly optimal.

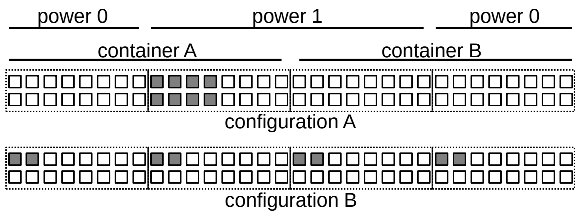


Figure 8: A simple example of a "datacenter" with two containers and two different power sources. In this example, the power fault domains overlap with the network (container) fault domains. The dotted rectangles represent the resulting cells and gray squares represent the allocated servers. Configuration A has a FTC of 128 and a FT of 0; configuration B spreads the machines across the fault domains, hence has a lower FTC of 64 and a higher FT of 1/2. Note that FTC and FT are negatively correlated.

Minimum k-way cut for BW. The basic procedure that we apply for minimizing the BW is based on minimum k-way cut, which partitions the logical machines into a given number of clusters. We often refer to this procedure as *min cut* or just *cut* for simplicity. In our setup, each of the clusters corresponds to a VLAN (we note that the traffic between different VLANs passes through core links). To utilize the cut for our purposes, we construct the datacenter communication graph as follows. Each node in the graph corresponds to a logical machine, and the weight of each edge in the graph is set as the average traffic rate between the two machines. We use Metis' graph partitioning function (gpmetis, see [21]), which allows us to define the size of each cluster within the partition. This feature is important, as the physical clusters need not be of equal size. Gpmetis outputs the set of logical machines which should belong to each cluster, while minimizing the total weight of the minimum k-way cut. Notice that the total weight is exactly BW, the bandwidth usage in the core under the suggested partition.

4.5 Algorithms to Improve both BW and FT

Based on the above building blocks, we first describe a set of algorithms that are not constrained by the number of moves. All the algorithms consist of two phases: in the first phase they minimize BW, followed by minimizing FTC with the option of penalizing swaps that increase the core bandwidth.

- **CUT+FT** : Apply CUT in the first phase, then minimize FTC in the second phase using machine swaps (no penalty for BW increase).
- **CUT+FT+BW** : As above, with the difference that in the second phase we add a penalty term for the bandwidth. More precisely, the delta cost of a swap is $\Delta FTC + \alpha \Delta BW$, where α is a weighting factor. By incorporating the penalty term we lose the convexity property, nevertheless the actual performance is not affected much.

Ignoring the number of moves for now, the algorithm CUT+FT+BW can be viewed as a plausible approach for solving our optimization problem (1): the first phase provides a low-bandwidth allocation, which is the starting allocation for the second phase. The second phase aims at reducing the FTC as much as possible, while being aware of costly bandwidth moves. The weight α specifies the required tradeoff between FT and BW[6]. Since the steepest descent direction is chosen, the cost FTC+αBW is reduced using a relatively small number of moves, which is yet another feature that prevents significant bandwidth increase.

[6]We note, however, that since we are minimizing FTC instead of maximizing FT, the α specified in the algorithm need not be equivalent to $\tilde{\alpha}$ in the objective (1). However, since α would in any case be adapted online by the perceived performance, this is not an issue.

The above algorithms can be used for the initial allocation of machines in a datacenter. However, during regular datacenter operation, these algorithms might not be applicable. This is because cut is not aware of the current machine assignment, and might therefore reassign almost all machines (e.g., if there are 100 containers, the probability that a logical machine will remain in the same container is 1%). To address this issue, we consider the following NM-aware algorithm:

- **FT+BW** : Starting from the initial allocation, do only the second phase of CUT+FT+BW.

Finally, we introduce an algorithm that directly exploits the skewness of the communication matrix.

- **CUT+RANDLOW** : Apply cut in the first phase. In the second phase, determine the subset of services whose aggregate bandwidth requirements are lower than others (the size of the subset is a parameter to the algorithm), then randomly permute the machine allocation of all the services belonging to the subset.

This algorithm takes the advantage of having services that do not consume much bandwidth. Consequently, even random "spreading" of these services could lead to significant FT improvements, without significantly affecting the bandwidth consumption.

4.6 Scaling to Large Datacenters

To scale the algorithms to large datacenters, we describe additional features of the two building blocks. Because there are many possible machine swaps, finding the actual best one would take a long time. Instead, we sample a large number of candidate swaps and choose the one that most improves FTC.[7] Due to the separable structure of FTC, we compute the improvement in FTC incrementally, allowing us to indeed examine a large number of swaps.

When performing the graph cut, the full graph representation, where each node represents a logical machine, often leads to an intractable optimization problem for our minimum k-way cut solver. To address that, we employ a *coarsening* technique, in which we group logical machines of the same service into a smaller number of representative nodes. Accordingly, the edge weight between each pair of such nodes becomes the sum of inter traffic rates between the logical machines of the two nodes.

5. EVALUATION

In this section we evaluate the algorithms described in Section 4. We first evaluate algorithms that optimize for bandwidth or fault tolerance (Section 5.3), then evaluate algorithms that optimize for both bandwidth and fault tolerance, but ignore the number of server moves required to get to the target server configuration (Section 5.4). In Section 5.5, we evaluate algorithms that also consider the number of server moves. See Section 6 for discussion on handling additional constraints within our framework.

5.1 Overview of Results

CUT+FT+BW performs the best. When ignoring the number of server moves, CUT+FT+BW achieves the best performance (see Figure 10). This algorithm achieves $30\% - 60\%$ reduction in bandwidth usage in the core of the network, while at the same time improving FT by 40% – 120%. The graph cut significantly reduces the bandwidth usage and provides a good starting point for

[7]Although this means that the optimization step is perhaps not the steepest one, it is enough to move in any descent direction in order to converge to the global optimum (see, e.g., [9]).

the steepest-descent moves that improve fault tolerance. The results of this algorithm serve as a benchmark for the other algorithms.

FT+BW is close to CUT+FT+BW. When executing FT+BW until convergence, it achieves results close to CUT+FT+BW, even without performing the graph cut (falling behind by only $10-20$ percentage points in BW reduction). This is perhaps surprising because FT+BW performs only steepest-descent moves, without the advantage of global optimization through graph cut. In scenarios where the number of concurrent server moves is limited, we can use FT+BW to perform incremental improvements (*e.g.,* move 10 servers every hour). This result shows that we can perform incremental updates without getting stuck at bad local minimum.

FT+BW achieves most improvement within few moves. FT+BW performs the "best" machines swaps right at the beginning and can thus achieve large improvement with few moves. For example, when limiting the number of moves to 20% of the servers, it reaches more than half of the potential benefits.

Random allocation in CUT+RANDLOW works well. Since many services transfer relatively little data, they can be spread randomly across the datacenter to achieve high fault tolerance without hurting bandwidth.

5.2 Methodology

We need the following information to perform the evaluation: a) network topology of a cluster, b) services running in a cluster and number of machines required for each service, c) list of fault domains in the cluster and list of machines belonging to each fault domain, and d) traffic matrix for services in the cluster. We describe each below.

Network Topology of a Cluster

We model the network as a hierarchical tree, each level of which *may* have a different oversubscription ratio, branching factor, and redundancy count. This model can represent a wide range of topologies from a Clos network/fat-tree (the Clos becomes a node with high branching factor, high redundancy count, and low oversubscription) to a hub-and spoke (high branching factor, redundancy count = 1, and high oversubscription). The topologies vary among the datacenters we study, including many hybrids such as a Clos network connecting a large number of sub-topologies, each of which is a 2-redundant aggregation tree. Given a hierarchical tree representing the network topology, *core links* are links from the root of the tree to the second-level nodes, which in our datacenters typically represent the VLANs. Accordingly, we use the network topology to determine whether machine pairs are located in different VLANs, which indicates that their traffic traverses through the core.

Services in a Cluster

We use four production clusters with thousands to tens of thousands of servers to obtain the list of services and the number of required machines in each service. Some statistics on the number of services and their sizes are presented in Section 2.

Fault Domains

Based on the network topology and physical wiring of the production clusters, we identified four types of fault domains: server containers (containing on the order of thousand servers), top-of-rack switches, server enclosures (containing three or four servers) and power domains (containing hundreds of machines).

Traffic Matrix

We obtain the service-to-service traffic matrix by collecting network traces of machine-to-machine communication that contain the amount of data sent and received. These traces are collected from a pre-production cluster that runs a full copy of Bing. This cluster contains thousands of servers, executes fraction of live traffic, and is used for testing before deploying code to the production datacenters. We do not have such traces for production clusters. Because machine moves are potentially expensive, we do not respond to changes in traffic patterns that happen on the order of minutes or even hours. Instead, we use the traces to compute the long-term, or steady-state, bandwidth requirements for each service pair.

Because we extract the data for network topology, fault domains and services from production clusters, we need to map the services from the pre-production cluster to services in production clusters. We directly match services with identical names and can match about $1/3$ of the total traffic in the pre-production cluster. We match the remaining production services (in descending order by number of servers) with remaining pre-production services (in descending order by required bandwidth). Such mapping increases bandwidth requirements for large services and thus makes the bandwidth optimization more difficult than other mappings.

Comparing Different Algorithms

All evaluated algorithms produce different server allocations for different input parameters, and therefore explore the tradeoff space between improving bandwidth and fault tolerance. The input parameters are the α (for CUT+FT+BW and FT+BW), number of server swaps to perform (for CUT+FT and FT+BW), and fraction of services to allocate randomly (for CUT+RANDLOW). Instead of comparing the performance of the algorithm on single configurations of problem parameters, we compare the entire achievable tradeoff *boundaries* for these algorithms. In other words, we run the algorithm with different values of the parameters and plot the BW and FT achieved (see Figure 10). The solid line in the figure clearly represents the best algorithm, since its performance curve "dominates" the respective curves of the other two algorithms.

We show the changes in BW and FT relative to the current server allocation in the datacenters. The current allocation algorithm considers communication patterns of only a few services (because this data is typically not available) and does not systematically consider all fault domains. In the following figures, three solid circles represent the FT and BW at starting allocation (at origin), after BW-only optimization (bottom-left corner), and after FT-only optimization (top-right corner) to provide context for results of the other algorithms. For each service, the relative improvement in FT corresponds to an increase in the number of available servers during a single worst-case failure; for example, a service with 20% improvement in FT will have 20% more servers available during such failure. In our graphs, we depict the average FT improvement across services.

5.3 Optimizing for Either BW or FT

Here we describe the results of applying algorithms that optimize either for bandwidth or for fault tolerance, but not both. BW-only algorithm applies the minimum k-way cut to the communication graph. The results for this algorithm to six different datacenters of various sizes (see Figure 2) imply that we can reduce the bandwidth requirement at the network core by 45% on average (relative to the current cluster allocation). However, since this algorithm clusters services in the same partitions (containers, VLANs, ...), it also significantly reduces fault tolerance (by 66%, on average).

FT-only optimization uses the FT+BW algorithm with α set to

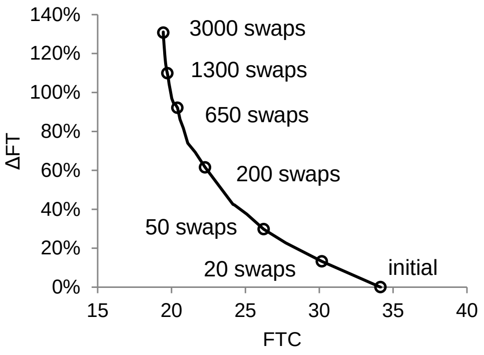


Figure 9: Convergence of FTC (the fault tolerance cost function, Equation 4) and the corresponding average worst-case survival for a smaller cluster of several thousand machines. The FT algorithm improves (reduces) FTC while indirectly improving (increasing) the fault tolerance of the services. Notice that even small changes in cost (*e.g.,* from 1,300 to 3,000 swaps) result in significant improvement in fault tolerance. Finding a swap takes from seconds to a minute, depending on the size of the cluster and the number of candidate swaps that are explored (this number is a tunable parameter in our algorithm).

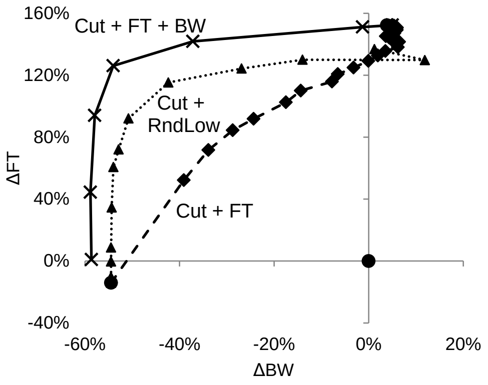


Figure 10: The achievable trade-off boundary for the CUT+FT, CUT+FT+BW and CUT+RANDLOW algorithms. For CUT+FT, each marker on the line represents the state after moving approximately additional 2% of servers in the cluster. Note that CUT+RANDLOW outperforms CUT+FT simply by randomly allocating the low-talking services; however, it does not improve the bandwidth usage of these services nor fault tolerance of the high-talkers.

zero. Applying this algorithm improves FT by 83% on average, but also increases the bandwidth requirement at the core by 7%. Although we report these changes in relation to the current server allocation at the clusters, we note that the absolute values for BW and FT metrics do not depend on the starting allocation. The graph cut approach completely ignores the starting allocation, and while the steepest descent in FT starts from a particular allocation, it converges to the globally optimal FT of the cluster.

As explained in Section 4, the FT-only algorithm uses optimization over convex FTC to indirectly improve FT. Figure 9 illustrates the negative correlation of these two metrics – as FTC decreases, the fault tolerance improves.

5.4 Optimizing for Both BW and FT

In this section we evaluate algorithms that optimize bandwidth and fault tolerance, but ignore the number of server moves required to reach the target service allocation (CUT+FT, CUT+FT+BW, and CUT+RANDLOW). All these algorithms first use the minimum k-way cut to reduce the bandwidth at the core, followed by performing gradient descent that improves fault tolerance, gradient descent that improves fault tolerance and bandwidth, and randomizing the low-talking services, respectively. We show results for one of the datacenters in Figures 10 and 11(left), and results for the remaining three datacenters in Figure 13a. We note that we have examined ways to incorporate FT optimization directly within the cut procedure. First, we tried to artificially partition each service to several subgroups, however this did not lead to satisfactory performance. We also tried to augment the cut procedure with “spreading" requirements for services. Unfortunately, this approach does not scale to large applications with hundreds of services.

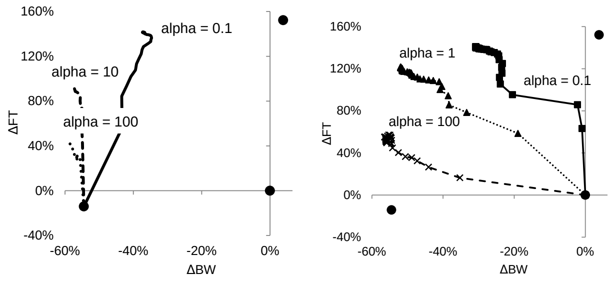


Figure 11: Change in fault tolerance and core bandwidth over time for CUT+FT+BW (left) and FT+BW (right) for three different values of α. Using larger values of α puts more weight on improving bandwidth at the expense of fault-tolerance.

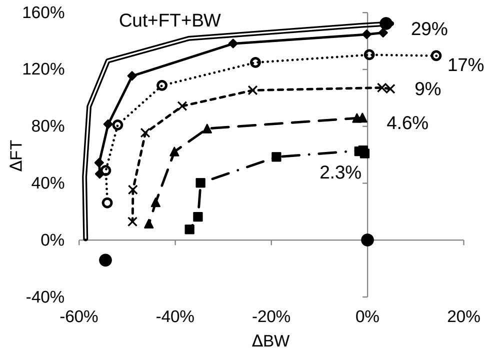


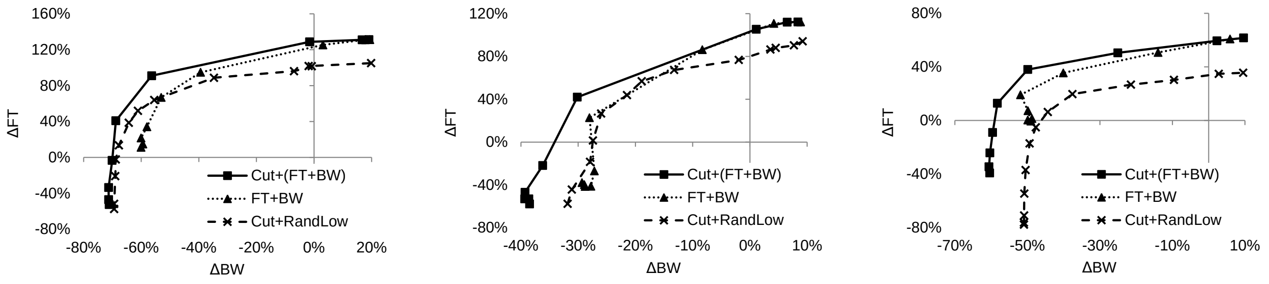
Figure 12: The achievable trade-off boundary for the FT+BW algorithm. Values next to the lines represent the fraction of the cluster that was moved. The outer-most line represents the result of the CUT+FT+BW.

CUT+FT

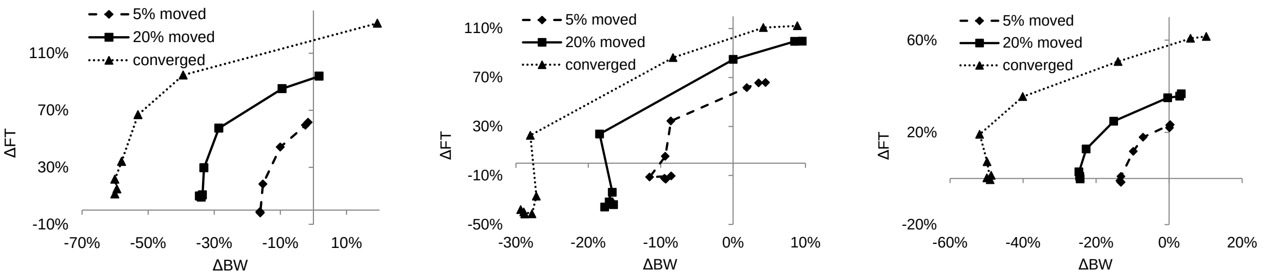
CUT+FT performs the minimum k-way cut (reaching the lower-left point in Figure 11(left)), followed by the steepest descent algorithm that only considers improvement in fault tolerance. We executed the algorithm many times, each time allowing it to swap an increasing number of servers. The resulting BW and FT metrics of the obtained server allocations are shown in Figure 10 (in particular, the CUT+FT curve). The diagonal line in this figure represents the achievable tradeoff boundary for this algorithm; by changing the total number of performed swaps, we can control the tradeoff between BW and FT. This formulation of the problem is convex, so performing steepest descent until convergence leads to the global minimum with respect to fault tolerance.

CUT+FT+BW

CUT+FT+BW is similar to CUT+FT, but gradient descent also considers improvement in bandwidth when selecting a machine swap. The results of this algorithm depend on the value of α;



(a) Tradeoff boundary for CUT+FT+BW (solid line), CUT+RANDLOW (dashed line), and FT+BW (dotted line, executed until convergence).



(b) Tradeoff boundary for FT+BW when limiting the number of moves to 5% of the total servers (dashed line), 20% of the total servers (solid line), or running FT+BW until convergence (dotted line, same as dotted line in (a) above).

Figure 13: Evaluation of CUT+FT+BW, CUT+RANDLOW, and FT+BW on three additional datacenters.

higher values of α put more weight on improvement of bandwidth at the cost of not improving fault tolerance as much. Figure 11 shows the progress of this algorithm for three different values of α. By running the algorithm until convergence with several different values of α, we obtain the "benchmark boundary" to which other algorithms can be compared (see the solid line in Figure 10). Because this algorithm is not optimizing over a convex function, it is not guaranteed to reach the global optimum. However, it still significantly improves both the bandwidth and fault tolerance of the cluster.

CUT+RANDLOW

CUT+RANDLOW first performs the minimum k-way cut, followed by randomizing the allocation of the least-communicating services responsible for total of $y\%$ of the total traffic in the cluster. The achievable tradeoff boundary for this algorithm[8] is in Figure 10. This algorithm achieves performance close to the CUT+FT+BW algorithm, but it does not optimize the bandwidth of the low-talking services nor the fault tolerance of the high-talking ones, which explains the gap between these two algorithms.

5.5 Optimizing for BW, FT, and NM

In this section we evaluate FT+BW, which also considers the number of server moves. This algorithm starts from the current server allocation and performs steepest descent moves on the cost function that considers the fault tolerance and bandwidth. The progress of this algorithm for different values of α is shown in Figure 11(right); as in CUT+FT+BW, using larger α skews the optimization towards optimizing bandwidth. In this figure, each marker corresponds to moving approximately additional 2% of servers. Notice that improvement is significant at the beginning and then slows down.

Figure 12 shows the achievable tradeoff boundaries of FT+BW for different fractions of the cluster that are required to move. For example, notice that we obtain significant improvements by moving just 5% of the cluster. Moving 29% of the cluster achieves results similar to moving most of machines using the CUT+FT+BW algorithm (see the outer double line in Figure 12). Results for three additional datacenters are presented in Figure 13(b).

Finally, notice that when running FT+BW until convergence (see Figure 13a), it achieves results close to CUT+FT+BW even without the global optimization of graph cut. This is significant, because it means we can use FT+BW incrementally (*e.g.,* move 2% of the servers every day) and still reach similar performance as CUT+FT+BW that reshuffles the whole datacenter at once.

5.6 Understanding the Improvements

Here we explain the actual allocation changes performed by the FT+BW algorithm for α values of 1.0 and 0.1 and how they lead to the improvements in bandwidth and fault tolerance. For α = 1.0, FT+BW reduced the core bandwidth usage by 47% and improved the average fault tolerance by 121%. Overall, the contribution of every single service on the core bandwidth consumption was reduced. This happened at the expense of fault tolerance of some services as they were packed closer to conserve bandwidth. The fault tolerance was reduced, stayed the same, and was improved for 7%, 35%, and 58% of services, respectively. Finally, Figure 14(left) shows the changes of bandwidth and fault tolerance for all services with *reduced* fault tolerance. Again, a few services contributed significantly to the 47% drop in bandwidth, but paid for it by being spread across fewer fault domains.

For α = 0.1, FT+BW achieved reduction of bandwidth usage by 26%, but improved the fault tolerance by 140%. In this case, fault tolerance was reduced only for 2.7% of the services (see the right plot on Figure 14) and the magnitude of the reduction was much smaller than for α = 1.0. This demonstrates how the value of α controls the tradeoff between fault tolerance and bandwidth usage.

To understand the impact of improved fault tolerance on the services, we compute the number of services that are affected by a potential hardware failure. We say that a service is affected by a potential hardware failure if its worst-case survival is less than a certain threshold H. We use $H = 30\%$ that is used in the alert sys-

[8]Using $y = 0, 25, 50, 60, 70, 80, 85, 90, 95, 98, 99, 99.9, 100$.

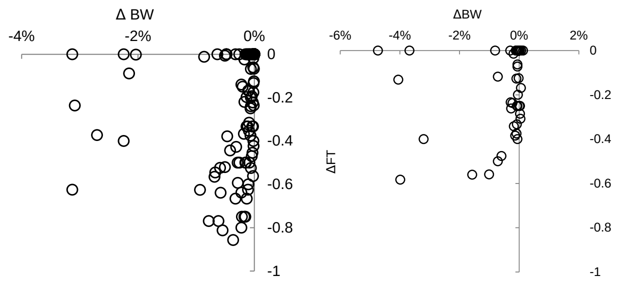


Figure 14: The relative change in core bandwidth (x-axis) and fault tolerance (y-axis) for all services (circles) that actually reduced their fault tolerance for α=1 (left) and α=0.1 (right).

tem for Bing. For FT+BW with $\alpha = 1.0$ discussed above, the fault tolerance increased by 121%. Based on simulations, the number of services potentially affected by hardware failures would drop by a factor of 14 when using the allocation proposed by FT+BW.

We believe that our approach is general, nevertheless the achieved improvements depend on communication patterns of the deployed applications, as we demonstrate below. We stress-test our algorithms using synthetically modified communication matrix to capture different extremes of communication types. In the first two experiments, we a) removed all communication between pairs of services that belong to different environments and, b) removed all communication between different services (kept only communication inside the service). The resulting matrices are close to communication patterns one might observe in a cloud computing environment with many independent parties. In both cases, we achieved results very close to using the original traffic matrix. These experiments highlight the robustness of our approach, as it can handle significantly different communication patterns.

Finally, we experimented with a traffic matrix with uniform communication pattern; in particular, for every pair of logical machines that belong to services that communicate (based on the original traffic matrix), we set their bandwidth requirement to a constant C. Thus, every pair of servers requires bandwidth of either 0 or C. Notice that the traffic in this matrix is not skewed as the original matrix. Using this matrix, we were able to achieve the same improvement in fault tolerance, since it does not depend on the traffic matrix. However, none of the algorithms were able to reduce the bandwidth usage in the core of the network; the best improvement was on the order of -10^{-7}. This demonstrates that the skewness of the communication patterns is crucial for our approach.

6. ADDITIONAL SCENARIOS

Besides the main objectives of improving fault tolerance and reducing bandwidth consumption in the core of the network, the deployment of our algorithms may need to consider further practical issues. We describe some of them here and discuss how our algorithms can be extended to support them.

Hard constraints on fault tolerance and placement. Certain important services might have hard constraints on the minimum required fault tolerance. Increasing the weight of a service monotonously improves its FT. Thus, we could exploit this monotonicity to efficiently find the weight that should be assigned to the service (through offline simulations). Further, when evaluating a machine swap, we could check the FT of the involved services and never perform a swap that would reduce FT below a specified limit. Similarly, if a logical machine has certain hardware requirements, we could also check them during swap evaluation.

Optimization of bandwidth across multiple layers. The focus in this paper is on reducing the bandwidth at the core, but the same algorithms could be applied at the lower levels of the network, such as reducing the bandwidth usage on the aggregation switch. Our optimization framework can address this extension by performing the cuts *hierarchically*.

Preparing for maintenance and online recovery. A large number of machines might become unavailable during maintenance or during long-lasting failures. Setting high weight w_j for the affected fault domain j will incentivize our algorithm to move machines away from that domain, thereby improving FT of applications.

Adapting to changes in traffic patterns. Since services might be added and removed and their traffic matrix can change over time, the FT+BW algorithm might be executed every time such a change is detected. When the change in the algorithm inputs is small, FT+BW would propose only a small number of machine swaps.

Multiple logical machines on a server. Our optimization framework could be extended to support single physical machine hosting multiple logical machines. Indeed, the basic idea of preserving a feasible allocation by swaps may carry over to the more general case where services could choose between "small", "medium" or "large" logical machines. Under such extensions, we might restrict the set of feasible swaps to avoid complicated bin packing issues. We leave this extension for future work.

In addition to the above scenarios, there are interesting extensions for future work. These include fault tolerance with correlated failures, non-uniform and time-varying communication patters, and additional performance metrics for fault tolerance and bandwidth (such as worst-case core link utilization).

7. RELATED WORK

Datacenter traffic analysis. Datacenter traffic has been studied in the past. In [8], the authors analyze the communication patterns in datacenters at the flow- and packet-level, while in [20] they concentrate on a particular, MapReduce-like application. In this paper, we study a complex Web application consisting of a large number of interconnected services and provide a detailed analysis of their communication patterns at the application level.

Datacenter resource allocation. Current approaches to resource allocation in multi-tenant datacenters can be divided into two major categories. On the one hand, Seawall [33] and NetShare [24] share the network bandwidth among coexisting tenants based on their weights. Node allocation is a non-goal for these algorithms. On the other hand, Oktopus [6] and SecondNet [17] provide guaranteed bandwidth to tenant requests. Oktopus uses greedy heuristics to allocate nodes and bandwidth for two specific types of topologies. SecondNet provides pairwise bandwidth guarantees between tenant nodes. Solutions in both categories do not consider application availability; since they are tailored for satisfying bandwidth constraints, it is nontrivial to extend them to include availability.

Virtual network embedding. Allocation of arbitrary virtual network topologies on top of physical networks is a well-studied problem [10]. Existing solutions include greedy heuristics [38], simulated annealing [27], multi-commodity flow models [34, 37], and mixed-integer formulations [11]. However, these algorithms do not consider availability constraints.

Survivable embedding. Survivability from node and link failures has been extensively investigated for optical and MPLS networks [23, 25, 35]. The key objective, however, is to ensure IP connectivity in presence of failures. Survivable virtual network embedding [31] deals with a similar problem in the context of network virtualization. These solutions do not consider node allocation, nor can they work at datacenter scale.

High availability in distributed systems. Resource allocation to achieve high availability in distributed systems has been studied before [3,5,7,36]. These projects divide the servers into fault domains with different failure probabilities, and allocate data and application components to maximize some measure of application availability. However, they do not consider bandwidth usage.

VPN and network testbed allocation. Algorithms for allocating virtual private networks (VPNs) in a shared provider topology [13, 18, 30] involves finding paths for a collection of source/destination pairs given a small, pre-defined set of customer nodes. The authors of [32] consider bandwidth constraints when allocating nodes in Emulab. Neither of these considers application availability.

8. CONCLUSION

Fault tolerance and reduction of bandwidth usage are often contradictory objectives – one requires spreading machines across the datacenter, the other placing them together. Indeed, simulations on large-scale Web application demonstrate that optimizing for one of these metrics independently improves it significantly, but it actually degrades the other metric. In this paper, we propose an optimization framework that provides a principled way to explore the tradeoff between improving fault tolerance and reducing bandwidth usage. The essentials of this framework are motivated by a detailed analysis of the application's communication patterns. In particular, our analysis shows that the communication volume between pairs of services has long tail, with the majority of traffic being generated by small fraction of service pairs. This allows our optimization algorithms to spread most of the services across fault domains without significantly increasing the bandwidth usage in the core.

Acknowledgements

We thank Peter Winkler, Seffi Naor, Mohit Singh, Sharon Goldberg and Srikanth Kandula for useful discussions. We are grateful to Michael Ma, Greg Foxman, Alexander Kolomeets and other members of the Bing's Autopilot team for providing important input during the project.

9. REFERENCES

[1] The avoidable cost of downtime. ca Technologies, `http://goo.gl/SC6Ve`, 2010.

[2] Calculating the cost of data center outages. Ponemon Institute, `http://goo.gl/ijLoV`, 2011.

[3] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, pages 17–32, 2010.

[4] Amazon.com. Summary of the Amazon EC2, Amazon EBS, and Amazon RDS service event in the EU west region. `http://aws.amazon.com/message/2329B7/`, 2011.

[5] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *ATC*, 2000.

[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, 2011.

[7] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. S. Turaga, and C. Venkatramani. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM*, pages 1319–1327, 2008.

[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, pages 267–280, 2010.

[9] D. Bertsekas. *Nonlinear programming*. Athena Scientific, 1999.

[10] M. Chowdhury and R. Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[11] M. Chowdhury, M. R. Rahman, and R. Boutaba. ViNEYard: Virtual network embedding algorithms with coordinated node and link mapping. *ACM/IEEE ToN*, 2011.

[12] J. Dean. Designs, lessons and advice from building large distributed systems. LADIS '09 keynote talk, 2009.

[13] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *ACM SIGCOMM*, 1999.

[14] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[15] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, pages 350–361, 2011.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM*, pages 51–62, 2009.

[17] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *ACM CoNext*, 2010.

[18] A. Gupta, J. M. Kleinberg, A. Kumar, R. Rastogi, and B. Yener. Provisioning a virtual private network: A network design problem for multicommodity flow. In *ACM STOC*, pages 389–398, 2001.

[19] J. Håstad. Clique is hard to approximate withinn 1- ε. *Acta Mathematica*, 182(1):105–142, 1999.

[20] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, 2009.

[21] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[22] R. Krauthgamer, J. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *SODA*, pages 942–949, 2009.

[23] M. Kurant and P. Thiran. Survivable routing of mesh topologies in IP-over-WDM networks by recursive graph contraction. *IEEE JSAC*, 25(5):922–933, 2007.

[24] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. NetShare: Virtualizing data center networks across services. Technical Report CS2010-0957, UC San Diego, 2010.

[25] W. Lau and S. Jha. Failure-oriented path restoration algorithm for survivable networks. *IEEE TNSM*, 1(1):11–20, 2008.

[26] Y. Liu and K. S. Trivedi. A general framework for network survivability quantification. In *MMB*, pages 369–378, 2004.

[27] J. Lu and J. Turner. Efficient mapping of virtual networks onto a shared substrate. Technical Report WUCSE-2006-35, Washington University, 2006.

[28] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[29] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM*, pages 39–50, 2009.

[30] S. Raghunath, K. K. Ramakrishnan, S. Kalyanaraman, and C. Chase. Measurement based characterization and provisioning of IP VPNs. In *ACM IMC*, pages 342–355, 2004.

[31] M. R. Rahman, I. Aib, and R. Boutaba. Survivable virtual network embedding. In *IFIP Networking*, pages 40–52, 2010.

[32] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM CCR*, 33(2):65–81, April 2003.

[33] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Sharing the data center network. In *USENIX NSDI*, 2011.

[34] W. Szeto, Y. Iraqi, and R. Boutaba. A multi-commodity flow based approach to virtual network resource allocation. In *IEEE GLOBECOM*, pages 3004–3008, 2003.

[35] K. Thulasiraman, M. S. Javed, and G. L. Xue. Circuits/cutsets duality and a unified algorithmic framework for survivable logical topology design in IP-over-WDM optical networks. In *IEEE INFOCOM*, 2009.

[36] H. Yu, P. B. Gibbons, and S. Nath. Availability of multi-object operations. In *NSDI*, 2006.

[37] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM CCR*, 38(2):17–29, April 2008.

[38] Y. Zhu and M. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *IEEE INFOCOM*, 2006.