Conference on File and Storage Technologies (FAST'02), pp. 203–217,
28–30 January 2002, Monterey, CA. (USENIX, Berkeley, CA.)

Appia: automatic storage area network fabric design

Julie Ward, Michael O'Sullivan[†], Troy Shahoumian, and John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA and [†]*Stanford University, Stanford, CA*
jward@hp.com, mosu@stanford.edu, troy_shahoumian@hp.com, john_wilkes@hp.com

Abstract

Designing a storage area network (SAN) fabric requires devising a set of hubs, switches and links to connect hosts to their storage devices. The network must be capable of simultaneously meeting specified data flow requirements between multiple host-device pairs, and it must do so cost-effectively, since large-scale SAN fabrics can cost millions of dollars. Given that the number of data flows can easily number in the hundreds, simple overprovisioned manual designs are often not attractive: they can cost significantly more than they need to, may not meet the performance needs, may expend valuable resources in the wrong places, and are subject to the usual sources of human error.

Producing SAN fabric designs automatically can address these difficulties, but it is a non-trivial problem: it extends the NP-hard minimum-cost fixed-charge multicommodity network flow problem to include degree constraints, node capacities, node costs, unsplittable flows, and other requirements. Nonetheless, we present here two efficient algorithms for automatic SAN design. We show that these produce cost-effective SAN designs in very reasonable running times, and explore how the two algorithms behave over a range of design problems.

1 Introduction

A SAN (storage area network) connects a group of servers (or hosts) to their shared storage devices (such as disks, disk arrays and tape drives) through an interconnection fabric consisting of hubs, switches and links. We present results for designs using today's dominant SAN fabric for the SCSI block-level protocol, FibreChannel [13]. The storage industry is in the process of adding switched Ethernet as an alternative block-level network transport. We believe that our work applies equally to both, and could also usefully be applied to file-based storage systems, and even general-purpose local-area networks (LANs).

An example FibreChannel SAN is shown in Figure 1.

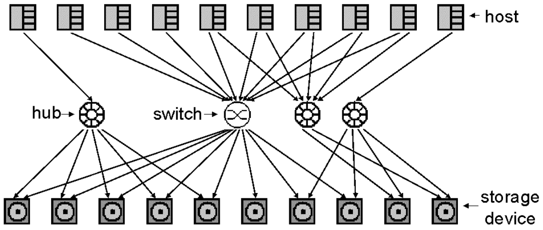

Figure 1: A simple, single-layer SAN fabric. Hosts appear in the top row, devices in the bottom row, and switches and hubs in between.

SANs offer many advantages over direct-connected local storage, including superior connectivity of servers to storage devices, better utilization of storage resources, centralized administration and management, increased scalability, and improved performance. In spite of these advantages, the adoption of SANs has been relatively slow. Some of this is due to interoperability difficulties between vendors, but as these are being resolved, the next barrier appears to be the complexities associated with designing the SANs, because this involves all of the problems of network design – in an environment with essentially no automatic flow control, and zero tolerance for packet loss, due to the low-level nature of the SCSI protocol.

As a result, designing even small SANs requires considerable time and effort from IT experts. Their manual methods often result in expensive, overprovisioned designs – and this becomes more of a problem as the designs get larger and more complex. This matters: it is not difficult to spend 10–20% of the total storage system cost on the SAN fabric elements, and SAN designs of a scale that require an investment of millions of dollars in the SAN fabric alone are becoming more common. We have witnessed a factor of three difference in the cost of a SAN between a manual design ($4m) that took several days and an automatically-generated one ($1.4m) that took a few minutes.

Design mistakes can be subtle and therefore easy to overlook, yet potentially very costly; poor performance is commonplace, and downtime in failure situations can result if the fault-tolerance aspects are mis-designed. As SANs grow to include hundreds or even thousands of storage devices, it becomes increasingly difficult, even for SAN experts, to manually design cost-effective and reliable SANs.

We believe that the most effective approach to these problems is to automate the design of SANs. Such designs must take into account the performance demands (to avoid queuing or packet loss), and they should try to minimize system cost, because SAN components are quite expensive. The result would enable the wider deployment of SANs, as well as increase the likelihood that the systems deployed would meet real needs.

This paper presents just such a solution: a tool to design SANs automatically. We call it Appia, after the Appian Way, one of the network of roads leading to ancient Rome.

1.1 Automated design of storage systems

Appia was developed to operate in concert with a set of tools that select and design the storage-device portions of a complete storage system [2, 4]. These tools use workload and device performance information to select and configure storage devices, and then determine appropriate data placements on those devices. Their goal is to design a system that meets performance goals with high reliability at low cost. A side effect is that the tools' output includes information about the workload data-flows from each host to each storage device: precisely the information that is needed to design the SAN fabric to connect the hosts to their storage.

Such tools significantly reduce the human intervention required to design storage systems: people can express their needs at a relatively high level, and the tools can design a storage system to meet their needs, taking into account all the low-level details, such as predicting the complex performance effects that result from mixing workloads on shared storage devices. Better yet, such tools can be used in an automatic control loop, allowing the storage system design to evolve completely automatically when dealing with load and system changes, without the need for human intervention.

We wanted to achieve the same benefits for SAN design. The results presented here are the first outcome of that goal. In particular, we present two algorithms for cost-effective SAN fabric design. These two approaches, which we call *FlowMerge* and *QuickBuilder*, demonstrate complementary strengths. *FlowMerge*, which is more computationally intensive, tends to find lower-cost designs for SANs with sparse connectivity requirements, whereas *QuickBuilder* excels when connectivity requirements are dense. We found that the better of two designs is, on average, within 33% of the optimal design cost for empirical test problems that are small enough to solve optimally. Moreover, these designs are found in a few minutes or less for SANs with 50 hosts and 100 devices, a size typical of the largest current installations. Because of their complementary strengths, both algorithms are included in Appia.

1.2 Structure of the paper

The remainder of this paper is organized as follows. Section 2 presents a statement of the SAN fabric design problem, including notation and related work. Section 3 presents an overview of the *FlowMerge* algorithm for finding cost-effective SAN fabric designs. The *QuickBuilder* algorithm is presented in §4. In §5 we present computational results comparing the effectiveness of the two algorithms. Furthermore, for small problems we compare the cost of designs produced by *FlowMerge* and *QuickBuilder* with the cost of optimal designs. Future work and conclusions are presented in §6 and §7.

2 The SAN design problem

The SAN design problem can be stated quite simply: we are given a set of hosts, a set of storage devices, and a set of requirements in the form of data flows between host-device pairs. Each flow has a desired bandwidth. The goal is to build a minimum-cost SAN to support all of these requirements simultaneously. To do so, one must select a set of fabric nodes (switches and hubs), a set of links connecting pairs of nodes (hosts, devices and fabric nodes), a topology with which to join these together, and a single path through the network for each flow. (The single-path restriction arises from SCSI request-ordering constraints.)

The resulting fabric design must be *feasible* - that is, it must satisfy constraints that ensure it is buildable, and it must support the connection and performance requirements. These constraints are: (1) the number of links connected to a host, device or fabric node must not exceed the number of ports available there (these restrictions are called *degree constraints*) and (2) the flow routing must honor the bandwidth limitations of links and fabric nodes. Because packets travel differently through hubs and switches, their bandwidth constraints differ. Packets routed into a switch are forwarded directly to the next destination in their path. In contrast, packets routed into a hub are transmitted through all connected hubs and all links attached to these hubs; they are seized by their next destination. Thus, the total flow into an interconnected set of hubs is limited by the minimum of the bandwidth of each individual hub, the bandwidth of each connected link, and the bandwidth of each port used by these links. The bandwidth of switches is therefore more efficiently utilized than hub bandwidth.

Data about the flows is readily available from solutions to the storage-system and data-placement design problems [2, 4], but it may also be obtained from the tried and true techniques of measurement of an existing system or estimation. Obviously, no design tool is better than the inputs it is given – but the comparison point here is manual design, not complete knowledge of the system's future behavior. It is easy enough to build in a certain amount of "slack", to allow for errors, or anticipated future growth. Indeed, we believe that it is better to have the slack specified up front as part of the goal, so that the design system can take it into account, rather than trying to build in slack "after the fact" by adding excess SAN elements in places where they may not do the most good.

The design algorithms we describe run fast enough that they can be used in interactive "what if" scenario exploration, in conjunction with manual input from a SAN design expert. The low-cost designs the tools produce may not always "look pretty"; some people prefer greater symmetry in their solutions, even at the expense of greater cost. As such, we believe it is important to use this kind of tool – at least at first – in a context where there is a chance for experts to modify the output it produces. Nonetheless, it is our aim to develop tools that can be placed into a completely automatic design-deploy-monitor-redesign loop.

2.1 Related work

SAN design is currently done manually by IT experts, who use error-prone ad-hoc methods or canned topologies that often result in grossly overprovisioned designs. While overprovisioning can be advantageous, it is important that it is done strategically to provide high performance, scalability, reliability, and robustness to changes in requirements. Some canned designs currently in use, such as the Brocade Core-Edge architecture [9], possess these characteristics. They are used when the SAN designers have no systematic way to predict the connectivity and data flow requirements in their SANs, and so opt for full connectivity between hosts and devices. But this flexibility comes at a very high price: many fabric elements are needed to provide this connectivity, especially at high bandwidths. In general, when any information is available about SAN requirements, far more cost-effective designs can be found.

As part of our search for algorithms to apply to this problem, we turned to the literature on network design. Unfortunately, most traditional network design approaches only address link costs, because switches are cheaper than trenching in wide area telephone networks, which are the target of most of this work. In the SAN case, the reverse is usually the case: in mid-2001, a fully loaded 64-port FibreChannel "storage director" (a high-end fabric switch) costs close to half a million dollars, while individual fibre links for use within a data center are priced around $100–$500. As a result, much of the existing research in network design proved less applicable than we had hoped.

In particular, the SAN fabric design problem generalizes and extends several NP-hard problems in network design. For example, it generalizes the nonbifurcated network loading problem [21, 6, 3, 16, 17]. In this problem, there are several commodities, each with an origin and destination node in the network, and a required amount of the commodity that must travel through the network between these nodes. One must choose a minimum cost set of capacitated links connecting a known set of nodes to satisfy these flow requirements simultaneously. The term "nonbifurcated" refers to the requirement that a single route for each commodity must be selected; i.e., flows cannot be split across multiple paths. Each link has an associated fixed cost, and multiple links between a given pair of nodes may be selected. This problem contains the Steiner tree problem, known to be NP-complete, in which one must find the minimum cost set of links to connect a given subset of the nodes in a network. (See [23] for a survey of work on the Steiner tree problem.) The nonbifurcated network loading problem is NP-hard even when all commodities share a single source [21].

If we relax the constraint that flows cannot be split, the SAN design problem generalizes the multicommodity network design problem [20, 8, 7, 19, 22, 10, 5]. This problem is known to be NP-hard even in the single commodity case [15]. Like the nonbifurcated network loading problem, it involves choosing a set of capacitated, fixed-cost links to connect a set of nodes to satisfy multicommodity flow requirements. Any number of links between a pair of nodes can be selected. In this case, however, flows can be split. Even so, multicommodity network design problems are notoriously difficult to solve in practice. This is true because their integer programming formulations' LP relaxations do not provide tight lower bounds. Even finding feasible solutions is often difficult. Surveys of work in this area are given in [18, 1, 24].

In the NP-hard problems mentioned above, one must find a minimum cost set of links to route the flows, when the nodes in the network are known. The SAN fabric design problem generalizes these problems, in that the nodes in the network are not known *a priori*. One must choose a set of hubs and switches with which to build the interconnection fabric between hosts and devices. Several different types of hubs and switches may be available, differing in attributes such as cost, bandwidth, and number of available ports; an arbitrary number of instances of each type may be used in the SAN. It is possible, however, to construct a candidate fabric node set containing the optimal set. Few authors have considered network design problems in which the topology is unknown. The Steiner tree problem is a special case of the capacitated network design problem in which some nodes may optionally be excluded from the network. The integer programming formulation of network design problems grows in dimension exponentially with the size of the set of nodes considered, and thus it is essential to find a small candidate node set. Unfortunately in the SAN design context it may be difficult to determine such a candidate set of reasonable size due to the number of different node types considered.

SAN design also generalizes other network design problems by associating capacity and cost with nodes. [17] includes node costs, and [27, 14] consider node capacities. A node's cost and capacity can be handled within the context of standard network design problems at the expense of an additional node and arc: each capacitated or cost-inducing node can be replaced by two uncapacitated and costless nodes with an arc between them possessing the original node's cost and capacity attributes. (This assumes unidirectional links, which will not always be the case in future SAN design problems.)

Another confounding feature of the SAN design problem is the presence of degree constraints on nodes. Degree constraints appear only in special cases of the network design problem such as the degree-constrained minimum spanning tree problem [12, 11, 25], known to be NP-hard [15].

The many features of the SAN design problem have been addressed individually or in small subsets in the work mentioned above. The first to address all of its features in a common framework was [26], in which an algorithm called *Merge* was presented. *Merge* found cost-effective designs for small problems but failed to find feasible designs for larger problems. The algorithms presented here are proven to find feasible designs under a reasonable set of conditions, and their designs are generally more cost-effective.

2.2 Notation

Some notation will be useful in describing our approaches. Let $\mathcal{H}$ and $\mathcal{D}$ represent the sets of hosts and devices, respectively. Denote the set of flows by $\mathcal{F}$. Let $\mathcal{N}$ be the set of all types of switches and hubs available. Each component $i \in \mathcal{H} \cup \mathcal{D} \cup \mathcal{N}$ has a maximum number of ports p_i, each with cost π_i. Although a SAN could be built from several different types of links differing in bandwidth and cost, we restrict attention in this paper to the case when there is one available link type whose bandwidth is β and cost is γ. To simplify exposition, we also assume that all ports have bandwidth β, though ports may differ in cost. Finally, fabric node type $n \in \mathcal{N}$ has cost c_n and maximum aggregate bandwidth b_n. The SAN fabric design problem defined by given sets of hosts, devices, flows and nodes is denoted by P.

3 The *FlowMerge* algorithm

The first of our algorithms is called *FlowMerge*, which earns its name from the way it pulls together separate flows into sets of flows that share fabric nodes. It was inspired by this simple fact: when two flows with a common host or device are routed together through a link, they conserve a port on that host or device. *FlowMerge* attempts to use fabric nodes in a way that alleviates a shortage of host and device ports, by selecting subsets of flows with common hosts or devices to route together through links.

FlowMerge is a recursive algorithm that creates a SAN design by introducing, at each recursive application, a set of fabric nodes and links, with no links between fabric nodes in the set. When the algorithm terminates, the fabric design consists of one or more "layers" of nodes, where there are links between but not within layers. An example of a layered fabric produced by *FlowMerge* is shown in Figure 2. The top and bottom rows of components contain hosts and devices, respectively, and the remaining components are fabric nodes.

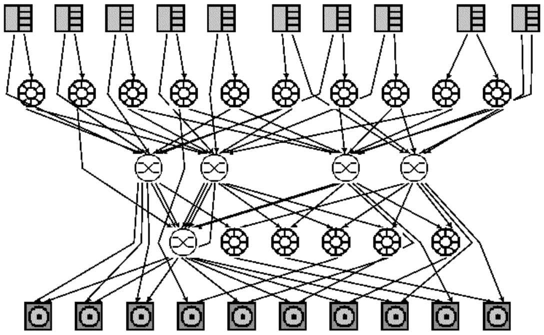

Figure 2: A sample SAN fabric produced by *FlowMerge*

The basic building block of a *FlowMerge* fabric is a *single-layer fabric*. This is a fabric that has no links between fabric nodes, so that each flow requirement is routed either along a direct link between its host and device, or along a two-link path that passes through a single fabric node. Figure 1 depicts an example of a single-layer fabric. In §3.1 we describe the procedure that introduces a single layer of nodes, which we call *Single-Layer FlowMerge*. In §3.2 we outline the recursive procedure that creates a multi-layered fabric through successive calls to *Single-Layer FlowMerge*.

3.1 *Single-Layer FlowMerge*

The input to *Single-Layer FlowMerge* is a set $\mathcal{H}$ of hosts, a set $\mathcal{D}$ of devices, and flow requirements $\mathcal{F}$ between them. *Single-Layer FlowMerge* produces a series of single-layer fabric designs to support the flow requirements. Each design in the series is feasible with respect to all except, possibly, the degree constraints on hosts and devices. The initial design consists of a direct host-device link for each flow. This design is typically infeasible because one or more hosts or devices has fewer ports than incident links. The difference between the numbers of incident links and available ports on a given host or device is called its *port violation*. Each subsequent design in the series has a smaller total port violation than the previous design, or a lower cost than the previous design if both designs are feasible.

To see how this series of designs is obtained, consider an arbitrary single-layer fabric. Associated with each fabric node in the design is a subset of flow requirements routed via that node. Similarly, associated with each direct host-device link in the fabric is a subset of flows routed along that link. In general, the flow requirements are partioned into disjoint subsets, such that each flow requirement is in exactly one subset. Each subset in the partition has an associated fabric node or direct host-device link through which all flows in the subset are routed. We call these subsets *flowsets*.

Single-Layer FlowMerge begins with the finest partition of the flow requirements: each flow is in its own flowset. At each iteration, a new, coarser, partition is obtained by merging two flowsets together. When merging two flowsets, we must select a fabric node type among available types with which to route the flows in the merged flowset, and the links connecting hosts and devices to the node along which we route the flows. The node type is selected based on the number of ports available on the node and the cost of using the node (including the cost of required ports and links). We select the flowsets to merge to alleviate port violations, favoring reductions on the hosts and devices with the most severe violations. Cost is a tie-breaker criterion. Once two flowsets are merged, they are never split. *Single-Layer FlowMerge* continues merging flowsets until either no two flowsets can be merged, or all port violations have been eliminated and no merger produces a cost savings. *Single-Layer FlowMerge* terminates, because after a finite number of mergers (one less than the number of flows) only a single flowset remains, so no further mergers are possible. Figure 3 demonstrates how *Single-Layer FlowMerge* works on a small example.

Pseudocode for the *Single-Layer FlowMerge* algorithm is shown in Figure 4. We use the following notation:

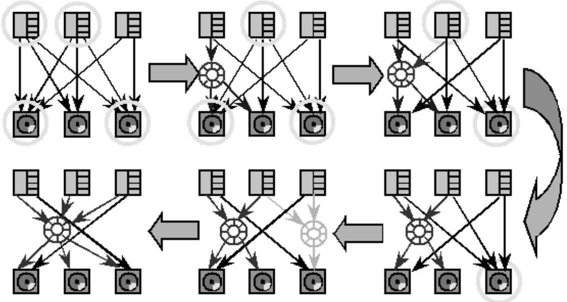

Figure 3: Example application of *Single-Layer FlowMerge*. The problem has 3 hosts and 3 devices, each with 2 ports, and a single type of switch available with 8 ports. The eight flows in the problem each have bandwidth 33 MB/s. Links and ports have bandwidth 100 MB/s. Six successive designs are shown, beginning with the one that assigns each flow to its own link. In each design, hosts and devices with the highest port violation are circled. For example, in the first design, the highest port violation is one: there are two hosts and two devices each with three incident links and only two ports. Each design in the series reduces the port violation on one host or device from the previous design by merging two flowsets together. After four mergers, all port violations are eliminated. The last merge eliminates one fabric node and thereby reduces the cost of the fabric.

- **F** is the partition of the set of flows $\mathcal{F}$ into flowsets (more explicitly, **F** is a collection $\{J : J \subset \mathcal{F}\}$ with the property that $\cup_{J\in\mathbf{F}} J = \mathcal{F}$ and $K \cap J = \emptyset$ for all $J, K \in \mathbf{F}$, $J \neq K$);

- $\mathcal{N}$ is the set of available fabric node types;

- l is the single available link type;

- $\mathcal{M} \subset \{(J,K,n) : J,K \in \mathbf{F}, J \neq K, n \in \mathcal{N} \cup \{l\}\}$ is a set of triples consisting of two flowsets and one node type or link;

- $score_m$ is a function defined on elements of $\mathcal{M}$.

We refer to elements of $\mathcal{M}$ as *mergers* because they represent the combinations of flowset pairs and node types that are candidates for merging.

In the *Single-Layer FlowMerge* psuedocode, each application of the outer loop results in a merger. We start an application of this loop by initializing the set of candidate mergers $\mathcal{M}$ to be all possible flowset pair-node combinations, and then eliminating infeasible combinations. Next, we compute the port violations on hosts and devices. If there are candidate mergers left to consider, we refine this set in the inner “Else while” loop. This loop considers port violation degree ranging from the current worst, v, down to 1. For each such degree, it “scores” each merger in $\mathcal{M}$ by counting the number of hosts and devices with that degree port violation on which it conserves ports. After scores are computed, mergers that do not achieve the highest score for this degree are removed from consideration. If multiple candidate mergers still remain, it eliminates all but those with the lowest cost. After the inner loop is finished, a single merger from the candidate set is then implemented. Since we are indifferent between all candidate mergers at this stage, we could introduce randomization into the algorithm in the selection of the merger from the final set of candidates.

Scores computed in the inner loop can be largely reused in successive applications of the outer loop. In our implementation, they are updated for flowset pairs containing hosts or devices whose port violation was reduced in the prior merger.

3.2 *Multi-Layer FlowMerge*

When *Single-Layer FlowMerge* is applied to a SAN fabric design problem, it will reduce at least one host's or device's port violation by at least one. (We omit the details of this proof in the interest of brevity.) However, *Single-Layer FlowMerge* may not successfully eliminate all port violations on hosts and devices. In this case, it is reapplied recursively to generate cascading layers of fabric nodes. Pseudocode for this recursive application, which we call *Multi-Layer FlowMerge*, is shown in Figure 5.

The central idea behind the recursion is as follows. We first apply *Single-Layer FlowMerge* to a SAN fabric design problem P. If all host and device port violations are eliminated from P, we have found a feasible SAN fabric design. At this point, we can stop, since *Single-Layer FlowMerge* found no cost-saving mergers and introducing new fabric nodes would only increase costs.

If instead there are remaining host and/or device port violations, the current set of fabric nodes is insufficient. We address the host port violations first, independently of the device port violations, by recasting the problem as a new SAN fabric design problem P_H that has only host port violations and no device port violations. The hosts of P become hosts of P_H. Subsets of flows in problem P are aggregated together to become flows for problem P_H according to their assignment to links in the one-layer solution to P. More specifically, for each flowset and each link into the flowset's fabric node, a new flow is created in P_H whose bandwidth is the aggregate bandwidth of flows assigned to that link. The new flow's device in P_H is the fabric node itself. If instead its flowset has no fabric node (and thus has a single direct link between a host and device), all flows routed along that link are aggregated into a single flow in P_H. For this flow we create a "dummy" device in P_H with a single port that costs the

```
Single-Layer FlowMerge
Input:  a SAN fabric design problem $P$.
Output:  a set of flowsets $\mathbf{F}$ and a fabric
node for each flowset.
   Let $\mathbf{F} = \{\{f\} : f \in \mathcal{F}\}$.
   While (true){
      Let $\mathcal{M} = \{(J,K,n) : J,K \in \mathbf{F}, J \neq K, n \in \mathcal{N} \cup \{l\}\}$.
      Remove from $\mathcal{M}$ all elements that
      represent infeasible mergers.
      Compute the port violation on each
      source and terminal with respect to
      the current set of flowsets and their
      associated nodes and link.  Let $v$ be
      the highest port violation among them.
      If $\mathcal{M} = \emptyset$, break.
      Else while ($v > 0$) and ($|\mathcal{M}| > 1$) {
         For each $m \in \mathcal{M}$ {
            Let $score_m = 0$.
            For each source and terminal $c$
            with port violation $v$
               If merger $m$ reduces the port
               violation on $c$
                  Let $score_m = score_m + 1$.
            Remove elements of $\mathcal{M}$ which did
            not achieve the highest score.
            Let $v = v - 1$.
         }
         For each $m \in \mathcal{M}$
            Compute the cost of merger $m$.
         Remove mergers in $\mathcal{M}$ which did not
         achieve the lowest cost.
      }
      Return a random $\hat{m} = (\hat{J}, \hat{K}, \hat{n}) \in \mathcal{M}$.
      If the merger $\hat{m}$ reduces the port
      violation on at least one source
      or terminal with a positive port
      violation, or if the merger has a
      negative cost, perform the merger:
      delete $\hat{J}$ and $\hat{K}$ from $\mathbf{F}$, discarding
      their respective nodes, and replace
      with a new flowset $\hat{J} \cup \hat{K}$ with a node of
      type $n$.
      Otherwise, break.
   }
   Return flowsets in $\mathbf{F}$ and their associated
   fabric nodes.
```

Figure 4: *Single-Layer FlowMerge*

```
Multi-Layer FlowMerge($P, L$)
Input:  a SAN fabric design problem $P$ and a
layer number $L$.
Output:  a feasible SAN fabric design
consisting of one or more layers of fabric
nodes, with no links between nodes in a given
layer.
   Apply Single-Layer FlowMerge to $P$.
   If there are remaining host port
   violations in current solution to $P$ {
      Recast problem as new Multi-Layer
      FlowMerge problem $P_H$.
      Apply Multi-Layer FlowMerge($P_H, L-1$).
      Add fabric for $P_H$ to fabric for $P$.
   }
   If there are remaining device port
   violations in current solution to $P$ {
      Recast problem as new Multi-Layer
      FlowMerge problem $P_D$.
      Apply Multi-Layer FlowMerge($P_D, L+1$).
      Add fabric $P_D$ to fabric for $P$.
   }
   If there are no remaining port violations
   in $P$
      Return fabric for $P$.
```

Figure 5: *Multi-Layer FlowMerge*

same as its original device's ports. Thus, the set of devices in P_H consists of fabric nodes from P and dummy devices corresponding to devices from P; none of these have port violations.

We then apply *Multi-Layer FlowMerge* to the P_H and create a multi-layered fabric for that problem. The next step is to incorporate the fabric for P_H into the solution we are building up for P. P_H's fabric layers are inserted into the fabric of P.

Similarly, if device port violations remain in P after the application of *Single-Layer FlowMerge*, then a new problem P_D is created in a way that mirrors the creation of P_H. It has all devices from P as its devices, aggregated flows from P as its flows, and hosts consisting of fabric nodes and dummy hosts corresponding to hosts in P. P_D is solved and its fabric is incorporated into P's solution.

In this brief overview of *Multi-Layer FlowMerge*, we have omitted many details. For example, there are special precautions taken which ensure that there are no links between hubs in the fabric. While this is not strictly necessary, it is the most efficient way to ensure that hub capacity constraints are honored.

3.3 Correctness and Effectiveness

Although the proof will be omitted here, *FlowMerge* finds a feasible SAN fabric design when the following two conditions hold:

> For each host and device, there exists an assignment of its flows to its ports such that the total bandwidth of flows assigned to a port is at most the port's bandwidth (β). (1)

> There is a switch type available having at least three ports and bandwidth at least β. (2)

Assumption (1) is clearly a necessary condition for the existence of a feasible fabric design. Assumption (2) is not necessary, in general, since a small SAN may require no fabric nodes at all. However, it is not at all restrictive; all real switches possess at least 8 ports and typically many more, and have bandwidth many times that of a link. The two assumptions together are sufficient to ensure that *FlowMerge* finds a feasible fabric design.

While we have no analytical optimality bounds on *FlowMerge* designs, we do have empirical results comparing its designs to those produced by *QuickBuilder* and, for small problems, optimal designs.

Our results indicate that *FlowMerge* is very effective at building one-layer fabrics, which are typically sufficient for problems that either have few hosts and devices and have sparse connectivity requirements between hosts and devices. But for SANs that are so large or whose connectivity requirements are so dense that they require multiple fabric layers, it is less effective than *QuickBuilder*. There are several explanations for these results.

First, the class of fabrics *FlowMerge* generates is more restrictive than those built by *QuickBuilder*. *FlowMerge*'s layered fabric structure, where each layer is built myopically, may exclude more cost-effective fabric designs. In each layer it tries to resolve as many port violations as possible before introducing the next layer. It never considers changing a fabric layer that was created in an earlier application of *Single-Layer FlowMerge*.

Second, because *FlowMerge* only considers pairwise mergers, it can get stuck in locally optimal solutions. To see why, suppose it has found a feasible partition for a layer and is seeking only cost-improving mergers. It will quit if no merger is profitable. In many examples, we have seen that a better solution could have been obtained if mergers of more than two flowsets were considered; this occurs frequently in the multilayered solutions.

Allowing backtracking, or permitting non-cost-improving mergers with some small probability (in the spirit of simulated annealing) are techniques that are likely to improve *FlowMerge*'s performance, particularly on problems requiring multiple layers of fabric. Results from our current implementation of *FlowMerge* will be presented in more detail in §5.

4 The *QuickBuilder* Algorithm

In this section we outline a second, two-phased approach to SAN fabric design, called *QuickBuilder*. It is based on the observation that since flows cannot be split across multiple paths in the network, each flow must be assigned to a single port on its host and device.This matters because the way in which flows are assigned to ports has a large impact on the remainder of the SAN design. A clever assignment creates a partition of the host and device ports into disjoint subsets of ports called *port groups*. The port group of port p is a set of ports that includes p; if q is a port in the port group and a flow assigned to q is also assigned to port r, then r is in the port group. In short, the port group of port p includes p, all ports p must communicate with, all ports they communicate with, etc. In the language of graph theory, port groups are the connected components of a graph in which the nodes are ports, and links connect port pairs with common flows assigned to them. The critical insight was that each port group can be treated as an independent, smaller design problem. In general, the fewer ports in a port group, the less fabric is required to support its flows. Thus, the finer the decomposition, the less costly the fabric. *QuickBuilder* seeks an assignment that results in a fine decomposition.

QuickBuilder first assigns each flow requirement to a single port on its host and a single port on its device (the *port assignment* phase); the flow will later be routed through these ports in the second phase. The assignment obtained in the first phase implies a partition into port groups. Fabric can be built for each port group separately.

The second phase of the algorithm considers each port group created in the port assignment phase separately, and finds a fabric to support the flows assigned to its ports. The fabric associated with a port group is an interconnected set of fabric nodes and links called a *module*, from which we obtain the name *module-building* phase for this part of the algorithm. The two phases are described in more detail in §4.1 and §4.2.

Two examples of *QuickBuilder* designs are shown below. The fabric in Figure 7 was developed by *QuickBuilder* with the same inputs that *FlowMerge* used to find the fabric in Figure 2. For this problem, *QuickBuilder*'s assignment of flows to ports led to two port groups, one of which is very large, containing all but two ports. The fabric contains one direct host-device link, and one very large module with three interconnected switches. Figure 6 is a solution to the SAN design problem for which *FlowMerge* designed the fabric in Figure 1. In this fabric, *QuickBuilder*'s port assignment created five port groups. Two port groups are supported by direct links, two larger port groups are supported by hubs, and the largest is supported by two switches connected to each other. In §5 we compare the *QuickBuilder* and *FlowMerge* solutions in more detail.

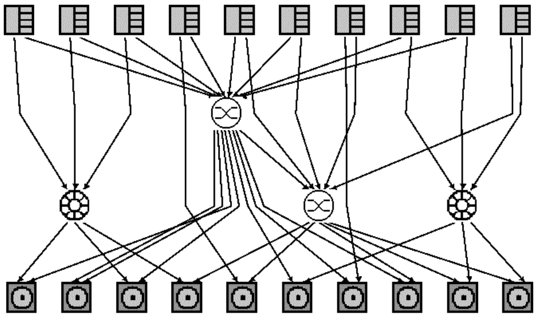

Figure 6: A sample SAN fabric produced by *QuickBuilder* (cf. Figure 1)

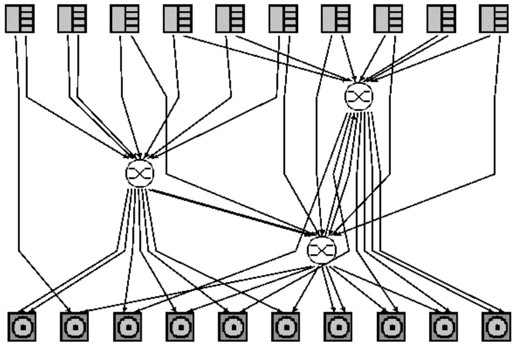

Figure 7: A second *QuickBuilder* SAN fabric (cf. Figure 2)

4.1 Finding port assignments

To find an assignment of flows to host and device ports, *QuickBuilder* considers flows one at a time, looking at each possible combination of host and device ports for each flow's assignment. It chooses the assignment among these that, when added to previous assignments, has the lowest estimated cost. *QuickBuilder* continues making the lowest estimated cost assignment for each flow until all flows have been assigned ports. Although the flows can be assigned in any order, we have found that considering them in order of decreasing bandwidth leads to cost-effective designs.

When estimating the cost of a flow being assigned to particular host and device ports, we account for the previous assignments of flows to these ports. If making this new assignment would cause the total bandwidth of flow to exceed the bandwidth available on either port, then the assignment is infeasible. Furthermore, this port assignment must not preclude the possibility of assigning all of the host's (or device's) unassigned flows to its ports. To determine whether the unassigned flows can be assigned, we apply an exhaustive bin packing algorithm, where the ports are bins, a port's capacity is the bandwidth unused by previously assigned flows, and the unassigned flows are the items to be packed. If there is no solution, this assignment is infeasible. Infeasible assignments have cost ∞.

If a port assignment is feasible, *QuickBuilder* estimates the cost of supporting the port groups before and after the port assignment is made. The cost of the assignment is the difference between the "after" and "before" cost estimates. The module cost estimation is similar to module construction; we describe both together in §4.2.

4.2 Building modules

The port assignment determined in the first phase of *QuickBuilder* uniquely determines the port groups. In this section, we describe how *QuickBuilder* creates a module to route the flows assigned to ports in a given port group. We also explain how, in the port assignment phase, *QuickBuilder* estimates the module cost for a given port group. For most port groups, the two processes involve the same computations.

When building a module or estimating the cost of a module for a port group, we assume for simplicity a single type of hub h and a single type of switch s to use in the module. Recall that bandwidth, number of ports, and cost of a type n fabric node are b_n, p_n and c_n, respectively. The module-building phase of the algorithm relies upon the assumption that there is a hierarchy among fabric elements, namely, $b_s > b_h$ and $c_s > c_h$. The building and estimation processes depend on properties of the port group. In particular, three cases are considered:

- **Case 1: Using a direct link.** If the port group has only two ports, then a module consisting of a single direct link between the two ports is sufficient. The cost of such a module is simply the cost γ of a link plus the costs of the host and device ports. The estimate of the module building cost is exact in this case.

- **Case 2: Using a (multi-)hub.** If the total flow bandwidth through the ports in the port group is less than b_h, then we use a hub or a *multi-hub*, which is a series of hubs, each connected to the next by a single link. The number of ports available on a hub is p_h; a multi-hub consisting of $i \geq 1$ hubs has $i(p_h - 2) + 2$ available ports. If the number of ports in a port group is k then $H = \lceil (k-2)/(p_h - 2) \rceil$ hubs are required. The module cost is the sum of the cost of the hubs Hc_h, the cost $(H-1)(2\pi_h + c_L)$ of connecting the hubs via links and hub ports, the cost $k(\pi_h + c_L)$ of connecting the host and device ports to the hubs including link and port costs, and the cost of the host and device ports in the port group. In this case, the *QuickBuilder* module cost estimate is also identical to the true cost of a module that would be built for this port group.

- **Case 3: Using a switch module.** If neither of the above conditions holds, then the module must contain at least one switch. We refer to a module that contains one or more switches as a *switch module*. To estimate the cost of a switch module, *QuickBuilder* estimates the number of switches that are needed to support the flows in the port group. In the interest of efficiency, we estimate this number of required switches without determining their exact connectivity in the module and how flows would be routed through them. To do so, we first make the simplifying assumption that all flows in the port group are routed through a single switch of infinite bandwidth. This helps us ignore the effects of flows traveling between multiple switches. *QuickBuilder* calculates the minimum number of ports that would be required to route the port group's flows through this infinite bandwidth switch, and then finds the minimum number of real switches that are required to provide that number of ports. Because there may, in fact, be multiple fabric nodes in the module and flows may travel between them, some of each node's bandwidth will be effectively "wasted" by this inter-node travel.

  To reduce the adverse effect of the infinite-switch assumption upon the estimate, *QuickBuilder* scales up the flows when making this calculation. For each port in the port group, it (temporarily) increases the bandwidth of each flow on that port uniformly by a fixed percentage (typically 10%) or until the capacity of the port is reached. Then, in order of decreasing total assigned flow bandwidth, ports in the port group are "connected" to the first switch port with enough remaining bandwidth to carry that port's flow. If multiple ports in the port group are connected to the same switch port, these ports are instead connected to the smallest required multi-hub which then is connected to the switch port. If k is the number of switch ports used, then $S = \lceil k/p_s \rceil$ is the minimum number of switches of type s needed to provide k ports. The estimated module cost is then the sum of the cost of the switches Sc_s, and the cost of links, hubs and ports used to connect the host and device ports to switch ports.

4.3 Building a switch module

When we are actually building a switch module for a port group, a more elaborate procedure than that of §4.2 is required to determine the exact connectivity of fabric nodes within the switch module. This procedure is outlined here. Switch module construction is a recursive procedure that introduces a series of switches in succession until all flows in the port group can be supported. Its input is a set of *external ports*, each with a set of flows entering the port (called *in-flows*) and exiting the port (called *out-flows*). In the initial call, the external ports are host ports with only out-flows and device ports with only in-flows. On subsequent calls to the procedure, some of the external ports are ports on switches *QuickBuilder* has already added to the switch module. Such ports may have both in- and out-flows.

When building a switch module, *QuickBuilder* first adds a new switch and connects external ports to this switch, selecting the best port to connect according to a merit function. When it connects an external port to the switch, it routes all out-flows (respectively, in-flows) from the port into (out of) the switch. Doing so creates *hanging flows.* These are flows that enter (respectively, exit) the switch via the connection of an external port to a switch port, but have not been assigned to a port by which to exit (enter) the switch. After an external port is connected to a switch port, any resultant hanging flows must be assigned to open switch ports. *QuickBuilder* continues connecting external ports to the switch until the switch is *saturated*, i.e., its bandwidth or port supply has been exhausted. When this occurs, *QuickBuilder* resets the set of external ports to be the current open external ports and switch ports that now contain in-flows and/or out-flows. It then invokes the switch-module-building procedure again on the new external ports.

4.4 Correctness and effectiveness

Like *FlowMerge*, *QuickBuilder* finds a feasible SAN fabric design when conditions (1) and (2) hold. The details of the proof are omitted here.

As with *FlowMerge*, we have no analytical bounds on the cost-effectiveness of *QuickBuilder* designs. However, empirical results indicate that *QuickBuilder* excels at solving large SAN design problems and those with dense flow requirements. In such problems, the flow assignment often results in one port group containing most or all of the ports. Thus, such problems require a large fabric through which almost all host and device ports are interconnected. *QuickBuilder* invokes its switch-module building routine to find this fabric. This routine generally makes very cost-effective use of switches for large port groups by minimizing the bandwidth "wasted" by flows routed through multiple switches in the module.

5 Evaluation of the algorithms

This section summarizes the results we obtained in applying integer programming, *FlowMerge* and *QuickBuilder* to several SAN fabric design problems.

The true test of our algorithms will happen only when their designs are implemented in a real business context and compared to those created manually by experts in the field. This comparison should include several metrics, including cost, performance, availability, scalability and even aesthetics. So far, we have only had a few opportunities to compare our designs to manual ones. Appia found much cheaper designs in these cases. In one notable example, a consultant worked for several days to produce a $4 million design on a problem that Appia solved for $1.4 million in a few minutes. (The consultant used several expensive, 64-port switches, and a completely symmetrical solution; Appia's *FlowMerge* found ways to achieve the same goals using much cheaper 16-port switches.) Nonetheless, we are loath to make strong claims about the benefits of our approach until we have had more opportunities to evaluate it on a wider range of real-world problems.

Nonetheless, our need to design and test Appia required us to generate a wide range of "realistic" test cases where we attempted to introduce elements of the real-world problems we had seen. We sought input from SAN designers in choosing our suite of test problems.

To that end, we generated 240 test problems in 24 categories, each with 10 test problems. The problem categories differed in size (defined by number of hosts and devices), a property we called *port saturation*, and characteristics of a problem feature called the *flow incidence matrix*. Since SANs are currently being designed on many scales, ranging from a handful to a few hundred servers and storage devices, we selected four size categories in this range. A host's or device's port saturation is defined to be its total bandwidth of associated flow requirements divided by the total bandwidth of its ports. The flow incidence matrix is a matrix whose rows correspond to hosts and whose columns correspond to devices. An entry in the matrix equals k if its corresponding host and device have k flows between them. We say a problem is *sparse* if its flow incidence matrix has relatively few positive entries scattered more or less uniformly throughout the rows and columns. Similarly, dense problems correspond to relatively dense and uniform matrices. A *clustery* flow incidence matrix is less uniform, corresponding to the situation when the hosts and devices can be partitioned into "clusters" that contain most of the flow requirements.

5.1 Effectiveness

Table 1 summarizes the computational results for test problems in each category for each of four methods. The first of these methods is the integer program (IP). This approach can solve only the smallest problems, but it does so optimally. A SAN fabric design problem with 10 hosts and 10 devices has over forty thousand binary variables and seventy five thousand constraints, a size far beyond the capabilities of today's commercial IP solvers. The LP (linear programming) relaxation of the IP can be solved for somewhat larger problems; its results are also presented in the chart. The LP relaxation is created by relaxing integrality constraints in the IP. It does not produce usable designs, but it provides a lower bound on the optimal design cost because it solves a less constrained problem. It can therefore be a useful benchmark for heuristics when the optimal cost is not known. We found, however, that the LP bound is quite weak – less than 35% of the optimal cost – for problems with high port saturation.

Statistics for running times of the respective approaches are also given for each category. Notice that in some of the categories, the IP and LP runs were terminated after 24 hours, before solutions had been found.

Results in Table 1 indicate that, on average, *FlowMerge* produces lower cost designs than *QuickBuilder* for smaller problems, whereas for large problems, *QuickBuilder* finds dramatically cheaper designs. The other problem characteristics do not conclusively predict which algorithm is preferable. Figure 8 makes this comparison of *FlowMerge* and *QuickBuilder* design costs more explicit for all but the smallest problems.

The relative advantages of *QuickBuilder* can be seen by a direct comparison of the fabrics in Figure 2 and Figure 7, found by *FlowMerge* and *QuickBuilder* respectively, for the same problem. The *FlowMerge* fabric uses five switches (the darker fabric nodes) and fifteen hubs, and costs $265,080. *QuickBuilder* produced a $133,440 fabric using only three switches. This problem has a very dense requirements matrix and high port saturation. In very dense problems, often every possible assignment of flows to ports results in one port group containing all or most of the host and device ports (to borrow terminology from *QuickBuilder*.) Thus, such problems require a large fabric through which almost all host and device ports are interconnected.

FlowMerge usually needs multiple fabric layers to connect all ports in a large port group. Its myopia in building independent layers and in performing only pairwise mergers impairs its effectiveness in such problems, as discussed in §3.3. For example, in Figure 2, the middle layer of fabric was introduced first without regard for how it would affect future layers, and subsequent layers were built independently of the others. Moreover, it overlooked cost-saving multi-flowset mergers in the outermost layers.

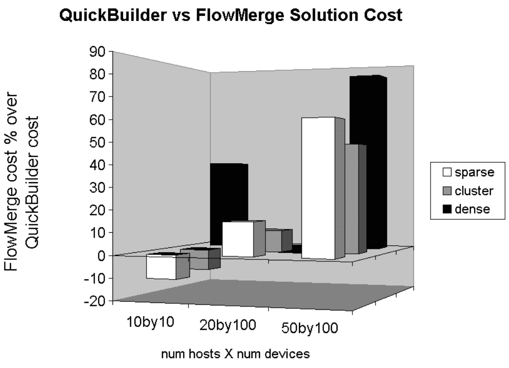

Figure 8: The percentage by which *FlowMerge* design costs exceed *QuickBuilder* design costs, averaged over twenty tests in each of nine categories. The categories are the combinations of the three largest problem sizes (number of hosts and devices each greater than five) and the three flow incidence matrix properties. The horizontal axis indicates the number of hosts and number of devices in the problem category. The bar shade and z-axis position indicate whether the flow requirements matrix is sparse, clustery or dense for that category.

Contrastingly, *FlowMerge*'s relative strengths for less dense problems are apparent when comparing the fabrics in Figure 1 and Figure 6 for the same problem. *FlowMerge*'s $63,720 fabric uses only one switch and three hubs, whereas *QuickBuilder* produced a more expensive $97,120 fabric. This might be explained by *QuickBuilder*'s quite myopic port assignment method, which ignores the flows that have yet to be assigned while making its current assignment. The port assignment determines the decomposition of ports into port groups, and thereby a decomposition of flows into disjoint subsets that can be routed through independent fabric elements. In this particular example, *FlowMerge*'s fabric has four distinct port groups, the largest containing sixteen ports. *QuickBuilder* created five port groups with twenty-four ports in the largest group. Large port groups typically lead to more fabric. *FlowMerge* excels at finding finer decompositions in less dense problems. Thus, *FlowMerge*'s strength is assigning flows to ports in such a way to yield smaller port groups, whereas *QuickBuilder* is better at building modules for large port groups when they are unavoidable. This supports an obvious strategy: run both algorithms, and pick the better solution.

Figure 9 and Figure 10 focus more closely on the smallest problems, with five hosts and five devices, because these problems can be solved optimally by the IP. The

Problem characteristics				Average (in $1000) and standard deviation of fabric cost				Average (in seconds) and standard deviation of solution time			
# hosts x # devices	avg # flows	flow matrix property	port satu-ration	optimal cost (± std dev)	LP cost (± std dev)	*FlowMerge* cost (± std dev)	*QuickBuilder* cost (± std dev)	optimal time (± std dev)	LP time (± std dev)	*FlowMerge* time (± std dev)	*QuickBuilder* time (± std dev)
5 x 5	14.3	sparse	high	27 ± 13	9 ± 1	31 ± 13	33 ± 12	9,080 ± 13,200	1.4 ± 1.26	0.1 ± 0.00	0.1 ± 0.00
5 x 5	14.0	sparse	low	11 ± 2	8 ± 0	13 ± 3	15 ± 3	1 ± 1	0.0 ± 0.02	0.1 ± 0.04	0.1 ± 0.00
5 x 5	21.8	clustery	high	40 ± 2	10 ± 1	41 ± 2	42 ± 3	19,800 ± 42,500	5.2 ± 1.39	0.3 ± 0.04	0.1 ± 0.00
5 x 5	21.5	clustery	low	12 ± 2	8 ± 0	17 ± 7	18 ± 9	3 ± 3	0.1 ± 0.02	0.3 ± 0.05	0.1 ± 0.00
5 x 5	24.2	dense	high	43 ± 1	11 ± 1	45 ± 1	46 ± 2	3,760 ± 2,370	9.9 ± 2.20	0.4 ± 0.05	0.1 ± 0.00
5 x 5	24.2	dense	low	16 ± 2	9 ± 0	38 ± 0	46 ± 2	1,690 ± 1,690	6.2 ± 0.97	0.4 ± 0.05	0.1 ± 0.00
10 x 10	28.1	sparse	high	unavail	19 ± 1	66 ± 13	77 ± 13	> 24 h	68.6 ± 61.1	0.1 ± 0.00	0.2 ± 0.00
10 x 10	28.5	sparse	low	unavail	17 ± 0	29 ± 3	32 ± 3	> 24 h	1.3 ± 0.83	0.1 ± 0.00	0.2 ± 0.00
10 x 10	40.9	clustery	high	unavail	21 ± 1	88 ± 20	94 ± 17	> 24 h	569 ± 576	0.1 ± 0.05	0.3 ± 0.03
10 x 10	39.7	clustery	low	unavail	17 ± 1	45 ± 11	55 ± 19	> 24 h	36.1 ± 33.5	0.1 ± 0.04	0.3 ± 0.03
10 x 10	94.1	dense	high	unavail	unavail	264 ± 16	138 ± 12	> 24 h	> 24 h	1.7 ± 0.15	0.6 ± 0.05
10 x 10	90.5	dense	low	unavail	unavail	89 ± 9	99 ± 10	> 24 h	> 24 h	1.3 ± 0.14	0.5 ± 0.00
20 x 100	180	sparse	high	unavail	unavail	467 ± 34	478 ± 39	> 24 h	> 24 h	7.5 ± 0.95	2.3 ± 0.17
20 x 100	161	sparse	low	unavail	unavail	342 ± 44	261 ± 29	> 24 h	> 24 h	6.2 ± 0.95	2.1 ± 0.36
20 x 100	226	clustery	high	unavail	unavail	503 ± 36	512 ± 61	> 24 h	> 24 h	15.1 ± 1.67	2.9 ± 0.15
20 x 100	217	clustery	low	unavail	unavail	375 ± 34	322 ± 57	> 24 h	> 24 h	12.7 ± 1.67	2.9 ± 0.32
20 x 100	214	dense	high	unavail	unavail	455 ± 58	441 ± 33	> 24 h	> 24 h	12.2 ± 1.22	2.7 ± 0.15
20 x 100	204	dense	low	unavail	unavail	246 ± 35	278 ± 42	> 24 h	> 24 h	9.5 ± 1.22	2.5 ± 0.16
50 x 100	448	sparse	high	unavail	unavail	1640 ± 231	1030 ± 20	> 24 h	> 24 h	258 ± 7.44	14.4 ± 0.16
50 x 100	402	sparse	low	unavail	unavail	1150 ± 59	718 ± 40	> 24 h	> 24 h	248 ± 6.51	21.9 ± 1.46
50 x 100	607	clustery	high	unavail	unavail	1560 ± 54	1010 ± 22	> 24 h	> 24 h	497 ± 29.4	19.6 ± 0.85
50 x 100	599	clustery	low	unavail	unavail	1010 ± 34	703 ± 32	> 24 h	> 24 h	487 ± 29.4	29.7 ± 2.90
50 x 100	539	dense	high	unavail	unavail	1900 ± 47	1080 ± 29	> 24 h	> 24 h	469 ± 20.7	18.3 ± 0.79
50 x 100	514	dense	low	unavail	unavail	1530 ± 45	805 ± 65	> 24 h	> 24 h	457 ± 20.7	33.2 ± 2.70

Table 1: Summary of computational results for four Appia design methods.
The results in each row are averaged across ten randomly-generated problems of the type shown under "problem characteristics." The first column indicates one of the four problem sizes used: 5 hosts, 5 devices; 10 hosts, 10 devices; 20 hosts, 100 devices; and 50 hosts, 100 devices. The second column reports the average number of flow requirements among problems in the category. The third column indicates qualititative properties of the flow incidence matrix. The fourth column describes the degree of port saturation on hosts and devices. For the "high" port saturation tests, 90% of port bandwidth of the hosts and devices is used, whereas for "low" saturation, only 40% of the port bandwidth is used.
The next four columns provide the average and standard deviation over the category tests of the cost of fabrics found by the four methods. The labels "optimal," and "LP," correspond respectively to the integer program and its LP relaxation. The term "unavail" means that we were unable to compute a result in less that 24 hours for tests in that category.
The last four columns contain the average and standard deviation of the solution times in each category, measured on an HP 9000 model with a PA8600 processor and 4GB of memory, running HP-UX 11.0. Numbers have been rounded to three significant digits.

former shows the relationship between the optimal design cost and the cost of the designs produced by the two heuristics. It indicates that for small problems, *FlowMerge* and *QuickBuilder* find solutions that are, on average 38% and 55% over the optimal fabric cost, respectively. The fourth bar contains the cost produced by the linear programming (LP) relaxation of the integer program, a lower bound on the optimal cost. In these small problems, the lower cost bound is, on average, half of the optimal cost.

Figure 10 contrasts the fabric costs for individual small tests for each of the four methods. In all of these small tests except for those that have both dense flow incidence matrices and low-saturated ports, *FlowMerge* and *QuickBuilder* find designs that average within 13% and 25% of the optimal design cost, respectively. The heuristics perform less well in the dense and low-saturation cases. The optimal fabrics use only inexpensive hubs, whereas both heuristics use a $24,000 switch, and many expensive switch ports, for each of these problems.

5.2 Efficiency

The graphs in Figure 11 show the algorithms' running times for all 240 test problems as a function of the number of flows in the problem. We chose number of flows as the independent variable because it is the most significant factor in the running times of the two algorithms. For the largest tests, with 50 hosts, 100 devices and 600 flows, *FlowMerge* finds a design in less than 10 minutes, and *QuickBuilder* finds one in less than 40 seconds. This means that adding a *QuickBuilder* run to a *FlowMerge* run is very cheap. Given the target use, it may make sense to use *QuickBuilder* interactively, and then invoke *FlowMerge* in batch mode for final review.

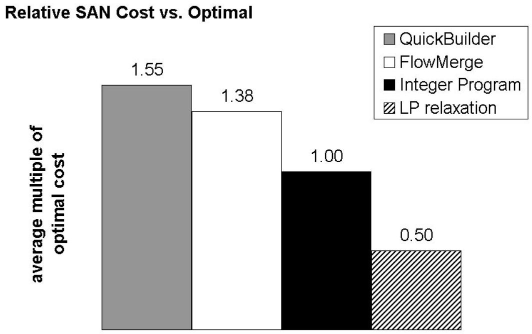

Figure 9: Cost comparisons of the resulting SAN designs for four different design algorithms, averaged across all 5 host, 5 device tests. "Integer program" (IP) produces optimal solutions; *FlowMerge* and *QuickBuilder* are heuristics that produce feasible solutions; LP relaxation produces a guaranteed lower bound, but (in general) infeasible solutions.

6 Future Work

We are actively pursuing several directions of future work:

- Extending the design tools to accommodate high availability requirements. A trivial solution often used for simple SANs is to replicate a single SAN fabric design, but this can become prohibitively expensive when port-count restrictions occur.

- Developing refinements that allow Appia to modify an existing design, rather than design one from scratch. This has obvious practical applications where an existing SAN is being extended; it also introduces some interesting tensions between the desire to produce a high-quality solution, and the desire to minimize the amount of rewiring required on the existing system while trying to use as many of its components as possible.

- Exploring the design of solutions that provide "slack," to allow graceful growth.

- Exploring topology-constrained solutions, such as Brocade's Core-Edge architecture, as one approach to producing designs that may be easier for people to modify by hand at a later date. This is a trivial problem for Appia compared to designing the topology itself – but its existing infrastructure makes it easy to supply this solution for people who prefer it.

- Packaging the tools so that they can be made more widely available, including integrating them more

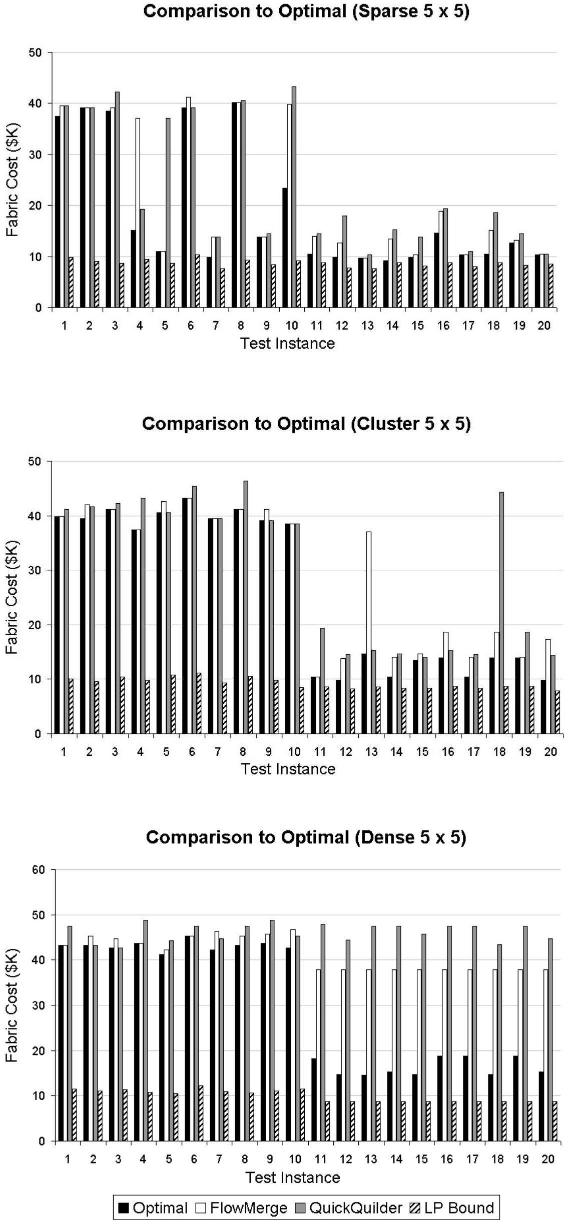

Figure 10: The graphs illustrate SAN solution costs for the four different Appia design algorithms across 20 different problems of the indicated type. For each graph, test instances 1–10 correspond to tests with high port saturation, and tests 11–20 have low-saturated ports. The problem scale was restricted to five hosts and five devices, to allow the optimal (integer programming) algorithm to complete in a reasonable time.

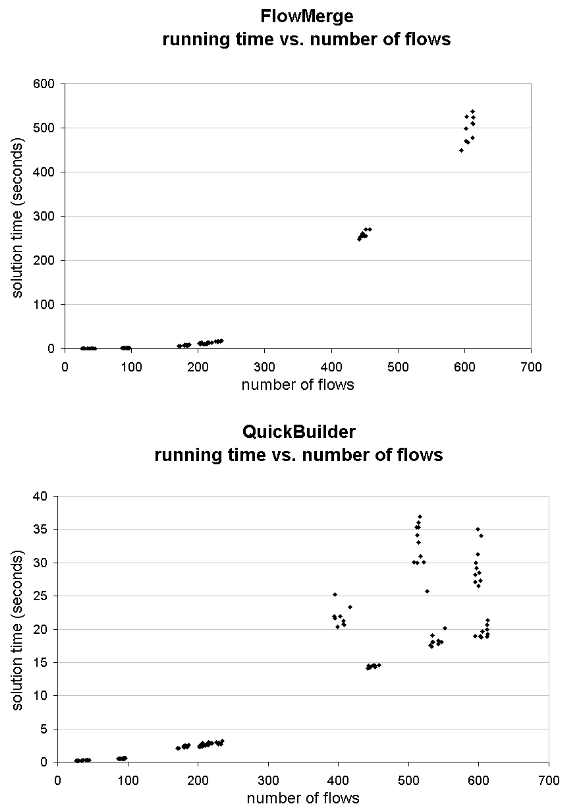

Figure 11: *FlowMerge* and *QuickBuilder* running times as a function of the number of flows.

tightly with our storage-system design tool suite [2, 4].

- Verifying that our algorithms work for new SAN protocols such as switched Ethernet, and for designing additional network types, such as LANs.

Opportunistically, we also expect to improve our algorithms' effectiveness and their runtime – but we feel that these are probably both "good enough" for us to be able to turn our attention towards the other opportunities listed above.

7 Conclusions

The Appia tool and its algorithms are able to design high quality SAN designs. Those designs are quite close to the optimal ones, in cases where we can evaluate them directly – and are several times less expensive than some manual designs we have seen, where over-provisioning by a factor of three "just to be safe" is a common approach.

In our interactions with customers and the domain experts who support them, we have learned that although it is helpful for Appia's SAN designs to be as cost effective as possible, it is probably even more important that they can be shown to be correct – the chance of human error has been greatly reduced. The value of this is extremely high in the complex, mission- and business-critical environments for which SAN design is done.

In summary, we feel that Appia and its algorithms solve a key, hard problem in storage systems – and one that is only going to grow in importance as the number, scale, and complexity of the SAN-based storage solutions grows.

7.1 Acknowledgements

We are endebted to Li-Shiuan Peh for getting the Appia project off the ground and aptly naming it, to Ren Wu and Eric Anderson for their help with implementation of our algorithms, to Rick Rosenthal for his helpful input for the exposition of this paper, and to several other colleagues at HP Labs for their support during this project.

References

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, and M.R. Reddy, *Applications of network optimization*, Network Models (M. O. Ball, T. L. Magnanti, C. L. Monma, and Nemhauser G. L., eds.), Handbooks in Operations Research and Management Science, vol. 7, North Holland, 1995, pp. 1–83.

[2] G.A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A Veitch, and J. Wilkes, *Minerva: an automated resource provisioning tool for large-scale storage systems*, ACM Transactions on Computer Systems (2001).

[3] A. Amiri, *A system for the design of packet-switched communication networks with economic tradeoffs*, Computer Communications **21** (1998), 1670–1680.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, *Hippodrome: running circels around storage administration*, File and Storage technologies (FAST) (Monterey, CA), January 2002.

[5] A. Atamtürk, *On capacitated network design cut-set polyhedra*, Research report, IEOR Department, University of California at Berkeley, December 2000, Available at http://ieor.berkeley.edu/˜atamturk.

[6] F. Barahona, *Network design using cut inequalities*, SIAM Journal on Optimization **6** (1996), 823–837.

[7] D. Bienstock, S. Chopra, and O. Gunluk, *Minimum cost capacity installation for multicommodity network flows*, Mathematical Programming **81** (1998), no. 2–1, 177–199.

[8] D. Bienstock and O. Gunluk, *Capacitated network design. polyhedral structure and computation.*, INFORMS Journal on Computing **8** (1996), 243–259.

[9] *Designing next-generation fabrics with Brocade switches*, White paper, Brocade Networks, San Jose, CA,

October 2001, http://www.brocade.com/SAN/pdf/CoreEdgeRev10901.pdf.

[10] S. Chopra, I. Gilboa, and S. T. Sastry, *Source sink flows with capacity installation in batches*, Discrete Applied Mathematics **85** (1998), 165–192.

[11] C.-H. Chu, G. Premkumar, C. Chou, and J. Sun, *Dynamic degree constrained network design: a genetic algorithm approach*, Proceedings GECCO-99. Genetic and Evolutionary Computation Conference. Eighth International Conference on Genetic Algorithms (ICGA-99) and the Fourth Annual Genetic Programming Conference (GP-99), vol. 1, 1999, pp. 141–148.

[12] N. Deo and S.L. Hakimi, *The shortest generalized Hamiltonian tree*, Proceedings of the 6th Annual Allerton Conference, 1968, pp. 879–888.

[13] *Overview of fibre channel technology*, Fibre Channel Industry Association, http://www.fibrechannel.com/technology/, accessed November 2001.

[14] P.C. Fetterolf and G. Anandalingam, *A Lagrangian relaxation technique for optimizing interconnection of local area networks*, Operations Research **40** (1992), 678–688.

[15] M.R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W. H Freeman and Company, San Francisco, CA, 1979.

[16] B. Gavish, *A general model for the topological design of computer networks*, IEEE Global Telecommunications Conference, no. 3, December 1986, pp. 1584–1588.

[17] ______, *Topological design of computer communication networks-the overall design problem*, European Journal of Operational Research **58** (1992), 149–172.

[18] B. Gendron, T. Crainic, and A. Frangioni, *Multicommodity capacitated network design*, Telecommunications Network Planning (B. Sanso and P. Soriano, eds.), Kluwer Academic Press, 1999, pp. 1–19.

[19] J.W. Herrmann, G. Ioannou, I. Minis, and J.M. Proth, *A dual ascent approach to the fixed-charge capacitated network design problem*, European Journal of Operational Research **95** (1996), 476–490.

[20] K. Holmberg and Di Yuan, *A Lagrangean heuristic based branch-and-bound method for the capacitated network design problem*, Proceedings of International Symposium on Operations Research, September 1996, pp. 78–83.

[21] J. M Kleinburg, *Single source unsplittable flow*, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, 1996, pp. 68–77.

[22] T. L. Magnanti, P. Mirchandani, and R. Vachani, *Modeling and solving the capacitated network loading problem*, Operations Research **43** (1995), 142–157.

[23] T. L. Magnanti and L. A. Wolsey, *Optimal trees*, Network Models (M. O. Ball, T. L. Magnanti, C. L. Monma, and Nemhauser G. L., eds.), Handbooks in Operations Research and Management Science, vol. 7, North Holland, 1995, pp. 503–615.

[24] T. L. Magnanti and R.T. Wong, *Network design and transportation planning*, Transportation Science **8** (1984), 1–55.

[25] S.C. Narula and C.A. Ho, *Degree-constrained minimum spanning tree*, Computers and Operations Research **7** (1980), 239–49.

[26] Li-Shiuan Peh, *The appia topology solver: implementation*, Technical report HPL–SSP–98–13, Hewlett-Packard Laboratories, Palo Alto, CA, September 1998, http://www.hpl.hp.com/SSP/papers/#Appia.

[27] V. Sridhar and J.S. Park, *Benders-and-cut algorithm for fixed-charge capacitated network design problem*, European Journal of Operational Research **125** (2000), 622–32.