Using Hardware to Configure a Load-Balanced Switch

THE LOAD-BALANCED SWITCH IS A PROMISING WAY TO SCALE ROUTER CAPACITY. IN THIS 100-TERABIT-PER-SECOND ROUTER, AN OPTICAL SWITCH SPREADS TRAFFIC EVENLY AMONG LINECARDS. WHEN THE NETWORK OPERATOR ADDS OR REMOVES LINECARDS, RECONFIGURING THE SWITCH CAN BE TIME CONSUMING, BUT A POLYNOMIAL-TIME ALGORITHM DRASTICALLY REDUCES THE REQUIRED MEMORY-INTENSIVE OPERATIONS, YIELDING A SWITCH-RECONFIGURATION TIME BELOW 50 MS.

Srikanth Arekapudi
Shang-Tse Chuang
Stanford University

Isaac Keslassy
Technion

Nick McKeown
Stanford University

Efficient router architectures should have predictable throughput and scalable capacity, as well as internal optical technology (such as optical switches and wavelength division multiplexing) that can increase capacity by reducing power consumption. Keslassy et al.[1] describe one such architecture—a 100-terabit-per-second (Tbps) Internet router with a single-rack switch fabric built from essentially zero-power passive optics that do not sacrifice throughput guarantees. Compared to currently available routers, this router has approximately 40 times more switching capacity than a single rack can contain, with throughput guarantees that no commercial router can match today.

The key to this scalability is the use of the *load-balanced switch*, which Chang et al.[2] first described. The sidebar "How a Load Balanced Switch Works" gives an overview of the switch's function. Keslassy et al.[1] extended the basic architecture of the load-balanced switch so that it has provably 100 percent throughput for any traffic pattern and doesn't reorder packets. It is scalable, has no central scheduler, is amenable to optics, and can simplify the switch fabric by replacing a frequently scheduled and reconfigured switch with a single, fixed, passive mesh of wavelength-division multiplexing (WDM) channels.

The load-balanced switch works by uniformly spreading packets over all linecards, and therefore must know which linecards are present and which are not. Unfortunately, the number of linecards will change as network operators add or remove linecards, or when linecards fail. If some linecards are missing, the switch fabric must be able to schedule the traffic uniformly over the linecards present. Keslassy et al.[1] describe a hybrid electro-optical architecture that solves this problem and will operate with any subset of linecards. Elsewhere, Keslassy, Chuang, and McKeown[3] describe an algorithm to configure the switch fabric and prove that it will always find a valid configuration in polynomial time.

When a linecard fails, network operators require recovery—which involves reconfiguring the optical switch—in less than 50 ms.[4-7] With the polynomial-time algorithm,[3] however, reconfiguration took more than 50 *seconds*. A possible solution is simply to convert the software algorithm to hardware, but that would be too slow by at least an order of magnitude because the algorithm is extremely memory intensive. A better alternative, as we show in this article, is to modify the hardware implementation.

The polynomial-time algorithm requires many repetitions of two graph-matching algorithms. The first finds the maximum flow in a graph, commonly through the Ford-Fulkerson algorithm.[8] The second algorithm decomposes a matrix into a minimal number of permutations, commonly through a Birkhoff-von Neumann decomposition.[9,10]

However, both the Ford-Fulkerson and the Birkhoff-von Neumann algorithms require a large number of memory accesses to find matches. To speed up the runtime, we adapted the original algorithms to minimize memory accesses. First, we modified the Ford-Fulkerson algorithm to work specifically for bipartite matches. On the basis of the binary matrix structure specific to our problem, we then used bit manipulation schemes to reduce the time needed to search for new matches.

Second, to decompose a matrix into permutations, we replaced the Birkhoff-von Neumann algorithm, which repeatedly finds a permutation using either a maximum size match or a simplified Ford-Fulkerson algorithm, with the Slepian-Duguid algorithm. This algorithm finds all the permutations at the same time, which requires only one iteration and reduces the number of preprocessing steps accordingly. We added a simple mechanism to search for the matrix elements not yet assigned to a permutation.

Our experimental results show that we can achieve a 50-ms recovery time for the 100-Tbps router for up to 640 linecards.

How a Load-Balanced Switch Works

As Figure A illustrates, the load-balanced router consists of a single stage of buffers sandwiched between two identical switching stages. The buffer at each intermediate input (center) is partitioned into N separate first-in, first-out (FIFO) queues, one per output. The switch contains N^2 of these *virtual output queues* (VOQs).

The two switch fabrics operate quite differently from a normal single-stage packet switch. Instead of picking a switch configuration on the basis of queue occupancy, both switching stages walk through a fixed sequence of configurations. At time t, input i of each switch fabric is connected to output $[(i + t) \bmod N] + 1$. In other words, the configuration is a cyclic shift, and each input is connected to each output exactly $1/N$th of the time, regardless of the arriving traffic. Each stage is thus a *fixed, equal-rate* switch. Although the two stages are identical, it helps to think of them as performing different functions. In this perspective, the first stage is a load balancer that spreads traffic over all the VOQs, while the second stage is an input-queued crossbar switch, in which each VOQ is served at a fixed rate.

When a packet arrives to the first stage, the first switch immediately transfers it to a VOQ at the (intermediate) input of the second stage. The intermediate input that the packet goes to depends on the load balancer's current configuration. The packet is put into the VOQ at the intermediate input according to its eventual output. Some time later, the second fixed, equal-rate switch will serve the VOQ. The packet will then be transferred across the second switch to its output, where it will depart the system.

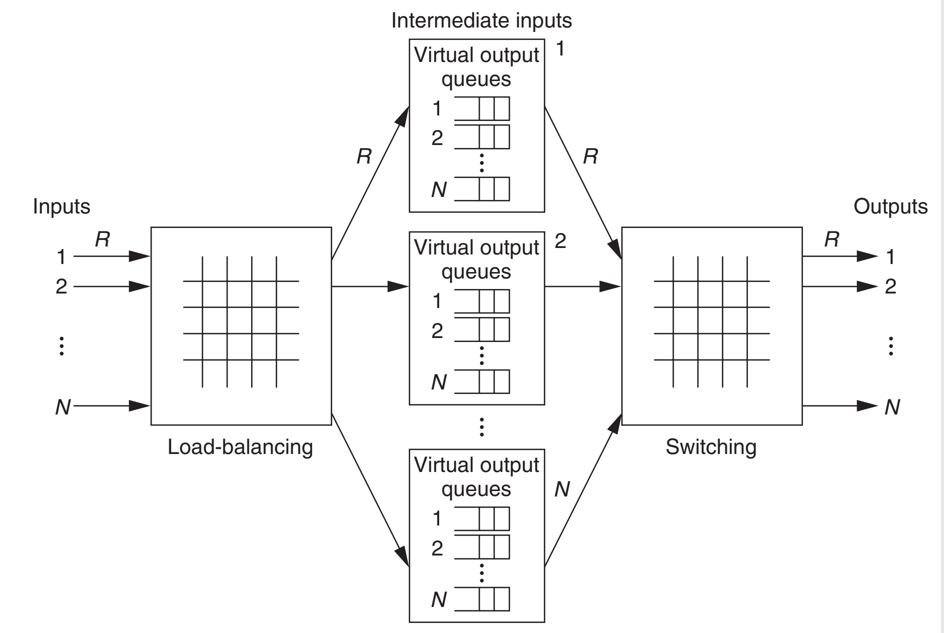

Figure A. A load-balanced switch.

$$C=\begin{pmatrix}1&1&1\\1&1&1\\1&1&0\end{pmatrix},\quad RL=\begin{pmatrix}2\\1\\1\end{pmatrix},\quad RR=\begin{pmatrix}2\\1\\1\end{pmatrix},\quad \Rightarrow \quad R=\begin{pmatrix}0&1&1\\1&0&0\\1&0&0\end{pmatrix}.$$

Figure 1. Sample connection assignment problem.

Configuration algorithm in brief

The configuration algorithm[1,3] involves G groups; group i contains L_i linecards, and the total number of linecards is

$$N=\sum_{i=1}^{G} L_i$$

The numbers L_1, L_2, … L_G are fixed for a given linecard arrangement. The objective of the algorithm is to create a schedule in which linecards spread packets evenly across all other linecards. Therefore, during every frame of N time slots, each sending linecard must connect exactly once to each of N receiving linecards, and vice versa. This is nearly the classical time-slot-assignment problem, also known as a Latin square problem when rates are equal, but with one twist: An additional constraint arises from the use of microelectromechanical system (MEMS) switches in the switch-fabric architecture. Within each time slot, the algorithm limits the rate from each transmitting group to each receiving group of linecards. Therefore, two linecards in a transmitting group might not be able to send simultaneously to two linecards in a receiving group.

An algorithm for constructing the schedule[3] first creates a schedule between sending and receiving groups by repeatedly solving the connection assignment problem described earlier. It then creates a schedule between sending linecards and receiving groups, and finally between sending linecards and receiving linecards. These last two steps repeatedly decompose matrices into a minimal number of permutations.

Connection assignment problem

The configuration algorithm of the load-balanced switch must solve the following connection assignment problem. Consider $2G$ nodes separated into G left nodes and G right nodes. The left nodes connect to the right nodes using a 0-1 capacity matrix $\boldsymbol{C}$ of size $G \times G$ whose rows correspond to the left nodes and whose columns correspond to the right nodes. The objective is to find 0-1 connection matrix $\boldsymbol{R}$ that is below capacity and satisfies a target number of connections per node. RL_i represents the target number of connections needed for left node i; similarly, RR_j represents the target number of connections needed for right node j. Figure 1 presents a sample connection assignment problem when $G = 3$. The first left node in the figure must make two connections, as the first element of RL specifies. The second right node must make one connection, as the second element RR shows. Therefore, the 0-1 solution matrix R has two elements on its first row, and one element on its second column.

Put mathematically, we want to solve the following problem. Find a 0-1 matrix $R \leq C$ such that

$$R=\begin{cases}\sum_{j'=1}^{G} R_{ij'} = RL_i & for\ all\ \ i\\ \sum_{i'=1}^{G} R_{i'j} = RR_j & for\ all\ \ j\\ R_{ij}\in\{0,1\} & for\ all\ \ i,j\end{cases}$$

This solution is not necessarily unique, but for the load-balanced switch configuration, the capacity matrix will always be large enough to guarantee a solution.[3]

Original Ford-Fulkerson algorithm

Given a capacity matrix, the Ford-Fulkerson algorithm can find the solution in polynomial time.[3] The algorithm consists of finding the augmenting paths from the source node to the sink node until there are no more augmenting paths. The resulting flow is the maximum flow. Either breadth-first search (BFS) or depth-first search (DFS) is suitable.

Modified Ford-Fulkerson algorithm

Our goal in modifying the Ford-Fulkerson algorithm was to implement it in hardware and reduce its runtime. Each entry in capacity matrix $\boldsymbol{C}$ is binary, so $\boldsymbol{C}$ represents a bipartite graph. We can therefore convert the Ford-Fulkerson graph to consider only the left and right nodes. Figure 2 shows the difference between a typical Ford-Fulkerson graph and our modified version. The left nodes are L_1, L_2, and L_3, and the right nodes are R_1, R_2, and R_3. Unlike the original Ford-Fulkerson algorithm, the modified version searches from the left to the right nodes, not from the source to the sink. It also disallows connections from the right to the left nodes.

Our modified Ford-Fulkerson algorithm has two parts. The first part uses a greedy approach to make connections between nodes. The second part uses back tracing to find the remaining connections.

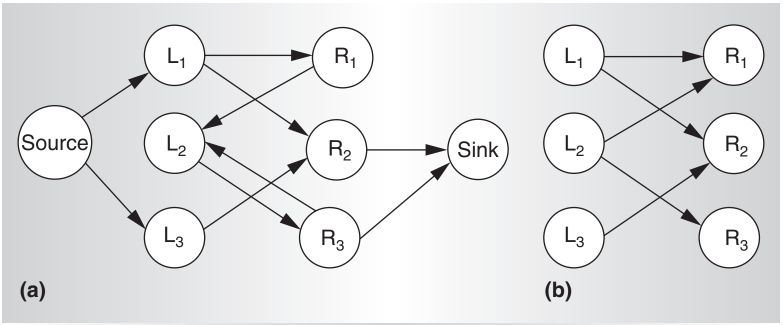

Figure 2. Original (a) and modified (b) Ford-Fulkerson algorithms.

$$P = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad RL' = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad RR' = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad C' = C - P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Figure 3. Matrix (***P***) of temporary connections after the greedy algorithm's application.

Greedy algorithm. For each left node, the greedy algorithm keeps adding as many temporary connections as possible to the right nodes. It can add a connection only if this connection is in the binary capacity matrix and adding it does not exceed the target number of left and right connections. Figure 3 shows the matrix ***P*** of temporary connections after applying the greedy algorithm. RL' and RR' represent the remaining target number of connections to be made for the left and right nodes. After the greedy algorithm's application, L_1 is connected to R_1 and R_2, and L_2 is connected to R_1, thereby meeting the target number of connections for L_1, L_2, R_1, and R_2. The only connections not yet met are for L_3 and R_3, as seen in RL' and RR'. The greedy algorithm cannot connect L_3 to R_3 because the only available connections from L_3 in the capacity matrix are to R_1 and R_2. After the greedy algorithm's application, the back-tracing algorithm makes the remaining target connections that RL' and RR' specify. The $\boldsymbol{C'}$ matrix specifies the connections the greedy algorithm does not use ($\boldsymbol{C'} = \boldsymbol{C} - \boldsymbol{P}$).

Back-tracing algorithm. Figure 4 depicts how we did back tracing using a simplified version of BFS. Initially the greedy algorithm finds the connections in the ***P*** matrix, the thin solid lines in Figure 4a. These edges are the current temporary connections. The dashed lines in the figure are the connections in the $\boldsymbol{C'}$ matrix. In our example, L_3 has no connection to R_3, but has connections to R_1 and R_2.

This is where the back-tracing algorithm starts. As the thick gray lines in Figure 4b show, the algorithm can trace back from

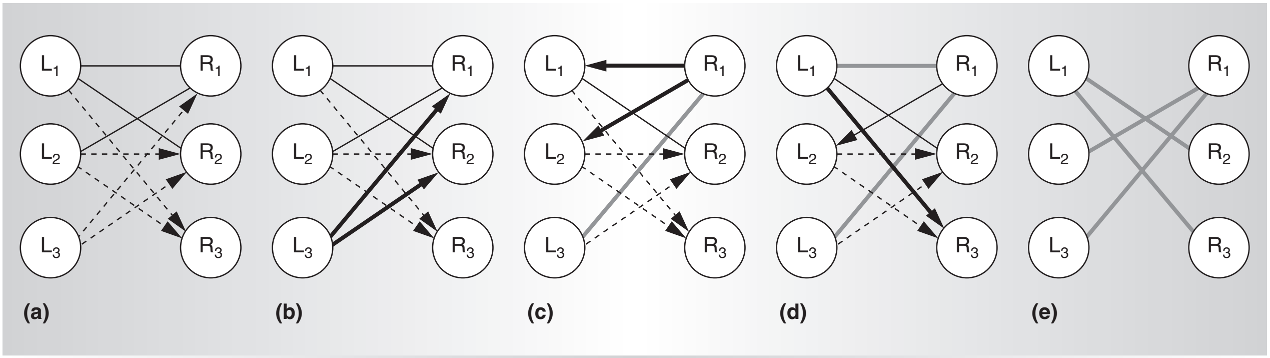

Figure 4. Back-tracing algorithm after applying greedy algorithm (a), tracing sending node L_3 (b), tracing R_1 backward (c), tracing L_1 (d), and updating temporary connections to yield the final schedule (e). Thin black lines represent temporary connections. Dashed lines are connections in the **C′** matrix. Thick gray lines are back traces.

either R_1 or R_2. If it traces back from R_1, it can go back only to L_1 and L_2, as Figure 4c shows. When back tracing from the right nodes, the algorithm considers only temporary connections. If the algorithm traces back to L_1, as Figure 4d shows, L_1 connects to R_3, which is a node that has yet to achieve its target number of connections. This ends the trace. The algorithm then updates temporary connections. Figure 4e shows the final connections.

The back tracing algorithm repeats for all other nodes that do not achieve their targets.

Implementation. Memories keep track of the matrix elements. Throughout both the greedy and back tracing algorithms, we store current capacity matrix C', which tracks the temporary connections made, and the number of connections that must still be made to each node. The back-tracing algorithm also requires a predecessor memory to remember the trace.

The greedy algorithm is memory-intensive because when it adds connections, it must search for the next available connection to a right node. With G right nodes, the search could require up to G memory accesses per left node and therefore a total of up to G^2 memory accesses.

The greedy algorithm uses bit manipulation schemes to reduce the number of memory accesses. It first arranges current capacity matrix C' associated with a left node as a bitmap of size G. It also represents the RR' array as a bitmap of size G, where it sets the bit if the corresponding RR'_j is positive. A logical AND between these two bitmaps gives a bitmap representation of the available connections. The algorithm then finds the next available connection by locating the first set bit in the resulting bitmap—a task it can accomplish in a single clock cycle through its priority encoder. Therefore, reusing the resulting bitmap can reduce the total number of memory accesses by a factor of up to G.

The back-tracing algorithm is memory intensive because it uses so many memory accesses. Part of the reason is the need to track the trace. Because the algorithm is based on a BFS, each search step might require adding up to G nodes to the predecessor memory. In one search step of a left node, for example, the algorithm might consider up to G right nodes.

In our implementation, the predecessor memory is a binary matrix of size $G \times G$. We implement this matrix using a memory structure that allows a memory write to an entire row, and a memory read of an entire column. Instead of writing each node individually, a search step writes the entire set of available nodes in parallel to the entire row. After a trace, to find the predecessor of a node in the trace, our implementation uses an encoder on the memory read's entire column to find the position of the single bit that is set in the column. This position corresponds to the predecessor's index. This bit manipulation scheme reduces each search step to a single memory access and thus reduces the total number of memory accesses by a factor of up to G.

Matrix decomposition problem

The configuration algorithm of the load-balanced switch must repeatedly decompose matrices into a minimal number of permutations. Keslassy, Chuang, and McKeown[3] proposed the Birkhoff-von Neumann decomposition algorithm for this task, but we believe that the Slepian-Duguid algorithm will yield a more efficient implementation.

To explain the matrix decomposition problem, we assume a 0-1 square matrix S and a positive integer n satisfying

$$S = \begin{cases} \sum_{j'} S_{ij'} = n & for\ all\ \ i \\ \sum_{i'} S_{i'j} = n & for\ all\ \ j \\ S_{ij} \in \{0,1\} & for\ all\ \ i,j \end{cases}$$

We want to decompose S into n permutation matrices, which means finding n permutation matrices P^k, $1 \le k \le n$, such that

$$P^k = \begin{cases} \sum_{j'} P^k_{ij'} = 1 & for\ all\ \ i,k \\ \sum_{i'} P^k_{i'j} = 1 & for\ all\ \ j,k \\ \sum_{k'} P^{k'}_{ij} = n & for\ all\ \ i,j \end{cases}$$

Although the decomposition is not necessarily unique, it always exists because the chromatic number of a bipartite graph equals its maximum degree. Consider matrix *S*, for example,

$$S = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

A permutation matrix has exactly one 1 in each row and column. Given matrix *S*, the algorithm can generate the following permutation matrices:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

In the load-balanced switch example, the binary matrices for *S* could be as large as 640 $\times$ 640; they are typically sparse, having at most sixteen 1s in each row and each column.

Earlier work

The polynomial-time Birkhoff-von Neumann decomposition algorithm[9,10] finds each permutation by applying a bipartite graph-coloring algorithm (Ford-Fulkerson or size matching, for example) on the *S* matrix. It then removes the permutation from the *S* matrix and repeats these steps until it finds all the *n* permutations. In our 100-Tbps router, the matrix size can be up to 640 $\times$ 640, requiring large memory structures for the coloring algorithms. Other graph-coloring algorithms use smaller data structures and parallelism.[11] However, none of these algorithms is fast enough for our implementation because all require at least one occurrence of the maximum-size-matching algorithm.

Greedy Slepian-Duguid algorithm

To reduce memory size, our implementation uses the sparsity of the matrices in the load-balanced switch example, representing the 1s of the binary matrix *S* as a list of row-column pairs. To reduce the number of memory accesses, we apply an algorithm that is based partially on the Slepian-Duguid algorithm.[12] Our algorithm attempts to produce *n* permutation matrices at once and uses the row-column pair list structure. The initial part of our algorithm uses a greedy scheme to assign the easily matched elements; the second part uses the Slepian-Duguid algorithm to reassign these elements and provide a solution.

Greedy algorithm. The greedy algorithm organizes the *n* permutations in a list structure consisting of row-column pairs; thus, inputs are essentially rows; outputs, columns. The algorithm arranges each permutation as an array of outputs. The *i*th array element, for example, refers to the output matched with input *i*. A valid permutation will not match more than one output to the same input. The algorithm attempts to find *n* permutations, so it maintains *n* such arrays arranged into matrix *A*, where each row corresponds to a different permutation and therefore to a different array.

Figure 5a illustrates how the greedy algorithm works. Matrix *A* is initially empty. The algorithm goes through the list of input-output pairs (i,o) and tries to assign each such

$$A_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad A_4 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad A_8 = \begin{pmatrix} 1 & 3 & 2 & 0 & 0 \\ 3 & 1 & 4 & 0 & 0 \\ 4 & 5 & 0 & 0 & 0 \end{pmatrix}, \quad A_{15} = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix}$$

(a)

$$B_0 = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 0 & 1 & 2 \end{pmatrix}, \quad B_1 = \begin{pmatrix} 1 & 3 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 5 & 5 & 1 & 2 \end{pmatrix}, \quad B_2 = \begin{pmatrix} 1 & 5 & 2 & 0 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}, \quad B_3 = \begin{pmatrix} 1 & 5 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 & 5 \\ 4 & 3 & 5 & 1 & 2 \end{pmatrix}$$

(b)

Figure 5. Matrix decomposition algorithm, which consists of greedy (a) and Slepian-Duguid (b) algorithms.

pair to a permutation for which both input i and output o are unassigned. It continues this process until it can assign no more (i,o) pairs.

Matrix $\boldsymbol{A}_k$ shows the state of the permutations after the algorithm assigns the kth (i,o) pair. If o is the (p,i)th element of A_k, then the pth permutation matches input i to output o in the kth step. Because matrix $\boldsymbol{A}_0$ is initially empty, the algorithm sets each matrix element to 0.

To understand further how the algorithm works, consider the assignment of the (2,1) pair in the fourth step. In A_4, the algorithm can assign the pair only to the second or third permutation, since output 1 is already scheduled in the first permutation. Assume the algorithm assigns this pair to the second permutation. It is possible that the algorithm cannot assign a specific pair. In A_8, for example, it cannot assign the (3,5) pair because the only permutation free for input 3 is the third permutation, and output 5 is already assigned in the third permutation. A_{15} shows the final state of the permutations, in which the (3,5) and (4,3) pairs remain unassigned.

Slepian-Duguid algorithm. The Slepian-Duguid part of our decomposition algorithm addresses each pair that the greedy algorithm could not assign. For each such pair (i^1,o^1) the algorithm

1. Identifies permutations P_{i^1} and P_{o^1} such that input i^1 is not assigned in P_{i^1} and output o^1 is not assigned in P_{o^1}.
2. Swaps input i^1 with i^2, where i^2 is an input such that (i^2,o^1) was already assigned in P_{i^1}. The algorithm must now track (i^2,o^1).
3. Swaps output o^1 with o^2, where o^2 is an output such that (i^2,o^2) was already assigned in P_{o^1}. The algorithm must now track (i^2,o^2).

The algorithm repeats steps 2 and 3 until there is an unassigned slot for (i^n,o^n) in either permutation P_{i^1} or P_{o^1}.

The $\boldsymbol{B}_i$ matrices are similar to the $\boldsymbol{A}_i$ matrices. Figure 5b shows how the algorithm assigns the (3,5) pair. It begins (step 1) by identifying the permutations that have input 3 and output 5 free. As $\boldsymbol{B}_0$, shows, P_3 (row 3) has input 3 free and P_1 (row 1) has output 5 free, so the algorithm assigns (3,5) in P_3 as $\boldsymbol{B}_1$ shows. As $\boldsymbol{B}_1$ also shows, two inputs are assigned to output 5 in P_3, so the algorithm swaps output 5 of P_3 with output 3 of P_1 for the same input. $\boldsymbol{B}_2$ shows the resulting matrix. Because output 5 is assigned only once in P_1, the algorithm stops here. Otherwise, it repeats the procedure of swapping until no permutation has multiple outputs assigned. $\boldsymbol{B}_3$ shows the final matrix.

Implementation. Memories keep track of which inputs and outputs are unassigned after each permutation. For each input i, a bitmap of size n stores the n permutations. If input i is not assigned in a given permutation, the algorithm sets the bit corresponding to this permutation in the bitmap. The algorithm works similarly for outputs. Then, in the greedy algorithm, an (i,o) pair can easily find a free permutation by taking a logical AND between the input and output bitmaps. By finding the first set bit in the resulting bitmap, the greedy Slepian-Duguid algorithm can match an (i,o) pair to a free permutation in a single clock cycle. Reusing the resulting bitmap reduces the total number of memory accesses by a factor of up to n.

Implementation results

We implemented the hardware using Verilog and synthesized it using a 0.13-μm process. For 40 groups and up to 640 linecards (up to 16 linecards per group), the modified Ford-Fulkerson algorithm used 10,000 gates and 24 Kbits of memory. The core for our greedy Slepian-Duguid algorithm used 25,000 gates and 230 Kbits of memory. Both cores ran within a 4-ns clock cycle time.

In our simulations, however, we used a cycle-accurate C model instead of the Verilog implementation, which was too slow given the number of experiments we wanted to run and the level of algorithmic complexity. We verified the C model's accuracy by comparing it with the Verilog implementation. We then used the C model to run several tests. The C model reports the number of clock cycles the simulation would run in Verilog. Because of the constraints in the load-balanced switch,[1] we generated different sets of results over 1,000 iterations with different ranges of linecards, between 0 and 640, spread over 40 groups. We spread the linecards randomly across the 40 groups, with at most 16 linecards in a group. For each range, we took the worst-case runtime.

Our simulations also assumed that a processor is connected to the cores uploading and downloading the necessary information into memories. To emphasize the processing time, our results do not include the times to transfer the initial matrices and to obtain the final results to and from the processor because we believe that transfer time is a negligible part of the overall processing time.

In Figure 6, we compare the runtimes of our implementation (new) with those of the earlier approach (old), which was simply to convert existing algorithms into hardware using Verilog. We measured total number of memory accesses and assumed that the hardware implementation can access memories in a pipelined manner and that each memory access requires one clock cycle. The number of memory accesses is thus the number of hardware clock cycles, each of which we assumed to be 4 ns.

The graph in the figure plots the most clock cycles needed in any of the tests we ran. We use a logarithmic scale to represent both plots on the same graph, and because the algorithms are polynomial, the plots appear logarithmic as well. We did not attempt to pipeline the Verilog implementation, however, and believe that we can reduce the time by an additional factor of at least two. Nonetheless, even without complete pipelining, the graph clearly shows that our implementation meets the 50-ms target over the required range of linecards.[1]

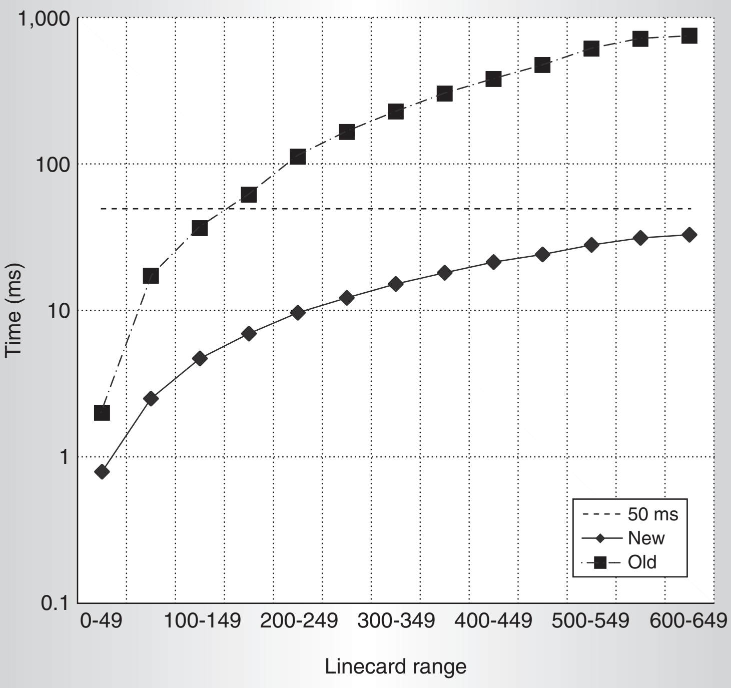

Figure 6. Worst-case configuration runtimes of 100-Tbps router for simple hardware conversion (old) and the proposed hardware-specific implementation (new).

Our hardware implementation already meets the 50-ms recovery time that network operators require. Further improvements are possible through the use of pipelining and multiport memories and by exploiting some parallelism in the greedy parts of the algorithms. Moreover, the schemes we propose lend themselves to generalization and might be suitable in accelerating the hardware implementation of other graph-coloring algorithms. MICRO

References

1. I. Keslassy et al., "Scaling Internet Routers Using Optics," ACM SIGCOMM '03, *Computer Comm. Rev.*, vol. 33, no. 4, Oct. 2003, pp. 189-200.
2. C.-S. Chang, D.-S. Lee, and Y.-S. Jou, "Load Balanced Birkhoff-von Neumann Switches, Part I: One-Stage Buffering," *Computer Comm. Rev.*, vol. 25, 2002, pp. 611-622.
3. I. Keslassy, S.-T. Chuang, and N. McKeown, "A Load-Balanced Switch with an Arbitrary Number of Linecards," *Proc. IEEE Infocom 2004,* IEEE Press, 2004; http://www.ieee-infocom.org/2004/Papers/41_3.PDF.
4. "Transport Systems Generic Requirements (TSGR): Common Requirements Criteria," Telcordia, GR-499 CORE, Issue 2, Dec. 1998.
5. "Synchronous Optical Network (SONET) Transport Systems: Common Generic Criteria," Telcordia, GR-253 CORE, Issue 3, Sept. 2000.
6. ANSI TR 68-2001, "Enhanced Network Survivability Performance," American Nat'l Standards Inst., Feb. 2001.
7. ITU-T Recommendation G.841, "Types and Characteristics of SDH Network Protection Architectures," Int'l Telecomm. Union, July 1995.
8. L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

9. C.S. Chang, J.W. Chen, and H.Y. Huang, "On Service Guarantees for Input-Buffered Crossbar Switches: A Capacity Decomposition Approach by Birkhoff and Von Neumann," *Proc. Int'l Workshop Quality of Service,* IEEE CS Press, 1999, pp. 79-86.
10. G.D. Birkhoff, "Tres Observaciones Sobre el Algebra Lineal," *Universidad Nacional de Tucuman Revista*, Serie A, vol. 5, 1946 pp. 147-151 (in Spanish).
11. R. Cole, K. Ost, and S. Schirra, "Edge-Coloring Bipartite Multigraphs in O(E log D) Time," *Combinatorica*, vol. 21, 2001, pp. 5-12.
12. J. Hui, *Switching and Traffic Theory for Integrated Broadband Networks*, Kluwer Academic Publishers, 1990.

Srikanth Arekapudi is a circuit design engineer at Advanced Micro Devices in the Computation Products Group. His research interests include low-power, high-performance circuit design. Arekapudi has a BTech in electronics and communication engineering from the National Institute of Technology, Warangal; an MS in electrical and computer engineering from the University of Massachusetts at Amherst; and a DEng in electrical engineering from Stanford University.

Shang-Tse (Da) Chuang is an architect at Nvidia Corp. His research interests include switch-scheduling algorithms, router architectures, and packet-buffer and linecard design. Chuang has a BS from the University of California at Berkeley, and an MS and a PhD from Stanford University, all in electrical engineering. He is a member of the IEEE.

Isaac Keslassy is an assistant professor of electrical engineering at the Technion, Israel Institute of Technology. His research interests include the design and analysis of high-performance routers, load balancing, packet buffering, and scheduling algorithms in optical and wireless networks. Keslassy has a PhD in electrical engineering from Stanford University. He is the American Technion Society-Women Division Career Development Chair and a member of the IEEE.

Nick McKeown is the ST Microelectronics Professor of electrical engineering and computer science at Stanford University. His research interests include the architecture, analysis, and design of high-performance switches and Internet routers, Internet Protocol lookup and classification algorithms, scheduling algorithms, optical switches, congestion control, and network design. McKeown has a PhD in electrical engineering and computer science from the University of California, Berkeley. He is the Robert Noyce Faculty Fellow at Stanford, a fellow of the Alfred P. Sloan Foundation and the Powell Foundation, and a recipient of the National Science Foundation's Career award. He is a fellow of the Royal Academy of Engineering and of the IEEE.

Direct questions about this article to Shang-Tse (Da) Chuan, Gates Bldg., Room 350, Stanford, CA 94305; stchuang@yuba.stanford.com.