Scheduling Algorithms for Input-Queued Switches: Randomized Techniques and Experimental Evaluation

Mark W. Goudreau* Stavros G. Kolliopoulos† Satish B. Rao‡

Abstract

A basic problem faced by designers of high-bandwidth switches and routers is to provide effective techniques for scheduling the routing of cells through crossbars. The problem is particularly important under heavy loads or when quality-of-service (QoS) is to be supported. Much previous work on scheduling has focused on maximum bipartite matching (MBM), maximum weight bipartite matching (MWBM), and heuristics to approximate MBM and MWBM solutions. In this paper, we introduce the Shakeup technique: a randomized approach that can be used in conjunction with a number of existing heuristics to substantially improve solution quality. The Shakeup approach is conceptually simple and is supported by both theoretical and experimental results. In addition, this paper provides for the first time a framework for experimental scheduler analysis. We give extensive head-to-head comparisons of stability ranges for a number of previously proposed schedulers, and work towards the development of benchmark traffic types.

"Just to stir things up seemed a great reward in itself."

—Sallust, Roman historian, 86–35 B.C.

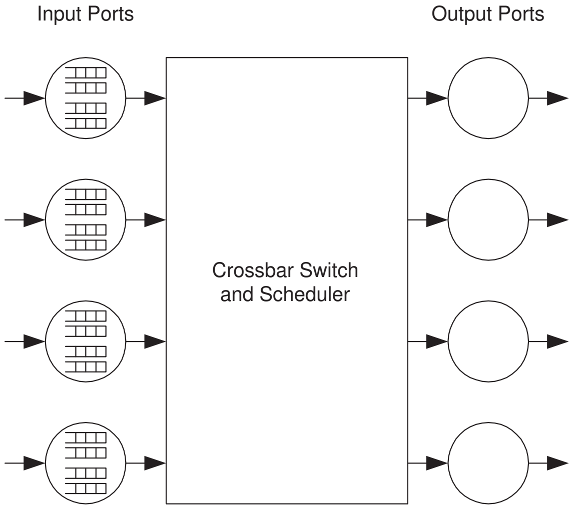

Figure 1: A crossbar switch that supports VOQ, $N = 4$.

1 Introduction

The growth of the Internet places continuously increasing demands on routers and switches. We begin to see switch designs capable of handling aggregate bandwidths of tens or hundreds of gigabits per second, with commensurate increase in the number of input and output ports [23, 28]. In addition, there will be demands to support flows with requested quality-of-service (QoS), which may include bandwidth, latency, and jitter restrictions. All services must be supported in an efficient way, such that the switch will be able to operate at nearly 100% capacity.

To support these increasing demands, the most promising switch architecture is one that provides scalable bandwidth as the number of switch ports grows. For the purpose of this paper, we assume the switching fabric is a crossbar switch that has N input ports and N output ports. The crossbar switch is capable of routing an arbitrary (partial) permutation of fixed-sized packets (cells) in one time step. When Karol et al. illustrated the limitations of using a single queue per input port [18], several research groups began to consider the use of N queues per input port—a technique known as virtual output queuing (VOQ) [3, 17, 21]. Intuitively, VOQ allows the switch to have greater flexibility when selecting a permutation. An example of a VOQ switch where $N = 4$ is shown in Figure 1.

For such VOQ switches, selecting appropriate permutations quickly is a fundamental problem. But what is the most appropriate permutation? One reasonable approach would be to route the most cells possible in the current time step. The problem is identical to calculating a maximum bipartite matching (MBM), a graph problem that has been studied in some detail (see Section 2). However, it is unlikely that the known MBM algorithms could be implemented in such a way that the scheduling calculation would not become a severe bottleneck. This has led to a number of heuristics that approximate the MBM; we characterize those algorithms as *unweighted.* Their main characteristic is that they do not consider the lengths of the different queues, only whether they are nonempty. Unweighted algorithms include Anderson et al.'s PIM [3], McKeown et al.'s SLIP [24], and a number of other approaches that greedily find a maximal matching.

A further drawback of MBM, demonstrated by McKeown [21], is that it can be *unstable*—that is, it can result in unbounded queue lengths—even for traffic flows for which no input or output port is expected to handle a throughput greater than its maximum capacity. Such sets of flows are said to be *admissible*, and can theoretically be handled by the crossbar switch. It was shown independently by Tassiulas and Ephremides [30, 29] and McKeown, Anantharam, and Walrand [22] that a generalization of the MBM approach, called maximum weight bipartite matching (MWBM), would be stable for all admissible flows. MWBM-based schedulers examine more than one bit of information per queue; they base their decisions also on the queue lengths. As with MBM, however, implementing full MWBM is problematic. This has resulted in a number of heuristics to approximate a MWBM, for example ILPF [25]. We call the MWBM-related heuristics *weighted.*

Unweighted and weighted approaches may be suitable in different contexts. For best-effort service, where there is no view of QoS, an unweighted approach is quite reasonable. If there is no call-admission control for flows, then the ability to support all admissible flows will not be a particularly useful feature, and maximizing instantaneous bandwidth is a reasonable design goal. On the other hand, if QoS is required and call-admission control is used, an approach like MWBM that is stable for all admissible traffic is highly desirable.

The algorithms considered in this paper choose a matching based only on the current state of the input queues. Some related work considers the scheduling problem under slightly different assump-

*C&C Research Laboratories, NEC USA, 4 Independence Way, Princeton, NJ 08540. goudreau@ccrl.nj.nec.com

†Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario L8S 4L7, Canada. stavros@mcmaster.ca

‡Computer Science Division, University of California, Berkeley, 679 Soda Hall, Berkeley, CA 94720. satishr@cs.berkeley.edu

tions. For example, Kam and Siu [16] and Li and Ansari [20] assume the scheduler has knowledge of flow requirements from the call-admission controller (CAC). Other approaches calculate a matching while leveraging off the matching from the previous time step; for example, see Kam and Siu [16] and Tassiulas [29]. The rationale in these cases is that the state of the input queues can only change by a small amount each time step; intuitively, the matching for time step i will very likely be a reasonable matching for time step $i + 1$ as well.

The contributions of this paper are as follows:

- We introduce the *Shakeup* technique, which can be used in conjunction with a number of the existing weighted and unweighted heuristics to improve their performance dramatically. The Shakeup technique assumes an initial matching is provided, and unmatched nodes have the ability to "knock out" existing matches. Intuitively, this provides a mechanism to jump out of local maxima in the matching heuristic. We discuss theoretical justification for the approach, which can be found in earlier work on bipartite matching. We then demonstrate the use of the approach with a variety of unweighted and weighted heuristics. Overall, the gains are impressive. In many cases, we see the range of switch stability increasing from approximately 85% to over 95% when Shakeup is used. We also demonstrate that even for stable flows, the use of Shakeup can greatly reduce the average cell delay.

- Through extensive simulation, we directly compare a number of different scheduling algorithms. Though earlier projects have performed some simulations that are similar in style [13, 21], the direct comparisons shown here are far more comprehensive, and provide useful insight into the tradeoffs between ease of implementation and quality of solution.

- We consider the range of stability to be a fundamentally important metric for scheduling algorithms. We provide an experimental definition of stability, and run stability tests for a wide range of scheduling approaches, traffic types, and traffic loads.

- We experiment with a number of different traffic types, designed to demonstrate the strengths and weaknesses of scheduling algorithms. The establishment of a benchmark suite of traffic types is an important contribution for the analysis of scheduling methods, particularly when the scheduling methods or traffic models are mathematically intractable. In particular, we examine "low-degree" traffic flows, which substantially degrade the performance of both unweighted and weighted heuristics.

2 Unweighted Approaches

In this section we provide background on the class of unweighted algorithms. We call an algorithm *unweighted* if it does not take into account the size of the different queues, only whether they are empty or not. We proceed to give some graph-theoretic terminology to be used in a representation of the switch. A *bipartite graph*, as shown in Figure 2, is an undirected graph for which the vertices can be partitioned into two sets and an edge goes only between the two sets. The switch can be modeled as a bipartite graph, where each input port is represented by a node in one set, each output port is represented by a node in the other set, and the existence of an edge between two vertices indicates that a cell exists to be routed between the corresponding input and output ports in the switch. A *matching* is a subset of edges such that no vertex is incident to more than one of the edges in the subset. Finding a matching corresponds to the selection of a partial permutation to route. A maximum (size) matching is a matching with the largest possible number of edges. In the rest of the paper, we use n and m to denote the number of

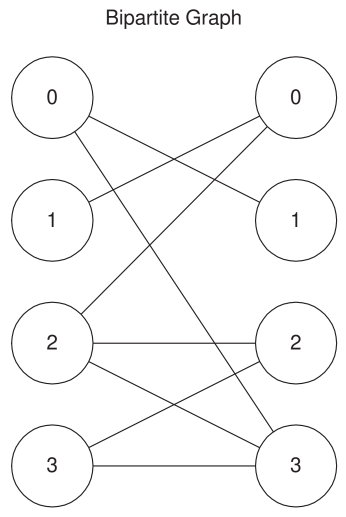

Figure 2: A bipartite graph abstraction of the scheduling problem. For this example, $N = 4$. Input port 0 has cells to send to output ports 1 and 3. Input port 2 has cells to send to output port 0. Input port 2 has cells to send to output ports 1, 2, and 3. Input port 3 has cells to send to output ports 2 and 3.

vertices and edges respectively in an underlying graph representing the state of the switch. For an $N \times N$ switch, $n = 2N$ and $m \leq N^2$.

The representation of the switch as the bipartite graph defined above allows for succinct algorithm descriptions. Almost all scheduling heuristics proposed aim to approximate a maximum or maximum weight (see next section) matching. A related concept is that of a *maximal* matching, to which it is not possible to add an edge while still maintaining a matching. We now survey the heuristics we test which are related to maximum matching.

MBM. We consider first computing an actual maximum size matching. The fastest algorithm to compute a maximum size matching runs in $O(n^{1/2}m)$ time worst-case [9, 10]. A drawback of maximum matching is that is can yield instability after admissible but non-uniform traffic even on a 3×3 switch [22]. Another problem with MBM is that the relevant algorithms are sequential in nature, requiring a centralized controller with knowledge of the state of each of the $O(n^2)$ queues at each time step. Because of these problems, several heuristics have been proposed that settle for a bipartite matching that is not maximum, but that can be calculated quickly (often in hardware) on real systems [3, 2, 7, 24, 31].

PIM. Parallel Iterative Matching (PIM) is a heuristic which will converge to a maximal matching in $O(\log n)$ iterations on average [3]. At every iteration unmatched inputs propose to every output for which they have a queued cell. If an unmatched output receives any requests, it grants to one by selecting *uniformly at random* over all requests. If an input receives more than one grant it randomly selects one among them.

SLIP. SLIP was developed by McKeown and analyzed extensively in [21]. Each input port and output port has a distinct priority wheel. An input sends a request to every output for which it has a queued cell. An output j chooses to accept a request from port i based on its priority wheel; if the grant is accepted by i, the priority wheel at j will be set to $(i + 1) \bmod N$. Input ports accept grants based on priority wheels, and advance their wheels in a similar fashion.

RND. The RND heuristic is similar to PIM. The difference is that every unmatched input i makes only a single request to a randomly chosen output port j for which i has a queued cell.

RR-RND. In the round-robin algorithm RR-RND, scheduling for a given time slot t is determined by a fixed priority schedule. The input i_1 with the highest priority chooses randomly among the outputs for which it has a cell queued. The input i_2 with the next highest priority chooses next among the not-taken outputs, and so on. At the next time slot $t+1$ the input-port priorities will be incremented by 1, modulo N. This approach can be implemented efficiently with a pipeline.

The PIM, RND and SLIP heuristics fall into the class of *iterative* heuristics. Iterative heuristics are not guaranteed to find a maximal matching in a single execution (iteration). Therefore, repeating them more than once may increase the size of the permutation. Of course there is a tradeoff with speed, since the control fabric may have to operate at a higher speed if multiple iterations are required for each permutation. The number of iterations is a parameter in our experiments (see Section 6).

3 Weighted Approaches

In this section we provide background on the class of weighted algorithms. We call an algorithm *weighted* if the sizes of the queues are queried during execution. In the MWBM problem the edges of the graph are assigned weights and the objective is to compute a matching of maximum weight. MWBM has the advantage that it has been proven theoretically to be stable under admissible traffic where the traffic flow from input i to output j can be modeled as a sequence of independent and identically distributed random variables [22, 25]. McKeown conjectures that MWBM is stable under all admissible flows [21]. On the other hand, MWBM algorithms are deemed impractical for hardware implementation. For input-output pair (i, j) let q_{ij} denote the occupancy of queue $Q(i, j)$, i.e. the number of cells queued at input i with destination j.

MWBM-LQF and MWBM-LPF. Two distinct flavors of maximum weight matching schedulers have been studied, each defining the weights on the edges of the bipartite graph in a different way. In the MWBM-LQF (Longest Queue First) approach [22] the weight w_{ij} of an edge is set equal to q_{ij}. In the MWBM-LPF (Longest Port First) algorithm [25], we define first weights for the ports. In particular set $S_i = \sum_{\text{output } j} q_{ij}$ for an input i. Analogously for an output j, $R_j = \sum_{\text{input } i} q_{ij}$. For the weighted bipartite matching instance define w_{ij} to be $S_i + R_j$ if $q_{ij} > 0$, and 0 otherwise. Both MWBM-LQF and MWBM-LPF are stable under admissible traffic with independent arrival processes [22, 25]. The fastest algorithm to compute a maximum-weight bipartite matching runs in $O(n^2 \log n + nm)$ time worst-case [8, 15, 14, 11]. If the maximum weight of an edge is bounded by C, the cost-scaling algorithm of Gabow and Tarjan runs in $O(\sqrt{n}mlog(nC))$ time worst case. See also [27, 12] for alternative cost-scaling algorithms.

ILPF. ILPF is a simple heuristic proposed in [25] to approximate the solution found by MWBM-LPF. The heuristic exploits the special assignment of edge weights as a function of port weights. The inputs and outputs are sorted according to their S and R values. The sorted list of outputs is traversed in decreasing order and output j is matched to the input i with the largest S_i value such that (i) i is still unmatched and (ii) $q_{ij} > 0$. No justification is offered in [25] as to why this heuristic may approximate the maximum weight matching found by MWBM-LPF. However, it is a well-known fact in matching theory [1] that ILPF will find the optimum solution if the underlying bipartite graph is complete, i.e. an edge exists between all input-output pairs.

RR-LQF. This scheduler is similar to the RR-RND algorithm described in Section 2. The difference is that each input i selects among the not taken outputs j the one with the maximum q_{ij} value.

4 The Shakeup Technique

Our Shakeup method comes in two varieties: one for unweighted and another for weighted heuristics. The motivation behind the Shakeup operation in both cases is to provide a way for the scheduler to escape from a local maximum, e.g. a maximal matching, and by doing so to speed the convergence of the queuing system to a steady state. Randomization is a crucial component of Shakeup. For our discussions on bipartite graphs, let A be the set of vertices representing input ports, and let B be the set of vertices representing output ports.

Unweighted Shakeup. Unweighted Shakeup (or Shakeup) assumes a bipartite graph along with some initial matching; the initial matching has perhaps been generated through some other heuristic. The basic idea is that each unmatched vertex in A will be allowed to force a match for itself, even if that means removing an existing match. Specifically, each unmatched vertex x in set A that is incident to at least one edge selects uniformly at random one of those incident edges, say (x, y). If $y \in B$ is unmatched, then (x, y) is added to the matching. If y is already matched through edge (z, y), $z \in A$, then (z, y) is removed from the matching and (x, y) is added. Finally, if more than one vertex in A attempts to match to the same vertex in B during a Shakeup, then one of the competing vertices in A is selected in a random fashion as the winner. In our experiments, we do not implement Shakeup in parallel but instead we go through the unmatched vertices in A in a randomly selected order. Note that unweighted Shakeup returns a matching that is at least as large as the initial matching.

Weighted Shakeup. Weighted Shakeup is slightly different in that it takes into account queue lengths. The basic idea behind extending unweighted to weighted is to give preference to those input and output ports that are heavily loaded; the unweighted Shakeup described above does not do this. Indeed, as all edges are treated equally in the unweighted Shakeup approach, it is possible that applying unweighted Shakeup to a weighted heuristic will make the solution worse. Weighted Shakeup, therefore, uses weights when selecting possible edges to add to the matching. Specifically, each unmatched vertex in set A that is incident to at least one edge randomly selects one of those edges. An edge is selected *with probability proportional to its relative size over all the destination queues*. That is, for input port i, if the queue holding cells destined for output j has length q_{ij}, then edge (i, j) is selected with probability $q_{ij}/(\sum_{k=1}^{N} q_{ik})$. Let (i, j') be the selected edge. If the vertex $j' \in B$ is unmatched, then (i, j') becomes part of the matching. If j' is already matched through edge (k, j'), then with probability $q_{ij'}/(q_{ij'} + q_{kj'})$, (k, j') is removed from the matching and (i, j') is added. Otherwise, the matching is not modified. Finally, unmatched vertices in A attempt to match to vertices in B in a randomly specified order. Thus ties between vertices competing for the same mate in B are resolved in a probabilistic manner.

Shakeups can be used in a stand-alone manner, or in conjunction with other heuristic approaches. We envision two general types of implementations: *orthogonal and hybrid*. For the orthogonal type, we alternate between phases of the "basic heuristic" (e.g., SLIP, PIM, etc.) and Shakeup. For example, if SLIP is the basic heuristic, the sequence of operations would be: SLIP, Shakeup, SLIP, Shakeup, SLIP, Shakeup, etc. For the hybrid type, the Shakeup phase is performed simultaneously with the basic heuristic phase. Each unmatched input port examines which output ports it can send cells to. If *any* of these output ports is not matched, the input port competes for it as defined by the basic heuristic, and does not do Shakeup in this iteration. On the other hand, if all the desired output ports are already matched, then the input port performs Shakeup during this iteration instead of the basic heuristic. Although the orthogonal and hybrid approaches are slightly different, the basic idea of using randomization to get out of local maxima is the same, and in practice both approaches are likely to have similar performance. For some implementations, however, the hybrid approach can require less communication than the orthogonal approach. In terms of physical implementation, the Shakeup schemes would be fairly simple to implement, using hardware similar to that needed by many of the existing unweighted heuristics. Some additional logic for a pseudo-random generator would be necessary, but we estimate that this logic does not have to be sophisticated. Our simulations obtained good performance by implementing Shakeup using the C built-in pseudo-random generator.

Theoretical Justification. At first sight one may argue that there is no apparent reason why unweighted Shakeup should improve scheduler performance and even converge to a maximal matching. Indeed, an unmatched input i sends only one request to an output j' at each iteration, and j' could very well be matched, in which case the size of the matching will remain the same. However by knocking off the matching the mate of j', a chain of knock-offs is triggered which leads, in a probabilistic sense, to the discovery of an *augmenting path*—a standard technique for increasing the size of a given non-maximum matching [5]. This idea has been used with success in the design of probabilistic algorithms. We proceed to give some basic facts.

Formally, a random bipartite graph $G_p(2N)$ is a graph with N vertices on each side of the bipartition, such that an edge between any two vertices on different sides is included with probability p independently of the others. The random graph model has been extensively studied, in part to capture the behavior of graph algorithms on "typical" inputs. See [6] for an excellent survey of the subject. In our problem p could be related to, although it is clearly different from, the probability that input i has a cell to send to output j for any pair (i, j) at the given time step at which we need to find a permutation. We should make clear that the bipartite graph of the switch cannot be accurately modeled by $G_p(2N)$, even for uniform traffic, due to the existence of queues. Consider a simplified unweighted Shakeup where in every iteration the unmatched vertices in A are considered in a fixed order. It was shown by Angluin and Valiant [4] that this simplified Shakeup delivers with high probability a perfect matching on $G_p(2N)$ in $O(N \log N)$ time, when $p > c \log N/N$, for some constant $c > 0$. Therefore, repeated application of unweighted Shakeup will give optimal throughput at a single time slot, under appropriate probabilistic assumptions on the traffic. This theoretical evidence is complemented by the experimental results in Section 6.

5 Framework for Comparison

To compare the schedulers we examine a variety of traffic types. In general, our goal here is to begin to define a benchmark suite of traffic types that could be used to compare different schedulers. These traffic types should be selected to represent both typical and worst-case situations.

For these simulations, each input port receives cells generated from its own source (that is, there are N sources). Each source is independent of the other sources, and generates at most one cell per time step.

We use a variety of probability density functions for selecting output port destinations. One of the major contributions of this paper is our investigation of *low-degree* traffic patterns, which incur poor performance for many schedulers. In this context, low-degree implies that the input and output ports of the bipartite graph have low degree. Intuitively, greedy heuristics should perform better for high-degree graphs than for low-degree graphs. High-degree graphs will have many good (i.e., optimal or near-optimal) matchings to choose from, while far fewer good matchings will exist for a low-degree graph. For a high-degree graph, there will be such a wealth of good solutions that greedily adding an edge to a matching probably will not greatly affect the solution quality. This is not the case for a low-degree graph, where early bad decisions can easily hurt the solution quality.

We consider three types of destination probability density functions:

High-degree balanced. Output ports are chosen with a uniform distribution over the N possibilities. All input ports and output ports will handle the same traffic load.

Low-degree balanced. Each input port is the source of exactly k flows and each output port is the destination of exactly k flows. All flows have the same rate. To create the flows at the start of the simulation, we use k random permutations, selecting each of the $N!$ possibilities with equal probability.

For example, let $N = 8$ and $k = 2$. Suppose the first permutation is $(1, 3, 7, 5, 6, 4, 2, 0)$ and the second permutation is $(5, 6, 2, 3, 1, 4, 0, 7)$. This implies input port 0 will send cells only to output ports 1 and 5 (with equal probability); input port 1 will send cells only to output ports 3 and 6; etc. Note that it is possible for an input port to have less than k possible destinations. In this example, input port 5 sends only to output port 4.

For this type of traffic, all input ports and output ports will have the same load. Also note that for the experiments described in Section 6, we use only a single random set of permutations; we make no claim that the set selected is representative of all traffic flows of this type.

Low-degree unbalanced. In [22], McKeown et al. provide an example of an admissible flow pattern that causes the queues to grow unboundedly if MBM is the scheduling algorithm. The example is essentially a case of unbalanced flows. This class is interesting because it tests the ability of schedulers to perform under uneven load on the ports. For our simulations, half of the input ports are the source of exactly k flows and half are the source of $2k$ flows. Each flow generates traffic at the same rate, so half the input ports are loaded at rate λ while the other half are loaded at 2λ. To create the flows at the start of the simulation, we use $2k$ random permutations, selecting each of the $N!$ possibilities with equal probability. For the heavily loaded input ports, we use all $2k$ flows defined by the permutations. For the lightly loaded ports, we use only the first k permutations to define the flows. Note that the output ports will support no less than k flows and no more than $2k$ flows.

For high-degree balanced or low-degree balanced traffic, a rate of $x\%$ implies all input and output ports are loaded at that rate. For low-degree unbalanced traffic, a rate of $x\%$ implies the heavily loaded input ports run at $x\%$, while the lightly loaded input ports run at rate $\frac{x}{2}\%$.

The burstiness of the traffic can also affect performance. We consider these two types:

Non-bursty (Bernoulli). At each time step, a cell is generated by a source with some fixed probability, independent of all previous traffic. The probability density function that determines the output port is the same for all cells generated by the source.

Bursty (Geometric). Traffic is generated as bursts of cells (ON bursts), followed by bursts of no cells (OFF bursts). The probability density function that determines the output port is the same for all ON bursts generated by the source. All cells within an ON burst will have the same destination. To create a burst with expected length m, the burst is discontinued at each time step with probability $1/m$. (The use of a geometric distribution to model bursty traffic can be found, for example, in [21].)

Most of our experiments assume non-bursty traffic. We will see later that our experiments confirm empirical findings by McKeown that suggest a linear relationship between expected burst length and expected cell delay for a range of scheduling approaches [21].

6 Experimental Results

The experiments provided some valuable insights into the relative performance of different algorithms. In Table 1 we provide

Scheduler	ρ_{hb}	ρ_{lb}	ρ_{lu}	δ_{hb}	δ_{lb}	δ_{lu}
MBM	100%	100%	97%	2.90	2.86	1.47
MWBM-LQF	100%	100%	100%	2.94	2.70	1.35
MWBM-LPF	100%	100%	100%	2.87	2.23	1.15

Table 1: Experimental stability rates (ρ) and average delays (δ) for MBM and MWBM approaches. High-degree balanced (hb), low-degree balanced (lb), and low-degree unbalanced (lu) flows are used for 32 ports. Average delays are for 100,000 time steps with 85% traffic load.

a summary of the experimental results for the three most powerful algorithms: MBM, MWBM-LQF and MWBM-LPF. The performance of these algorithms should be the benchmark against which heuristics are tested. It is interesting to notice how close the performance of MBM is to the performance of the weighted matching algorithms for the inputs we examined. Ideally, unweighted heuristics should perform well also, and often they do, but we demonstrate there is considerable room for improvement.

A general conclusion that can be drawn from the test data is that the addition of Shakeup on top of most of the examined heuristics improves performance. Another lesson from the experiments is that most of the previously known heuristics reach their full potential in terms of stability typically in 2-3 iterations, assuming a switch size of $N = 32$. In contrast, Shakeup typically exhibited slowly improving performance as the number of iterations increased beyond this point.

The experiments we designed fall into three major categories. We now elaborate on each category providing the description of the different settings and specific comments on the observed results.

6.1 Stability vs. Iterations

In the first category we test the various algorithms to determine the maximum input rate for which they are stable as a function of the number of iterations of the algorithm to select a single permutation. We call this the *stability rate*.

Our definition of stability is by necessity experimental. Let A be an algorithm running for i iterations. We consider the range of traffic rates from 0% to 100%, as defined for the traffic types described in Section 5. For every rate value r queried by the search mechanism, we simulate the system for at most $T = 256N^2$ steps. At every time step, we check to see if the sum of all the queue lengths Q has become greater than or equal to $16N^2$; if so, we declare the algorithm unstable at rate r. Also, we have *checkpoints* every 1,000 time steps; if the value of Q at a checkpoint is equal to or less than the value of Q at the previous checkpoint, we declare the algorithm stable at rate r. Finally, if we have not declared the algorithm either stable or unstable after T time steps, we declare the algorithm stable at rate r. If the algorithm is stable for rate r, the search mechanism considers only larger rates in the future. If the algorithm is unstable, we consider only smaller rates in the future. For efficiency, the actual implementation does not perform a linear search on all possible rates; it does a binary search instead.

In summary, the experimental space for the first category of experiments consists of all the possible values of the tuple $(A, i, r, \text{flow type})$. The number of iterations i ranges from 1 to 8. Observe that for MBM, RR-RND, and RR-LQF, iterations are meaningless since maximal matchings are computed in the first iteration. For those algorithms we get a horizontal line in the graph. The two axes show rate and number of iterations. The relevant graphs are in Figures 3-8. In each graph for this category we aggregate the results for related algorithms A. Typically there will be three algorithms in a graph, called X, X-HYBRID and X-SHAKE. X is any vintage algorithm like PIM or SLIP. X-SHAKE implements the orthogonal use of Shakeup: at every one of the i iterations we run X followed by an execution of unweighted Shakeup. (We note that if the orthogonal shakeup operation is considered as a separate iteration, then $2i$ iterations are actually used.) X-HYBRID implements the hybrid use of Shakeup: at each of the i iterations we run the basic heuristic X "merged" with Shakeup as outlined in Section 4.

Unweighted algorithms. The graphs for this group of experiments demonstrate that the addition of Shakeup to the unweighted heuristics can typically improve the maximum stability rate by 10%. The increase becomes palpable after already two iterations and slowly increases thereafter with the number of iterations. As expected, the increase is smaller for the "easy" high-degree balanced input, on which almost all unweighted heuristics achieve by themselves stability rate of 95% or higher after 3 iterations.

Weighted algorithms. The graphs for the ILPF algorithm (Fig. 7) show that Shakeup provides more modest improvements than in the unweighted case, of the order of $2 - 3\%$. This is not surprising given than ILPF is a heavy-duty centralized algorithm that does global sorting. As expected, weighted Shakeup (SHAKE-WT) performs slightly better than pure Shakeup which can marginally degrade performance every other iteration (due to the fact that it blindly unmatches the heavily loaded ports matched by ILPF). The situation is similar for RR-LQF although the gains from (weighted) Shakeup are larger, of the order of $5 - 6\%$.

6.2 Average Delay vs. Iterations

The *delay* of a cell is the number of steps it waits until it is routed. A cell that is routed upon arrival has delay 0. We take *average delay* to mean the average cell delay over all cells that were successfully routed within the time window of the simulation. (In general, the input queues can still contain cells that do not contribute to this value.)

The stability experiments outlined above give information that allows in most cases a clear ranking of the algorithms based on the maximum input rate they can support. However, this ranking would not give the full picture. For a given rate r it could very well be the case that both algorithms A and A' are declared (eventually) stable but they differ significantly in the average delay they incur on the packets. On the other hand, testing the algorithms solely based on average delay and ranking them accordingly could be misleading as well. First, an algorithm may actually settle for a large but stable average delay for a given rate r, information that is lost by looking at the average delay only. Second, what is the rate r at which one should examine the average delay incurred by A? If r is high enough so that A is unstable, the average delay obtained will be essentially meaningless and A will be judged unfairly under inputs it cannot possibly handle. Using the data from the stability results we avoid this pitfall by choosing $r = 85\%$ for all the average delay graphs, a value at which almost all of the algorithms tested were shown to be stable. In summary, the experimental space for the Average Delay Graphs consists of the possible values of the tuple $(A, i, r = 85\%, \text{flow type})$. The two axes show average delay (in time steps) and number of iterations. The relevant graphs are in Figures 9-14.

The average delay graphs confirm the picture given by the stability graphs for the advantages gained by Shakeup. The gains are more tangible for the unweighted algorithms. The improvements obtained by the application of Shakeup range from roughly 10% to a quite dramatic 10-fold improvement for SLIP and RR-RND on low-degree balanced inputs.

6.3 Average Delay vs. Burst Size or Ports

In this final category, we continue the qualitative exploration of the different algorithms by plotting average delay against two other parameters of importance.

In the first case, the experimental space consists of all possible values of the tuple ($A, i = 9, r = 85\%$, burst size, flow type). The axes show average delay and burst size. See Figures 15 and 16. The number of time steps is set again at $100,000$. The burst size increases by increments of 17 in the range $[17, 136]$. The results are as expected, i.e. for all algorithms the average delay increases as a function of the burst size, and this increase happens essentially at the same rate for all the algorithms. The curves for the heavy-duty MBM and MWBM algorithms are essentially the same as the ones for the rest, which implies that the increase of the average delay with the burst size is essentially unavoidable. The conclusion is that increased buffer space is necessary to store packets if bursty traffic is anticipated.

In the second case, the experimental space consists of all possible values of the tuple $(A, i = 9, r = 85\%, p, \text{flow type})$ where p denotes number of ports. The axes show average delay and number of ports. We test for $p = 4, 8, 16, 32, 64$. Figures 17 and 18 present the results for unweighted and weighted algorithms, respectively. The results demonstrate that most of the algorithms scale reasonably well, i.e. the average delay increases very slowly as the number of ports grows. There are some exceptions though. The average delay for RR-RND in Fig. 17 increases sharply with the number of ports for low-degree balanced flows. RR-RND is by definition a very rigid algorithm in the way it prioritizes input ports. As the number of ports increases, this inflexibility incurs a higher overhead. The average delay for SLIP in Figure 17 fluctuates between 30 and 140 for low-degree balanced and between 3 and 26 for unbalanced inputs. SLIP's behavior is hard to predict a priori due to the different priority wheels maintained by each port. If these wheels are aligned the wrong way—an event which apparently occurs on the same traffic type simply as N grows—SLIP's performance suffers. The variation in average delay for the weighted-type algorithms in Fig. 18 is bounded by 5; therefore these algorithms scale in a satisfactory manner.

7 Conclusions

The first major contribution of this paper is the presentation a simple and theoretically justified approach, called Shakeup, that introduces randomness into scheduler heuristics. Shakeup allows the heuristics to escape from local maxima, substantially improving the quality of solution. Through extensive simulations, we demonstrate that the use of Shakeup is advantageous for all examined heuristics in terms of both stability and average delay.

The second major contribution of this paper is its comparative study of scheduling algorithms and heuristics, with and without Shakeup. We point out the importance of low-degree traffic patterns, which induces poor performance in many of the heuristic schedulers.

One issue we do not address in detail is the feasibility of implementing Shakeup on real systems. Currently, hardware implementation of random-number generators is rare. However, we believe that the benefits of randomization in the context of switch design, for which the input flows are often unpredictable, are considerable. There are many algorithms that perform provably better on random inputs than on worst-case inputs. Moving randomization into the algorithms is often an effective means of dealing with arbitrary inputs (for an overview of randomized algorithms, see [26]). Despite the fact that a number of machines have been built that mechanically generate random numbers (see Knuth [19] for an overview), we believe greater emphasis should be put on the implementation problem. In the context of Shakeup, we conjecture that a fast but low-quality generator would provide sufficient performance. If the performance metric is to route the most cells possible in any given time step (the MBM problem), an iteration of Shakeup can never make the quality of the solution worse, regardless of the random-number generation mechanism used.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[2] M. Ali and H. Nguyen, "A neural network implementation of an input access scheme in a high-speed packet switch," in *Proc. of GLOBECOM 1989*, pp. 1192–1196, 1989.

[3] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High-speed switch scheduling for local-area networks," *ACM Transactions on Computer Systems*, vol. 11, no. 4, pp. 319–352, November 1993.

[4] D. Angluin and L. G. Valiant, "Fast probabilistic algorithms for Hamiltonian circuits and matchings," *Journal of Computer and System Sciences*, vol. 18, pp. 155–193, 1979.

[5] C. Berge, "Two theorems in graph theory," *Proceedings of the National Academy of Science*, vol. 43, pp. 842–844, 1957.

[6] B. Bollobás, *Random Graphs*. Harcourt Brace Janovich, 1985.

[7] T. X. Brown and K.-H. Liu, "Neural network design of a Banyan network controller," *IEEE Journal on Selected Areas of Communication*, vol. 8, no. 8, pp. 1428–1438, October 1990.

[8] R. G. Busaker and P. J. Gowen, "A procedure for determining minimal-cost flow network patterns," Tech. Rep. ORO-15, Operational Research Office, Johns Hopkins University, Baltimore, MD, 1961.

[9] E. A. Dinitz, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Math. Dokl.*, vol. 11, pp. 1277–1280, 1970.

[10] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM Journal on Computing*, vol. 4, pp. 507–518, 1975.

[11] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, pp. 596–615, 1987.

[12] A. V. Goldberg and R. Kennedy, "Global price updates help," *SIAM Journal on Discrete Mathematics*, vol. 10, pp. 551–572, 1997.

[13] D. Guo, Y. Yemini, and Z. Zhang, "Scalable high-speed protocols for WDM optical star networks," in *Proceedings IEEE INFOCOM'94*, vol. 3, pp. 1544–1551, 1994.

[14] M. Iri, "A new method of solving transportation-network problems," *Journal of the Operations Research Society of Japan*, vol. 3, pp. 27–87, 1960.

[15] W. S. Jewell, "Optimal flow through networks," Tech. Rep. 8, Operations Research Center, MIT, Cambridge, MA, 1958.

[16] A. C. Kam and K.-Y. Siu, "Linear complexity algorithms for bandwidth reservations and delay guarantees in input-queued switches with no speedup," in *Proceedings of the 6th International Conference on Network Protocols*, pp. 2–11, 1998.

[17] M. Karol, K. Eng, and H. Obara, "Improving the performance of input-queued ATM packet-switching," in *Proceedings IEEE INFOCOM'92*, pp. 110–115, 1992.

[18] M. J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input versus output queueing on a space-division packet switch," *IEEE Transactions on Communications*, vol. COM-35, no. 12, pp. 1347–1356, December 1987.

[19] D. E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison Wesley, third ed., 1998.

[20] S. Li and N. Ansari, "Input-queued switching with QoS guarantees," in *Proceedings IEEE INFOCOM'99*, vol. 3, pp. 1152–1159, March 1999.

[21] N. W. McKeown, *Scheduling Algorithms for Input-Queued Cell Switches*. PhD thesis, University of California at Berkeley, 1995.

[22] N. McKeown, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," in *Proceedings IEEE INFOCOM'96*, (San Francisco, CA), pp. 296–302, March 1996.

[23] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz, "The Tiny Tera: A packet core switch," in *Hot Interconnects IV*, (Stanford University), pp. 161–173, August 1996.

[24] N. McKeown, P. Varaiya, and J. Walrand, "Scheduling cells in an input-queued switch," *Electronic Letters*, vol. 29, no. 25, pp. 2174–2175, December 1993.

[25] A. Mekkittikul and N. McKeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *Proceedings IEEE INFOCOM'98*, (San Francisco, CA), pp. 792–799, March 1998.

[26] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge Unveristy Press, 1995.

[27] J. B. Orlin and R. K. Ahuja, "New scaling algorithms for the assignment and minimum mean cycle problems," *Mathematical Programming*, vol. 54, pp. 41–56, 1992.

[28] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. D. Troxel, D. Waitzman, and S. Winterble, "A 50-Gb/s IP router," *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pp. 237–248, June 1998.

[29] L. Tassiulas, "Linear complexity algorithms for maximum throughput in radio networks and input queued switches," in *Proceedings IEEE INFOCOM'98*, vol. 2, pp. 533–539, 1998.

[30] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pp. 1936–1948, December 1992.

[31] T. P. Troudet and S. M. Walters, "Neural network architecture for crossbar switch control," *IEEE Transactions on Circuits and Systems*, vol. 38, no. 1, pp. 42–56, January 1991.

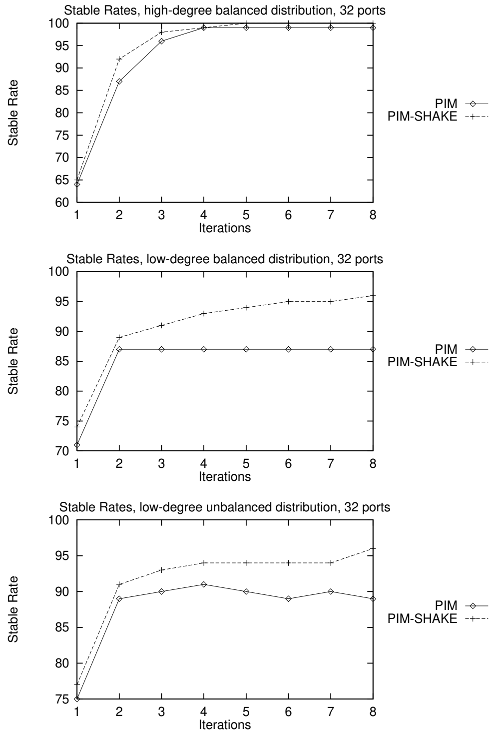

Figure 3: Stability levels for PIM in regular and Shake varieties.

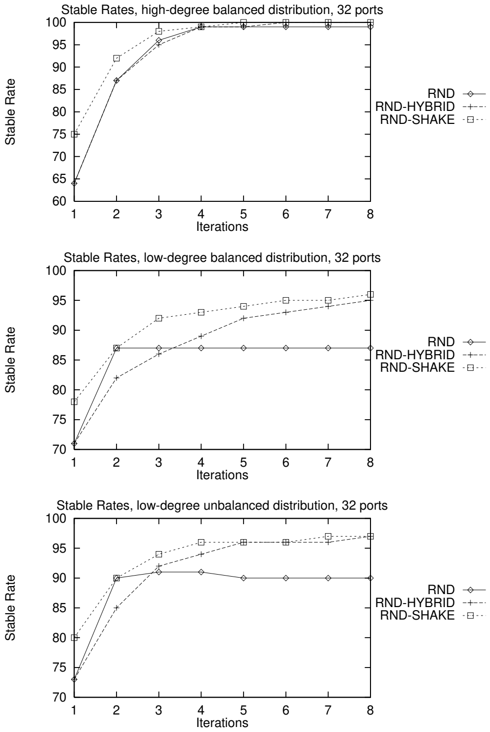

Figure 5: Stability levels for RND in regular, Shake, and hybrid varieties.

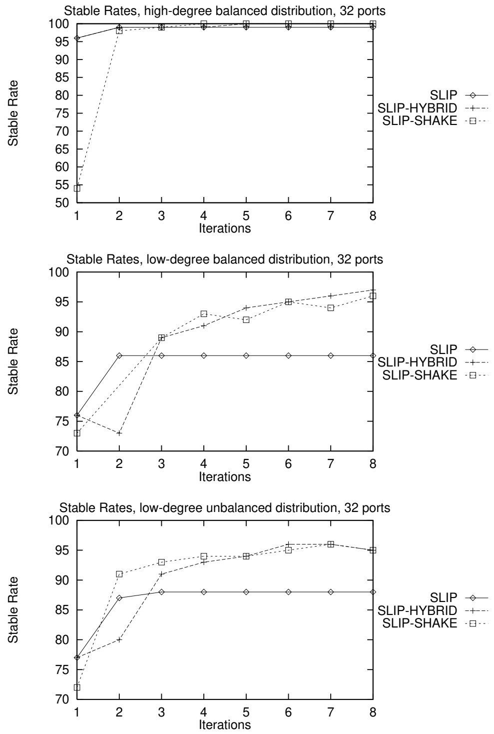

Figure 4: Stability levels for SLIP in regular, Shake, and hybrid varieties.

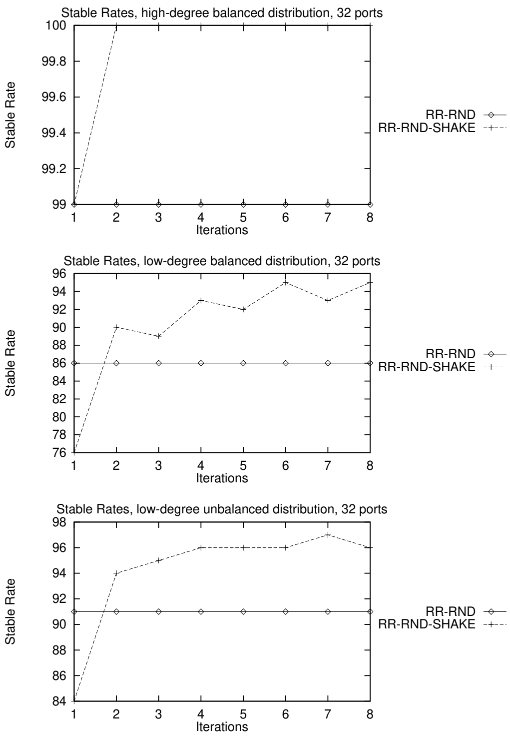

Figure 6: Stability levels for RR-RND in regular and Shake varieties.

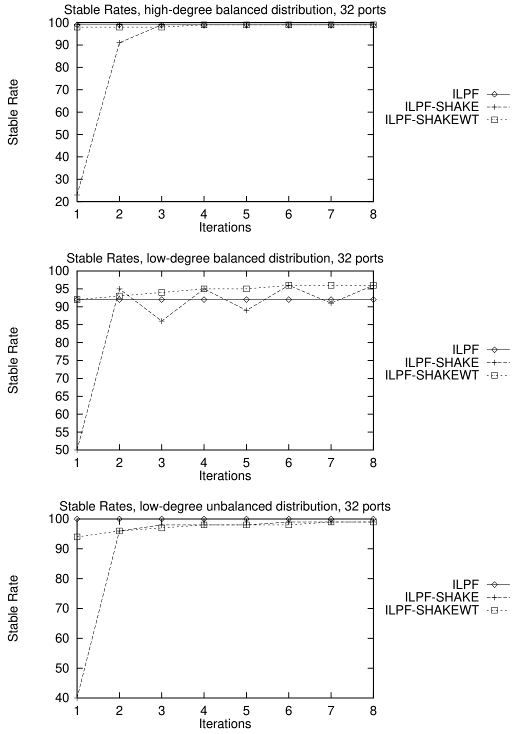

Figure 7: Stability levels for ILPF in regular, Shake, and weighted Shake varieties.

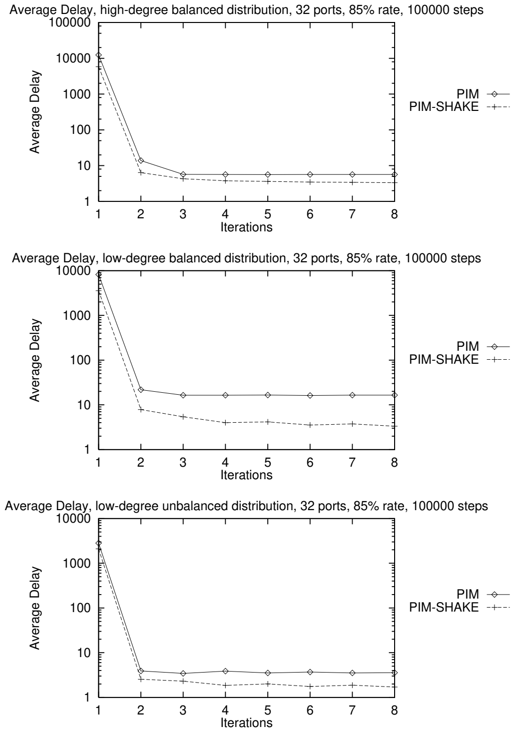

Figure 9: Average delay for PIM in regular and Shake varieties.

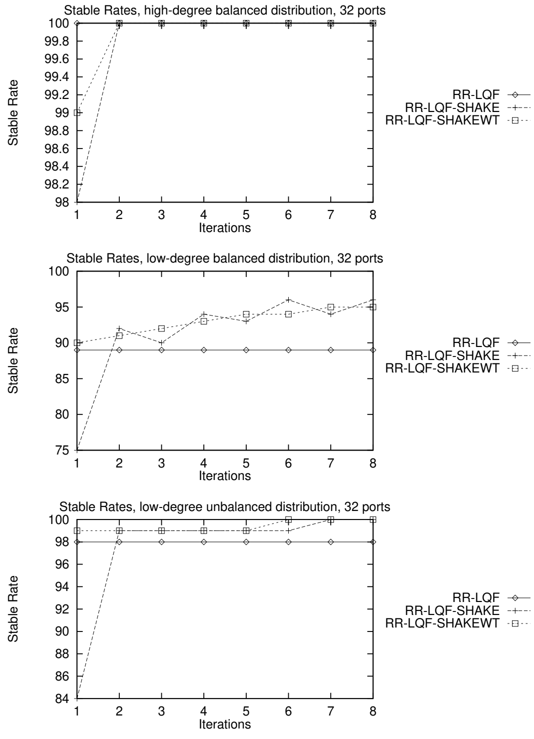

Figure 8: Stability levels for RR-LQF in regular, Shake, and weighted Shake varieties.

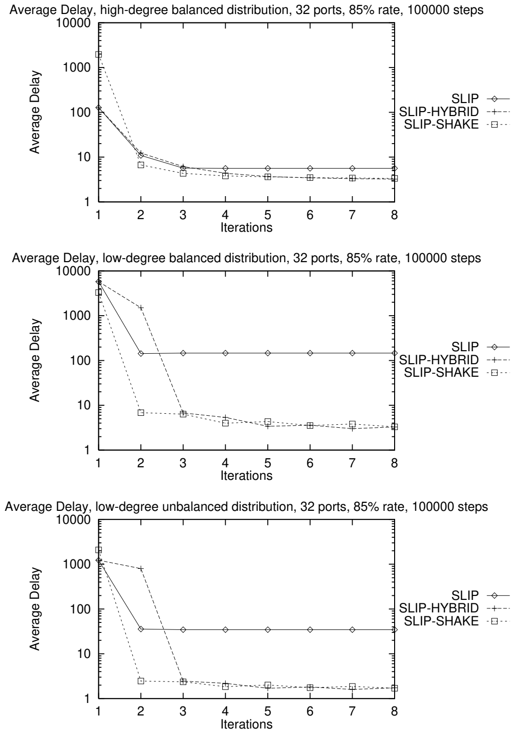

Figure 10: Average delay for SLIP in regular, Shake, and hybrid varieties.

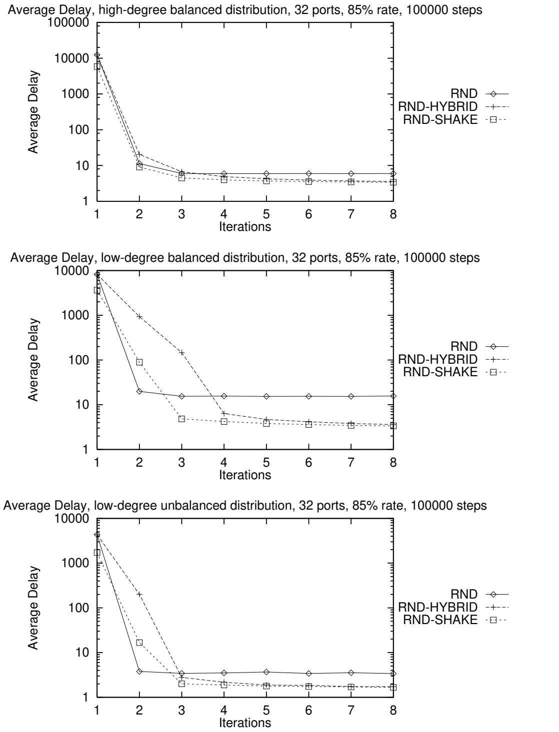

Figure 11: Average delay for RND in regular, Shake, and hybrid varieties.

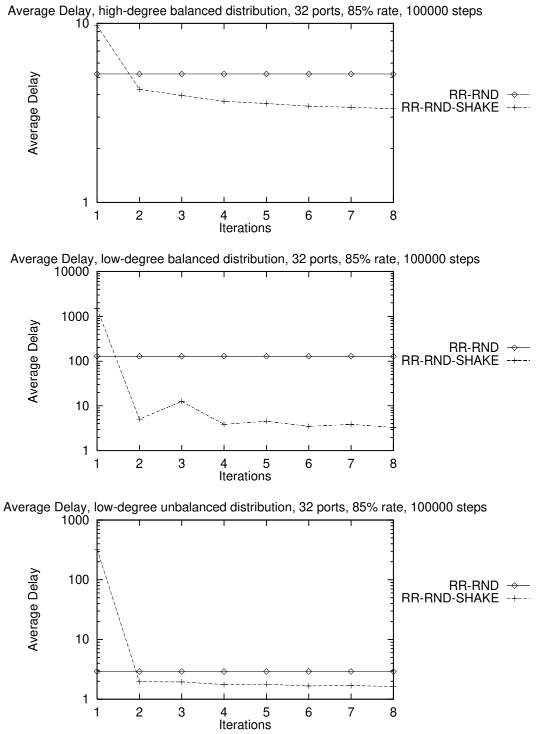

Figure 12: Average delay for RR-RND in regular and Shake varieties.

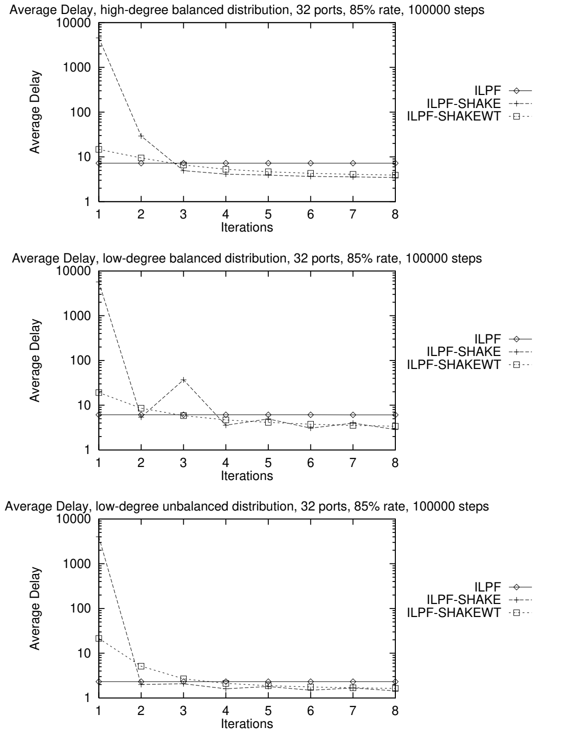

Figure 13: Average delay for ILPF in regular, Shake, and weighted Shake varieties.

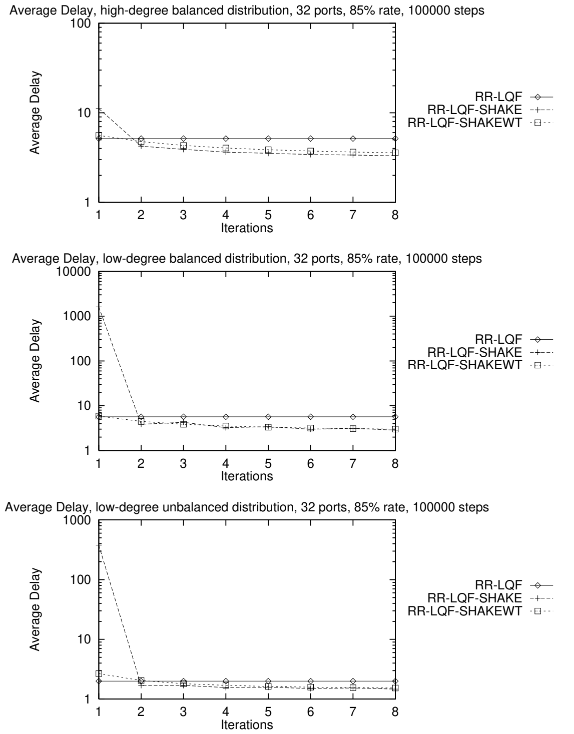

Figure 14: Average delay for RR-LQF regular, Shake, and weighted Shake varieties.

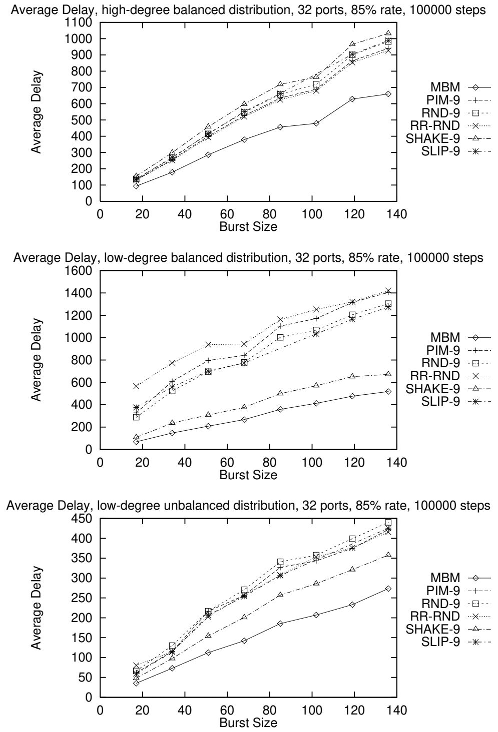

Figure 15: Average delay for unweighted heuristics with bursty traffic.

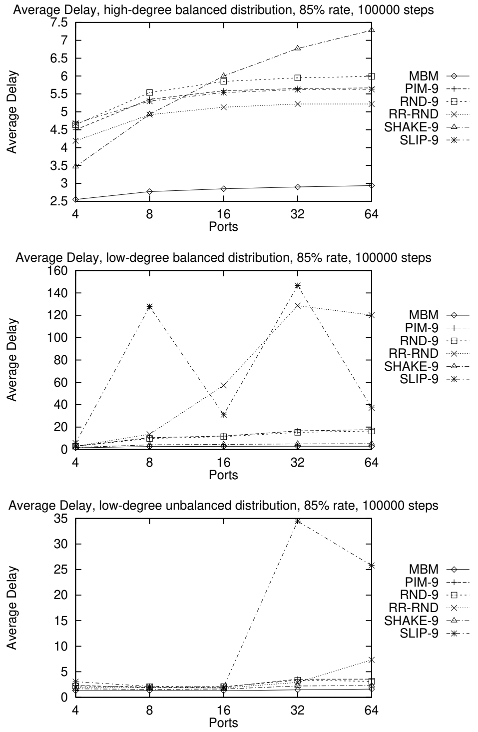

Figure 17: Average delay for unweighted heuristics with different switch sizes.

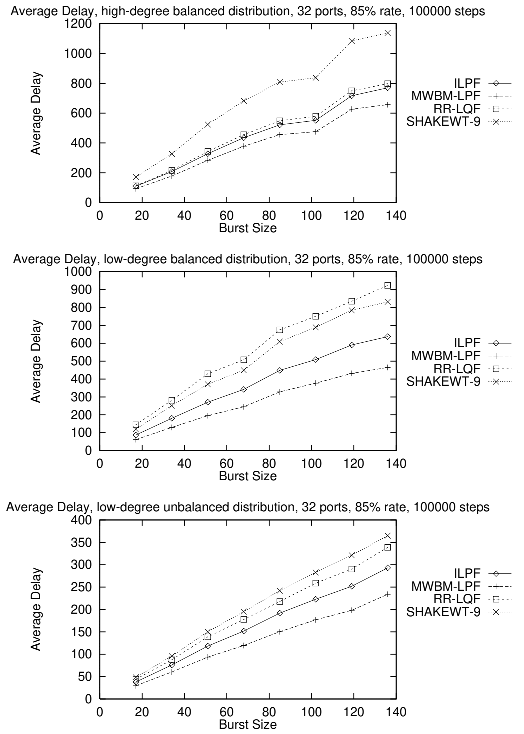

Figure 16: Average delay for weighted heuristics with bursty traffic.

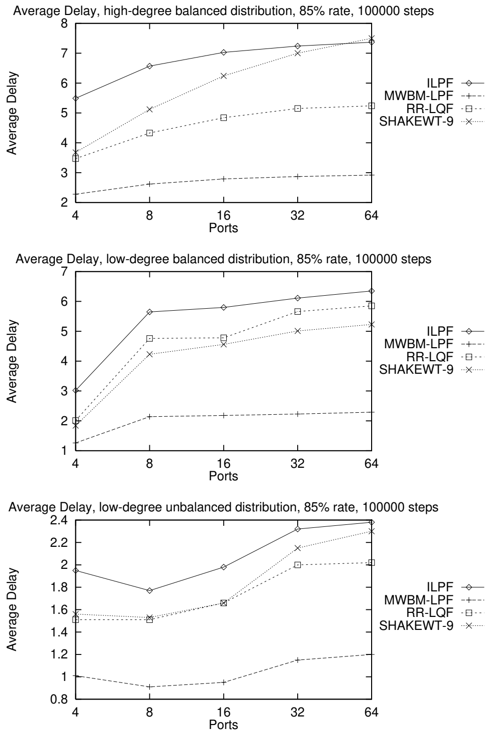

Figure 18: Average delay for weighted heuristics with different switch sizes.