Experimental Analysis of Scheduling Algorithms for Aggregated Links

Wojciech Jawor* Marek Chrobak* Mart Molle*

Abstract

We consider networks with aggregated links, that is networks in which physical link segments between two interconnected devices are grouped into a single logical link. Network devices supporting such aggregated links must implement a distribution algorithm responsible for choosing the transmission link for any given packet. Traditionally, in order to maintain packet ordering within conversations (which is crucial for maintaining high performance of transmission protocols), all distribution algorithms transmitted packets from a given conversation on a single link. This approach is attractive for its simplicity, but in some conditions it tends to under-utilize the capacity of the link. Sending packets from the same conversation simultaneously along different physical links is likely to improve performance, but, with packets of different lengths, one needs to schedule the transmissions carefully to ensure that the packets arrive in the correct order.

Recently, Jawor *et al.* [6] formulated this packet scheduling problem as an online optimization problem and proposed an algorithm called Block. The focus of the work in [6] was purely theoretical – to achieve an optimal competitive ratio. In this paper we study this problem in an experimental setting. To this end, we develop a refined version of Algorithm Block (retaining its competitive ratio), and we show, through experimental analysis, that it indeed significantly reduces the maximum amount of time packets spent in buffers before transmission, compared to the standard methods based on hashing.

1 Introduction

Link aggregation is a method of grouping two or more physical links between two network devices in such a way that a client can treat the group as a single virtual link. A collaborative use of multiple links, apart from increased bandwidth, improves resiliency to failures. In an event of a link failure, traffic may be redistributed from the broken link to the remaining links in the group. This way the connection and data flow between the interconnected devices is maintained, and the loss only reduces available capacity. Another benefit is that traffic load may be balanced across different links. These advantages, combined with low costs of the technology, have made it very popular: In a survey [3] of 38 major ISPs, only two (smaller) ISPs did not have parallel links between nodes.

However, the technology also introduces new challenges. Since packets transmitted between two interconnected devices may be serviced concurrently by several physical links (possibly of varying bandwidth), the order in which the data packets arrive at the receiver (ie., the order in which the last bits of packets arrive) may be different from the order in which they originally arrived at the sender. The reordering is an important issue for systems with aggregated links, as it may

*Department of Computer Science, University of California, Riverside, CA 92521. Supported by NSF grants OISE-0340752 and CCR-0208856. {wojtek,marek,mart}@cs.ucr.edu

have direct bearing on the performance of the transmission control protocols, which routinely treat certain instances of out of order packet delivery as a congestion signal [4]. This influences the performance of the whole system. In fact, Bennett *et al.* [2] argue that the parallelism in Internet components and links is one of the *main* reasons of packet reordering. This contradicts a common belief that reordering is caused by "pathological" behavior (ie., by incorrect or malfunctioning network components).

Network traffic. The traffic entering the network is often divided into disjoint *conversations*, where a conversation is the traffic between a distinguishable source-destination pair. (In literature the term *flow* is often used; unfortunately it conflicts with the scheduling terminology.) In the network, packets are sent from one node to the next. Every output of a node is equipped with a scheduler which receives packets from conversations that traverse the node and for each packet chooses its transmission time and an output link.

We assume that all physical links have the same speed. Even in this scenario, however, packet reordering will occur, due to the variance of lengths between packets and the fact that the ordering of the packets at the destination is determined by the arrival times of their last bit. For example, suppose a conversation contains two packets of lengths p_1 and p_2, that arrive at the sender in this order. If $p_1 < p_2$, and if we send both packets simultaneously along different physical links, the second packet will arrive at the destination $p_2 - p_1$ time units (bits) earlier than the first one. Thus the sender needs to wait with sending the second packet for at least $p_2 - p_1$ time units. Note that this does not necessarily mean that the one link is not utilized during this time, for it may be used to transmit packets from other conversations.

Online algorithms. In many practical scenarios algorithms do not have the access to the entire input instance at the beginning of computation. Instead, the input is revealed over time, and the algorithm must react to the new information without the knowledge of the future. Such algorithms are called *online*, as opposed to *offline* algorithms, which have a complete knowledge of the input instance at the beginning of computation.

Online algorithms are evaluated using *competitive analysis* introduced by Sleator and Tarjan [7]. For an online algorithm A, let $\mathsf{A}(I)$ denote the schedule produced by A on an instance I. For a minimization problem, an algorithm A is defined to be *r-competitive* if

$$|\mathsf{A}(I)| \leq r \cdot |\mathsf{OPT}(I)|,$$

where $\mathsf{OPT}(I)$ is an optimum (offline) schedule of I, and $|\cdot|$ denotes the value of the objective function of a schedule. The smallest such value r is called the *competitive ratio* of A.

Past work. The IEEE 802.3ad standard [5] defines an implementation of aggregated links between devices in Local Area Networks. The standard does not specify the packet scheduling algorithm, but it does assume that the algorithm neither reorders, nor duplicates the packets. In practice, the requirement of maintaining packet ordering is met by ensuring that all packets that compose a given conversation are transmitted on a single physical link. The assignment of conversations to links is achieved by using a hash function. This approach has several drawbacks: First, it does not fully utilize the capacity of a link if the number of conversations is smaller than the number of links. Second, such an algorithm does not provide load balancing, i.e., if traffic increases beyond a single channel's bandwidth, it is not distributed among additional links. (For illustration,

suppose we have two physical links and three conversation. The in the hashing scheme, one link's load will be double that of the other.) Third, it is hard or even not possible to design a hash function that would distribute the traffic well in all situations. And finally, in some (common) configurations the link aggregation algorithm (which is a part of the Link Layer) must violate the layered architecture of network protocols by accessing higher layer information in order to compute useful hash functions.

A different approach to the problem has recently been proposed by Jawor *et al.* [6]. They formulate the problem as an online multiprocessor job scheduling problem with the objective to minimize maximum flow time. The processors represent physical links and jobs represent packets. To model the packet ordering preservation property, the jobs are required to complete in the first-released-first-completed order (FRFC, in short).

Algorithm Block developed in [6] (which we describe in detail later on) produces schedules in which packets composing a single conversation may be sent on several different links in the aggregation group. The algorithm makes sure that the packets arrive at the receiver in the correct order by taking packet lengths into account to determine their transmission times. In fact, the algorithm does not distinguish between different conversations, but treats the whole input sequence as one large conversation in which the packet ordering is induced by ordering of packets in the original conversations. The main result in [6] is that Block is $O(\sqrt{n/m})$-competitive for the maximum flow time objective function, where n is the number of packets and m the number of links in the aggregation group, They also show that this bound is asymptotically optimal.

Our results. The focus of the work in [6] was purely theoretical, namely to achieve an optimal (asymptotic) competitive ratio. In contrast, in this paper we study this problem in an experimental setting. Although we retain the fundamental idea of Block (and preserve its competitive ratio), in order to achieve satisfactory experimental performance, we enhance Algorithm Block with several heuristics. Roughly, Block works by scheduling incoming packets in disjoint blocks, often delaying packet transmissions to wait for the start of the next block, even though there may be non-utilized links. This does not affect the asymptotic competitive ratio, but is likely to adversely affect performance in practice. Our heuristics are designed to join these block more efficiently, to optimize link utilization while preserving correct packet ordering.

We present a series of experiments on real network traces in which we compare the two algorithms and an algorithm based on the hashing function. Our experiments show that our algorithm significantly reduces the maximum amount of time packets spent in buffers before transmission, compared to the standard methods based on hashing. We also discuss implementation issues; in particular, we propose a hardware implementation of our method.

2 FRFC Scheduling

To model the situation described in the introduction as a scheduling problem, we associate a job with each network packet, and a machine with each output link. The goal is to optimize link utilization, under the constraint that packets complete their arrivals at the receiver in the order of their arrivals at the sender. Using scheduling terminology, we state this problem as follows: We are given n jobs (packets) organized in disjoint chains (conversations), with each job j specified by a pair (r_j, p_j) where r_j is a positive release (arrival) time, and p_j is the processing time, or length of the job. We assume that $\min_j r_j = 0$. In addition, the jobs are ordered so that if a job j precedes

j' (we simply write $j < j'$) then $r_j \leq r_{j'}$. The ordering of job indices within a chain represents the ordering of packets in a conversation. We assume that at the device where scheduling takes places the packets arrived in a correct order, which justifies the ordering of the release times.

A schedule $\mathcal{A}$ is a function, which for each job j specifies the job's starting time $S_j^{\mathcal{A}}$ and a machine executing j. The completion time of job j is $C_j^{\mathcal{A}} = S_j^{\mathcal{A}} + p_j$. The objective is to construct a schedule $\mathcal{A}$ which satisfies the following constraints: (1) $S_j^{\mathcal{A}} \geq r_j$, (2) $C_j^{\mathcal{A}} - S_j^{\mathcal{A}} = p_j$, (3) if jobs i and j are scheduled on the same machine then $C_i^{\mathcal{A}} \leq S_j^{\mathcal{A}}$ or $S_i^{\mathcal{A}} \geq C_j^{\mathcal{A}}$, and (4) for any two jobs $j < j'$ from the same chain, we have $C_j^{\mathcal{A}} \leq C_{j'}^{\mathcal{A}}$. Whenever the condition (4) is satisfied we say that jobs are scheduled in FRFC order.

In addition to the above constraints we also want to optimize the machine utilization. Let $F_j^{\mathcal{A}} = C_j^{\mathcal{A}} - r_j$ denote the *flow time* of a job j in schedule $\mathcal{A}$. We aim at constructing schedules which minimize the *maximum flow time* $F_{\max}^{\mathcal{A}} = \max_j F_j^{\mathcal{A}}$ of $\mathcal{A}$. The use of this function is motivated by *Quality-of-Service* applications in which, in order to provide the delay guarantees, an upper bound on the time *each* job spends in the system must be given. This helps avoid undesirable starvation issues. In the process, we will also construct schedules $\mathcal{A}$ to minimize maximum completion time, or *makespan*, $C_{\max}^{\mathcal{A}} = \max_j C_j^{\mathcal{A}}$. By $F_{\max}^*(I)$ we denote the (offline) optimum flow time of I, and by $C_{\max}^*(I)$ its optimal makespan. (Note that these optimum may be realized by different schedules.)

3 Algorithm Block

Algorithm Block [6] schedules jobs in blocks by repeatedly using an offline scheduling procedure as a black-box. We describe this scheduling procedure first.

This next algorithm is designed to minimize the makespan of the output schedule. To simplify the description of the algorithm we will assume that $r_j = 0$ for all jobs j, i.e., all jobs are available at time 0. Note that in this special case minimizing maximum completion time of all jobs is equivalent to minimizing the maximum flow time of the schedule. Also, recall that, although all jobs have equal release times, the order in which they are required to complete in the final schedule is fixed.

The algorithm is shown in Figure 1. It receives on input a set of jobs indexed $1, 2, \ldots, n$, and outputs an FRFC schedule of these jobs. The schedule is constructed in two steps: First an auxiliary schedule $\mathcal{X}$ is computed, and then it is shifted to the right to meet the release time.

Theorem 3.1 (Jawor *et al.* [6]) *Algorithm* COpt *computes a schedule that minimizes* $\max_j C_j^{\mathcal{A}}$.

As we mentioned before, COpt is an essential component of an online algorithm Block. This latter algorithm is online and was designed to minimize the maximum flow time of a schedule.

Algorithm Block: The algorithm proceeds in phases numbered $1, 2, 3, \ldots$, where phase i starts at time β_i. First, let $\beta_1 = 0$. Consider phase i, and let Q_i be the set of jobs pending at time β_i. We apply algorithm COpt to schedule all jobs in Q_i and shift the resulting schedule forward by β_i. Suppose that the last job from Q_i completes at time $\beta_i + \delta_i$. Then $\beta_{i+1} \geq \beta_i + \delta_i$ is the first time when there is at least one pending job. (If no more jobs arrive, the computation completes.)

```
/* initialize */
for i ← 1 to m do T_i ← 0

/* create auxiliary schedule X */
for j ← n downto 1 do
    l ← argmax{T_i : 1 ≤ i ≤ m}
    schedule j on machine l at time S_j^X ← T_l − p_j
    T_l ← S_j^X

/* construct the final schedule A */
ξ ← min_j S_j^X
for j ← 1 to n do S_j^A ← S_j^X − ξ
```

Figure 1: Algorithm COpt

4 Algorithm JBlock

In this section we introduce Algorithm JBlock, which is an improved version of Block. The new algorithm maintains the competitive ratio of Block, but performs better in practice thanks to simple heuristics. In order to introduce the new algorithm we first make some general observations about Block.

There are two places were Block could be improved. First, the algorithm does not start a new block unless all jobs from previous block have been completed, even though some machines may be idle and new jobs may be pending. Second, during the process of building a new block the algorithm shifts jobs to the right, all by the same amount. This behavior may degrade the performance because shifting is applied under the assumption that all jobs in the block have equal release times, and thus, minimizing maximum flow time is equivalent to minimizing maximum completion time. As this assumption is not valid in practice, we conjecture that shifting some jobs to the right by smaller amount may reduce the maximum flow time of jobs in the block. We now describe two heuristics that are designed to address these two issues.

Joining blocks. We first aim at improving the procedure of joining blocks in the output schedule. More precisely we would like to solve the following problem: Given completion times $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_m$ for each machine and a set Q of pending jobs, compute an FRFC schedule $\mathcal{A}$ such that the following conditions are satisfied: (c1) If job j is scheduled on machine i, then $S_j^{\mathcal{A}} \geq \lambda_i$; (c2) $C_j^{\mathcal{A}} \geq \lambda_1$, for all $j \in Q$, and, (c3) $\mathcal{A}$ has minimum makespan.

Intuitively, the value λ_i represents the last completion time of jobs from the previous block scheduled on machine i. The condition (c2) must be satisfied in order to preserve the FRFC property in the final schedule. The property ($c3$) is required to preserve the competitive ratio.

Since in Block each block is constructed using Algorithm COpt we concentrate on modifying this algorithm first. Let the *load of a machine* be the sum of processing times of jobs scheduled on that machine. Throughout this section we assume that in the schedule returned by COpt the load of machine i is not greater than the load of machine i', for any two machines $i < i'$. (Otherwise jobs can be appropriately reassigned.)

```
/* Step 1. Initialize */
V ← schedule computed by COpt on Q
for i ← 1 to m do
      τ_i ← min{S_j^V : j scheduled on machine i}

/* Step 2. Shift jobs */
Δ ← max(λ_1 − C_1^V, max_i(λ_i − τ_i))
for j ← 1 to n do S_j^A ← S_j^V + Δ
```

Figure 2: Algorithm JCOpt

Consider Algorithm JCOpt, shown in Figure 2. We claim that this algorithm solves the problem stated above. The main idea is simple. We first compute the assignment of jobs to machines using the ordering of values λ_i and Algorithm COpt, and then adjust the starting times to satisfy conditions (c1) and (c2). In the assignment procedure we use the fact that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_m$ and that machine loads increase with machine indices.

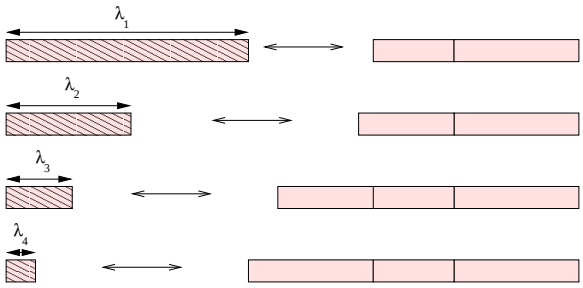

Figure 3: The idea of JCOpt

Theorem 4.1 *Algorithm* JCOpt *computes an FRFC schedule which satisfies conditions* $(c1)$, $(c2)$, *and* $(c3)$.

We defer the proof of the above theorem to the Appendix.

4.1 Local Reduction of Flow Times

In this section we describe the second heuristic, which improves the performance of Block. We wish to eliminate the uniform shifting of jobs at the end of JCOpt. The improvement is based on the fact that not all jobs in a given block have equal release times, and thus, minimizing maximum flow time is not equivalent to minimizing maximum completion time.

We now modify Step 2 of JCOpt and prove that the schedule produced by the modified algorithm also satisfies conditions (c1), (c2) and (c3). In addition we introduce several other modifications that will be useful during the implementation of the algorithm. The modified version of JCOpt, which we call JCOpt2, is shown in Figure 4. Let us briefly explain the modifications.

In lines 1-6 we compute the auxiliary schedule exactly in the same way it was computed in COpt. In line 7 we reassign jobs to satisfy the requirement that for any two machines $i < i'$ the

1: /* *Step 1: Create auxiliary schedule* $\mathcal{X}$ */
2: `for` $i \leftarrow 1$ `to` m `do` $T_i \leftarrow 0$

3: `for` $j \leftarrow n$ `downto` 1 `do`
4: $l \leftarrow \text{argmax}\{T_i : 1 \leq i \leq m\}$
5: schedule j on machine l at time $S_j^{\mathcal{X}} \leftarrow T_l - p_j$
6: $T_l \leftarrow S_j^{\mathcal{X}}$

7: reassign jobs in $\mathcal{X}$ to obtain $|T_i| \geq |T_{i'}|$ for $i \geq i'$

8: /* *Step 2: Shift jobs* */
9: $\xi \leftarrow \min_i T_i$
10: `for` $i \leftarrow 1$ `to` m `do` $\alpha_i \leftarrow \lambda_i$

11: $\kappa \leftarrow \lambda_1$
12: `for` $j \leftarrow 1$ `to` n `do`
13: $l \leftarrow$ index of machine which executes j
14: $S_j^{\mathcal{A}} \leftarrow \max(S_j^{\mathcal{X}} - \xi, \kappa - p_j, \alpha_l)$
15: $\kappa \leftarrow \alpha_l \leftarrow S_j^{\mathcal{A}} + p_j$

Figure 4: JCOpt2, a modified version of JCOpt.

load of i is not greater than the load of i'. This reassignment can be easily obtained by sorting values $|T_1|, |T_2|, \ldots, |T_m|$, since $|T_i|$ is equal to the load of machine i. (In the actual implementation at this point we only need to compute a mapping $\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, m\}$ which specifies the ordering of machine loads. We do not need to physically modify the assignments of jobs to machines yet, as this can be done in the loop starting on line 12.)

In line 9, we compute the amount by which all jobs must be shifted in order to guarantee the feasibility of the schedule. Originally we needed to compute $\min S_j^{\mathcal{X}}$ (i.e., a minimum over n elements). However, here we use the fact that all jobs are available at time 0, and the fact that T_i is equal to the minimum starting time of all jobs on machine i, which reduces the computation to a minimum over m elements. In lines 12-15 we shift the jobs. This loop contains the computations previously done in two separate loops.

The following theorem states that the above modifications do not increase the maximum completion time of the jobs in the schedule. We defer the proof to the Appendix.

Theorem 4.2 *Let $\mathcal{A}$ be the schedule produced by* JCOpt *on an instance Q and let $\mathcal{A}'$ be a schedule produced by* JCOpt2*. Then $\mathcal{A}'$ satisfies conditions* $(c1)$ *and* $(c2)$*, and $C_{\max}^{\mathcal{A}} = C_{\max}^{\mathcal{A}'}$.*

We have completed describing the two heuristics. We now present the new online algorithm.

Algorithm JBlock**:** The algorithm proceeds in phases numbered $1, 2, 3, \ldots$, where phase i starts at time β_i. First, let $\beta_1 = 0$ and let $\lambda_1 = \lambda_2 = \cdots = \lambda_m = 0$. Consider phase i, and let Q_i be the set of jobs pending at time β_i. Let λ_i to be the last completion time on machine i or the current time, whichever is larger. We apply algorithm JCOpt2 to schedule all jobs in Q_i with λ_i as defined in the previous sentence. Let $\beta_i + \delta_i$ be the first time when one of the machines completes all jobs

assigned to it. Then $\beta_{i+1} \geq \beta_i + \delta_i$ is the first time when there is at least one pending job. (If no more jobs arrive, the computation completes.)

Example. Let us illustrate Algorithm JBlock on an example. Figure 5 shows a 3-machine schedule of jobs: $1 = (0, 1, 1), 2 = (0, 1/2, 1), 3 = (1/4, 2, 1), 4 = (1/4, 1, 1), 5 = (1/4, 1/2, 1), 6 = (1/2, 1, 1), 7 = (4, 1, 1)$ (note that we use the same example as for Block). The first and third blocks are shown in light gray; the second and fourth blocks are shown in dark gray.

Initially $\lambda_1 = \lambda_2 = \cdots = \lambda_m = 0$, therefore the first block is identical to the first block constructed by Block on the same instance. The next time, when there are available jobs and a machine is idle is time 1/4. The pending jobs are $3, 4, 5$, also $\lambda_1 = \lambda_2 = 1$, $\lambda_3 = 1/4$. We build the second block using algorithm JCOpt2. It first assigns jobs $3, 4, 5$, each to a different machine. Since machine executing 3 is most loaded (taking into account the block alone), job 3 is assigned to machine 3. Machines 1 and 2 execute the remaining jobs. Finally the jobs are shifted one by one. First job 3 to time 1/4, then job 4 to time 5/4, and job 5 to time 7/4. Blocks containing 6 and 7 are constructed similarly.

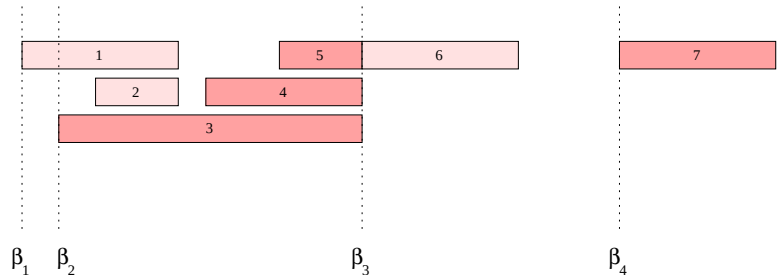

Figure 5: Illustration of algorithm JBlock

4.2 Implementation of JBlock

We now propose a hardware implementation of Algorithm JBlock. In our implementation frames processed in two consecutive phases are stored in two separete buffers, called the primary and secondary buffer (depending on the role they currently play). The actual computation is performed on frames stored in the primary buffer, whereas the incoming frames (i.e., frames to be processed in the next phase) are stored in the secondary buffer. Once the computation proceeds to the next phase, the secondary and primary buffers are switched.

Figure 6 presents the components used during the computation (the secondary buffer is not shown). The primary buffer (as well as the secondary buffer) is a stack (a FILO queue). When JCOpt creates the auxiliary schedule (lines 1-7) it processes frames from the last one to the first one, i.e., exactly in the same order in which the frames leave the stack. For each frame i the algorithm computes the starting time (in the auxilary schedule) denoted S_i and a machine M_i which executes it. During this computation the algorithm uses Array 1 to hold variables $T_1, T_2, \ldots, T_m$.

After all frames are processed and stored in Stack 2, the algorithm uses Array 1 and 2 to compute the mapping σ (line 8) which is stored in Array 3 and used when frames are assigned to appropriate output buffers. Observe that during this computation (the loop in lines 14-17) frames must be processed in the order of their arrivals, i.e., exactly in the same order in which they leave Stack 2. This part of the computation also uses variables $\alpha_1, \alpha_2, \ldots, \alpha_m$, which are initially set to

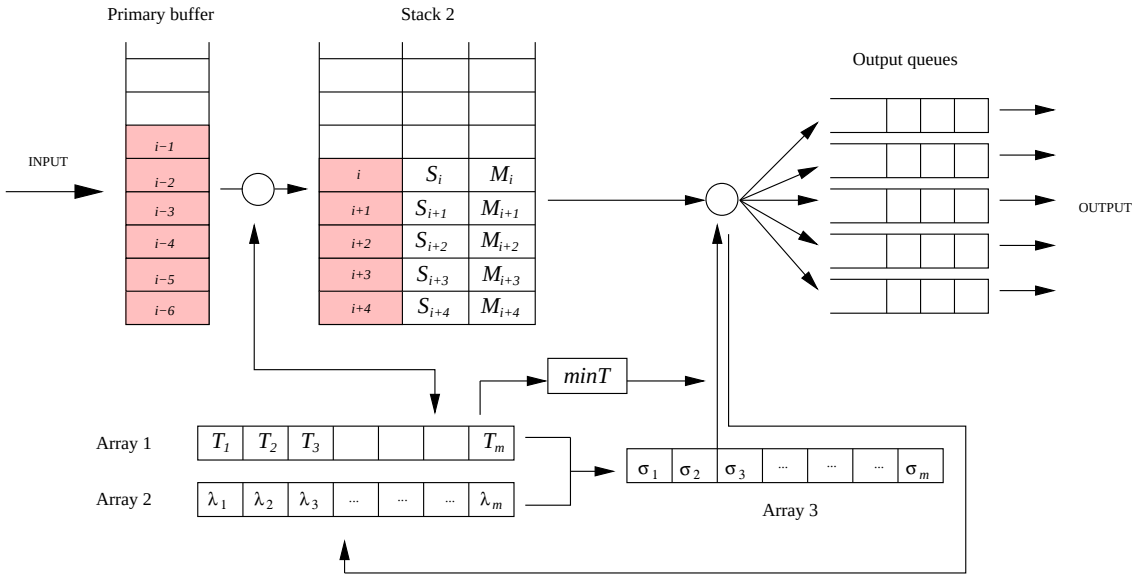

Figure 6: Implementation of JBlock

$\lambda_1, \lambda_2, \ldots, \lambda_m$, respectively. Since Array 2 already contains these values, it may be used to hold variables $\alpha_1, \alpha_2, \ldots, \alpha_m$ during this part of computation. This is beneficial, since, by the definition of the algorithm, at every iteration of the loop starting on line 14, variable α_i is equal to the last completion time of jobs scheduled on machine i. Therefore, when the next phase is executed, say at time t, each variable λ_i can simply be set to $\max(t, \alpha_i)$.

5 Experiments

We now describe our experiments and discuss the results. We implemented four algorithms: Hash (using a hashing function), Block and JBlock, and an offline algorithm, FOpt, to compute the optimal schedule. Next, we tested the algorithms on real network traces.

Data and configuration. The simulations use trace data from the MAWI traffic archive [1] collected over the period January – March, 2006. The line is a 100-Mbps trans-Pacific link with 18-Mbps CAR (committed access rate). All traces were measured during hours 2:00pm–2:15pm.

Each trace consists of one entry per packet, which includes the packet length (in bytes) and a time stamp for moment that the *end of the packet* left the router port. Therefore, the length of packet n divided by link speed (100 Mbps in this case) gives us its service time, req_n, and the difference between time stamps n and $n-1$ is int_n, the time available for transmitting packet n. Since the trace is collected *after* the packets have been serialized for transmission over the link, there should never be any queueing if we "replay" the trace feeding a link with the same capacity. Therefore, as a first step, we drop all packets from the trace for which $req_n > 1.2int_n$, because they must represent measurement errors. In the second step, we superimpose a total of 12 separate traces from the archive to create an synthetic trace file with more traffic and unsynchronized arrivals between different traces. The resulting synthetic trace is a good approximation to the output queue at a router port that forwards traffic arriving on 12 different input ports.

A set of experiments is repeated three times, each time on a different set of traces: first on a superposition of traces collected every Wednesday, 1/11 -3/29; second on a superposition of traces collected every Thursday 1/12-3/30; third on a superposition of traces collected every Friday 1/13-3/31. The total number of packets was 97,319,767, 98,423,380, and 96,781,700, respectively for the three sets of traces. Overall we used 36 traces.

In each set of the experiments we vary the number of links in the LAG and their bandwidth. We perform two experiments on LAGs containing 4 links (in the first case the bandwidth of an individual link is equal 100Mbps and in the second 200Mbps), and two experiments on LAGs containing 8 links (in the first case the bandwidth of an individual link is equal 50Mbps and in the second 100Mbps). The bandwidths and the number of links were chosen so that there are two pairs of experiments in which the total bandwidth of the LAG is the same (namely 400Mbps and 800Mbps). This allows us to compare the impact of the degree of parallelism of the LAG among experiments while keeping the total bandwidth constant.

We identified conversations based on the source and destination IP addresses and port numbers. Each conversation received a unique identification number, and was identified by this number throughout the computation. The identification is done before the traces are superimposed, which means that in the final experiment we distinguish between conversations that originated in two different traces, even if their source and destination address and port numbers match.

Algorithms. We implemented the following four algorithms: Algorithm Hash, that distributes the packets on links using a hashing function, Algorithm Block, Algorithm JBlock, and Algorithm FOpt which computes the optimal offline solution.

Among the above algorithms only Hash exploits the partitioning of the input trace into individual conversations. That algorithm determines the destination link in the LAG based on conversation parameters (source and destination addresses and/or port numbers). The hashing functions used in practice assume that the conversation parameters are distributed in the parameter space uniformly at random and, therefore, the outgoing link will also be chosen uniformly at random if the hashing function is designed properly. This assumption may be invalid as it depends on the topology of the network. However, since our goal was not to determine the efficiency of a particular hashing function, but the efficiency of the overall approach, we do not choose the outgoing link based on the conversation parameters. Instead, for each conversation we choose it uniformly at random.

In case of algorithms Block and JBlock it is possible that two different packets (transmitted on parallel links) arrive at the destination at the very same time. Since we have no guarantees on how the hardware at the destination treats such packets we have modified both algorithms to include idle times between completion time of such packets. We have chosen the length of this idle time to be the time required to transmit 1 byte on a single link.

Methodology. In each experiment we compare the maximum flow times of packets scheduled using Hash, Block, JBlock, and the offline optimum. Since computing the offline optimum of an input instance with more than one conversation is $\mathcal{NP}$-hard [6], we, once again, treat all packets as one large conversation with the ordering of packets induced by original conversations. (For this reason it is possible that the algorithm using a hashing function *outperforms* the offline optimum.)

We measure the maximum flow time in the intervals of 1 sec. for each of the algorithms, i.e., at the beginning of each 1 sec. interval we report the maximum flow time of packets that have been transmitted in the previous interval. In case of OPT packets that arrive in each interval are treated

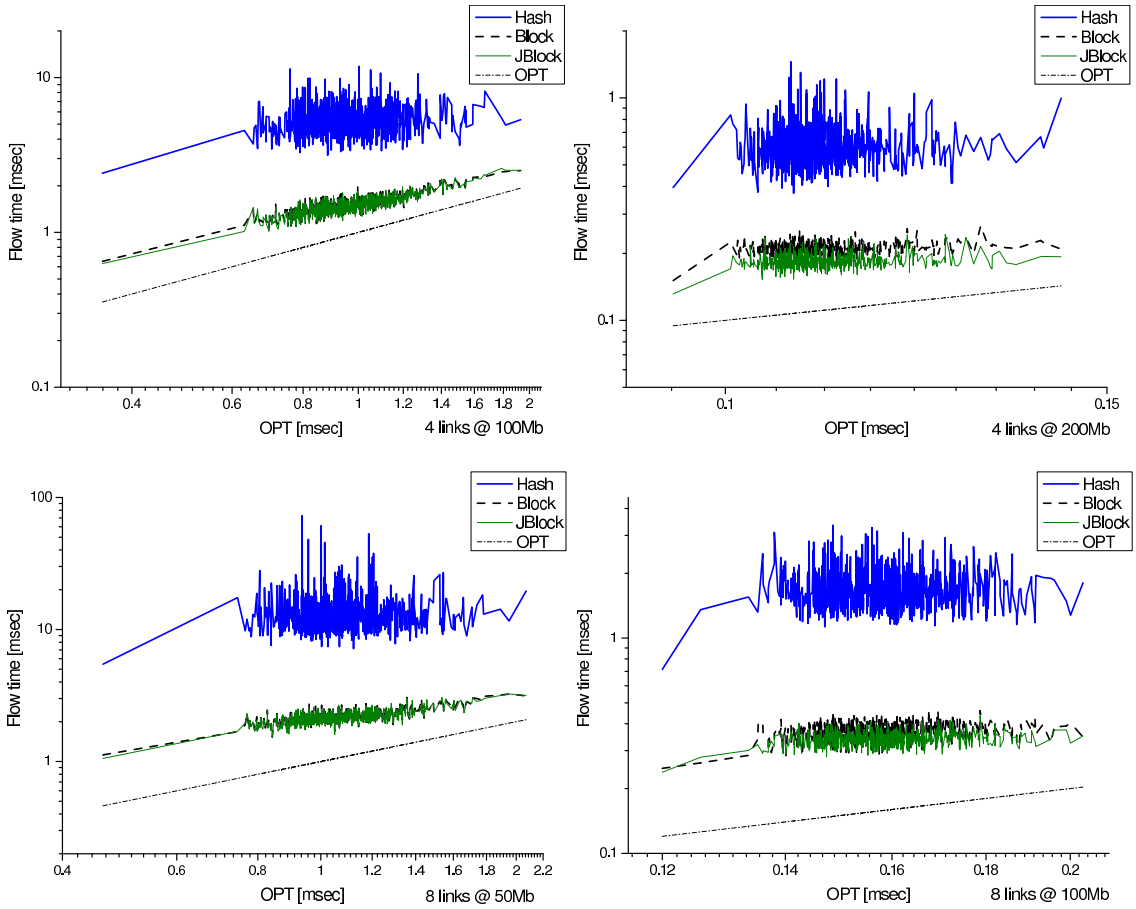

Figure 7: Relative performance of the scheduling algorithms to OPT, on Wednesday traces.

as a separate input instance. We decided to examine the performance of the algorithms on short intervals, instead of the whole 15 minute interval, because measuring the maximum flow time over the entire instance would provide little information on the local behavior of the algorithms.

Results and discussion. The results of the experiments are presented in Figure 7. The graph was obtained from the raw data by sorting the time quanta according to the increasing value of the offline optimum achieved in these quanta, using these values as the x-coordinate, and plotting the results of the other algorithms on the y-coordinate. (Both axes are in logarithmic scale.) Additional graphs (Figures 8 and 9) can be found in the Appendix, including one graph with raw data.

The experiments show that even though we imposed an additional restriction on the schedules constructed by Block and JBlock , both algorithms outperform the hashing algorithm, regardless of the number of links in the aggregation group and their bandwidth. Recall that the additional restriction is based on the fact that Block and JBlock (as well as the offline optimum) maintain the ordering of packets globally, while the hashing algorithm maintains packet ordering only within conversations. The consequence of this fact is following: the packet distribution algorithm does not need to violate the layered architecture of network protocols (in order to obtain the information on the assignment of packets to conversations) to dramatically increase its efficiency.

The difference between the traditional approach (in which a given conversation may only be sent on one output link) and the new approach (in which a single conversation may be distributed among several output links) is biggest in the case when the links composing the LAG are relatively slow (see the configurations with links operating at the rate of 50Mbps). How could these differences be explained? Algorithm Hash performs poorly, when the number of conversations is smaller than the number of links, because some links must remain idle. In our trace data the number of conversations is large (around 5mln in each experiment); however, a large number of conversations is only a necessary condition for Hash to perform well. It is not sufficient, since the algorithm also requires the traffic load to be distributed among these conversations uniformly. Apparently, this is not the case. Since algorithms Block and JBlock work with the total traffic load and not individual conversations, they handle this problem well.

Here are a few more observations that can be made based on the graphs in Figures 7, 8, and 9: (a) The hashing algorithm seems insensitive to the particular traffic patterns in different "quanta", because the main body of each clump is horizontal (except for outliers at the top and bottom), (b) Block and JBlock show better correlation to traffic patterns (although generally less than the optimum algorithm), (c) the difference between Block and JBlock is surprisingly small, except in the case of 4 links at 200 Mbps, where JBlock is visibly superior. The improvements of JBlock in the case of 4 links at 200 Mbps appear to be due to the fact that in this configuration both algorithms are likely to spend most time at block boundaries, where JBlock is superior to Block. (It is well known in queueing theory that one faster server with the combined capacity of two slow servers will finish its workload faster, because it can devote its total effort to one customer if necessary.) The 4 links at 200 Mbpbs configuration will have shorter busy periods and more frequent transitions from one block schedule to the next compared to configuration of 8 links at 100 Mbps, so improving the utilization of the block transitions will be more important.

References

[1] MAWI Working Group Traffic Archive `http://tracer.csl.sony.co.jp/mawi/`.

[2] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on networking*, 7(6):789–798, 1999.

[3] R. Gareiss. Is the internet in trouble? *Data Communications Magazine*, Sept. 1997.

[4] L. Gharai, C. Perkins, and T. Lehman. Packet reordering, high speed networks and transport protocol performance. In *Proceedings of the International Conference On Computer Communications and Networks*, pages 73–78, 2004.

[5] C. S. IEEE. Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. In *IEEE Std 802.3.. Standard for Information technology. Telecommunications and information exchange between systems. Local and metropolitan area networks. Specific requirements.* The IEEE, Inc., 2002.

[6] W. Jawor, M. Chrobak, and C. Dürr. Competitive analysis of scheduling algorithms for aggregated links. *Algorithmica*, 51:367–386, 2008.

[7] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28:202–208, 1985.

A Proofs

The proofs below use the following claim proven in [6].

Claim A.1 (Jawor *et al.* [6]) *Let $\mathcal{O}$ be any feasible schedule of jobs $1, 2, \ldots, n$. Then for any $j = 1, 2, \ldots, n$, we have $C^{\mathcal{O}}_{\max} - C^{\mathcal{O}}_j \geq C^{\mathcal{A}}_{\max} - C^{\mathcal{A}}_j$, where $\mathcal{A}$ is the schedule computed by* COpt.

A.1 Proof of Theorem 4.1

We first observe that $\mathcal{A}$ is an FRFC schedule. This follows from the fact that $\mathcal{V}$ is an FRFC schedule and $S^{\mathcal{A}}_j = S^{\mathcal{V}}_j + \Delta$ for all j.

Fix a job j. Let i' denote the machine on which j is executed in $\mathcal{V}$ and $\mathcal{A}$. By the definition of JCOpt we have $S^{\mathcal{A}}_j = S^{\mathcal{V}}_j + \Delta \geq S^{\mathcal{V}}_j + \max_i(\lambda_i - \tau_i) \geq S^{\mathcal{V}}_j + \lambda_{i'} - \tau_{i'} \geq \lambda_{i'}$ proving that (c1) holds. We also have $C^{\mathcal{A}}_j = C^{\mathcal{V}}_j + \Delta \geq C^{\mathcal{V}}_j + \lambda_1 - C^{\mathcal{V}}_1 \geq \lambda_1$. This proves that (c2) holds.

It remains to prove that $\mathcal{A}$ has minimum makespan among all FRFC schedules satisfying conditions (c1) and (c2). Let $\mathcal{O}$ be any schedule which satisfies these conditions. We prove that $C^{\mathcal{A}}_{\max} \leq C^{\mathcal{O}}_{\max}$.

Suppose that $\Delta = \lambda_1 - C^{\mathcal{V}}_1$. Then $C^{\mathcal{A}}_1 = \lambda_1$. By Claim A.1 we have $C^{\mathcal{A}}_1 \geq C^{\mathcal{O}}_1 + C^{\mathcal{A}}_{\max} - C^{\mathcal{O}}_{\max} \geq \lambda_1 + C^{\mathcal{A}}_{\max} - C^{\mathcal{O}}_{\max} = C^{\mathcal{A}}_1 + C^{\mathcal{A}}_{\max} - C^{\mathcal{O}}_{\max}$, where the last inequality follows from the fact that $\mathcal{O}$ satisfies (c2) and the equality from the case condition. This proves that $C^{\mathcal{A}}_{\max} \leq C^{\mathcal{O}}_{\max}$.

Now suppose that $\Delta = \max_i(\lambda_i - \tau_i)$. Choose k such that $\Delta = \lambda_k - \tau_k$. Let j_i denote the first job started on machine i in $\mathcal{A}$ and let $J = \{j_i : i \geq k\}$. By the definition of JCOpt and the choice of k we have $S^{\mathcal{A}}_{j_k} = \lambda_k$. Moreover $S^{\mathcal{A}}_{j_l} \leq S^{\mathcal{A}}_{j_k}$ for $l \geq k$.

We distinguish two cases. Suppose that all jobs in J are scheduled on different machines in $\mathcal{O}$. Then, at least one of these jobs must be scheduled on machine i such that $\lambda_i \geq \lambda_k$. Let j_x be such a job. We have

$$C^{\mathcal{O}}_{\max} \;\geq\; \lambda_i + C^{\mathcal{O}}_{\max} - S^{\mathcal{O}}_{j_x} \;\geq\; \lambda_k + C^{\mathcal{A}}_{\max} - S^{\mathcal{A}}_{j_x} \;\geq\; C^{\mathcal{A}}_{\max},$$

since $C^{\mathcal{O}}_{\max} - S^{\mathcal{O}}_{j_x} \geq C^{\mathcal{A}}_{\max} - S^{\mathcal{A}}_{j_x}$ by Claim A.1.

Now suppose that there are two jobs $j_x, j_y \in J$ such that $j_x < j_y$ and both are scheduled on the same machine in $\mathcal{O}$. Since $\mathcal{O}$ satisfies (c2) we have $S^{\mathcal{O}}_{j_y} \geq C^{\mathcal{O}}_{j_x} \geq \lambda_1$. So

$$C^{\mathcal{O}}_{\max} \;\geq\; \lambda_1 + C^{\mathcal{O}}_{\max} - S^{\mathcal{O}}_{j_y} \;\geq\; \lambda_k + C^{\mathcal{A}}_{\max} - S^{\mathcal{A}}_{j_y} \;\geq\; \lambda_k + C^{\mathcal{A}}_{\max} - S^{\mathcal{A}}_{j_k} \;=\; C^{\mathcal{A}}_{\max},$$

since $C^{\mathcal{O}}_{\max} - S^{\mathcal{O}}_{j_y} \geq C^{\mathcal{A}}_{\max} - S^{\mathcal{A}}_{j_y}$ by Claim A.1. The last inequality holds since, as observed earlier, $S^{\mathcal{A}}_{j_y} \leq S^{\mathcal{A}}_{j_k}$ for $y \geq k$.

This ends the proof.

A.2 Proof of Theorem 4.2

Fix a job j. Let i' denote the index of the machine executing j in $\mathcal{A}'$. Observe that $\alpha_i \geq \lambda_i$ for all $i = 1, 2, \ldots, m$, during the execution of the loop in lines 14-17, therefore $S^{\mathcal{A}'}_j \geq \alpha_{i'} \geq \lambda_{i'}$, and condition (c1) is satisfied. Similarly, $C^{\mathcal{A}'}_j \geq \kappa \geq \lambda_1$ for all j, hence condition (c2) is satisfied as well. Moreover, since at the beginning of each iteration for which $j > 1$, we have $\kappa = C^{\mathcal{A}'}_{j-1}$, $\mathcal{A}'$ is an FRFC schedule.

It remains to prove that $C^{\mathcal{A}}_{\max} \geq C^{\mathcal{A}'}_{\max}$. To prove this equality we show that

$$\Delta \geq S^{\mathcal{A}'}_j - (S^{\mathcal{X}}_j + \xi) \tag{1}$$

by induction on j. This suffices since $S^{\mathcal{A}}_j = S^{\mathcal{X}}_j + \xi + \Delta$.

Consider $j = 1$. Distinguish the following cases. If $S^{\mathcal{A}'}_1 = S^{\mathcal{X}}_1 + \xi$ then (1) holds since $\Delta \geq 0$. If $S^{\mathcal{A}'}_1 = \kappa - p_1 = \lambda_1 - p_1$ then $S^{\mathcal{A}'}_1 - S^{\mathcal{Y}}_1 \leq \lambda_1 - C^{\mathcal{Y}}_1 \leq \Delta$. Finally, if $S^{\mathcal{A}'}_1 = \lambda_{i'}$ then $S^{\mathcal{A}'}_1 - S^{\mathcal{Y}}_1 \leq \lambda_{i'} - \tau_{i'} \leq \Delta$, where i' denotes the machine executing job 1 in $\mathcal{V}$.

Now take $j > 1$ and assume that the inequality holds for all $j' < j$. Once again, distinguish the following cases. If $S^{\mathcal{A}'}_j = S^{\mathcal{X}}_j + \xi$ then (1) clearly holds. If $S^{\mathcal{A}'}_j = \kappa - p_j$ then, since $\kappa = C^{\mathcal{A}'}_{j-1}$ we have $C^{\mathcal{A}'}_j = C^{\mathcal{A}'}_{j-1}$. Therefore (1) holds by the inductive assumption. Finally, let i' denote the machine executing j in $\mathcal{V}$, and consider the case when $S^{\mathcal{A}'}_j = \alpha_{i'}$. By the definition of the algorithm $\alpha_{i'}$ is equal to the last completion time on machine i'. Therefore job j is scheduled back-to-back with some other job $j'' < j$, so (1) holds by inductive assumption.

A.3 Other Experimental Results

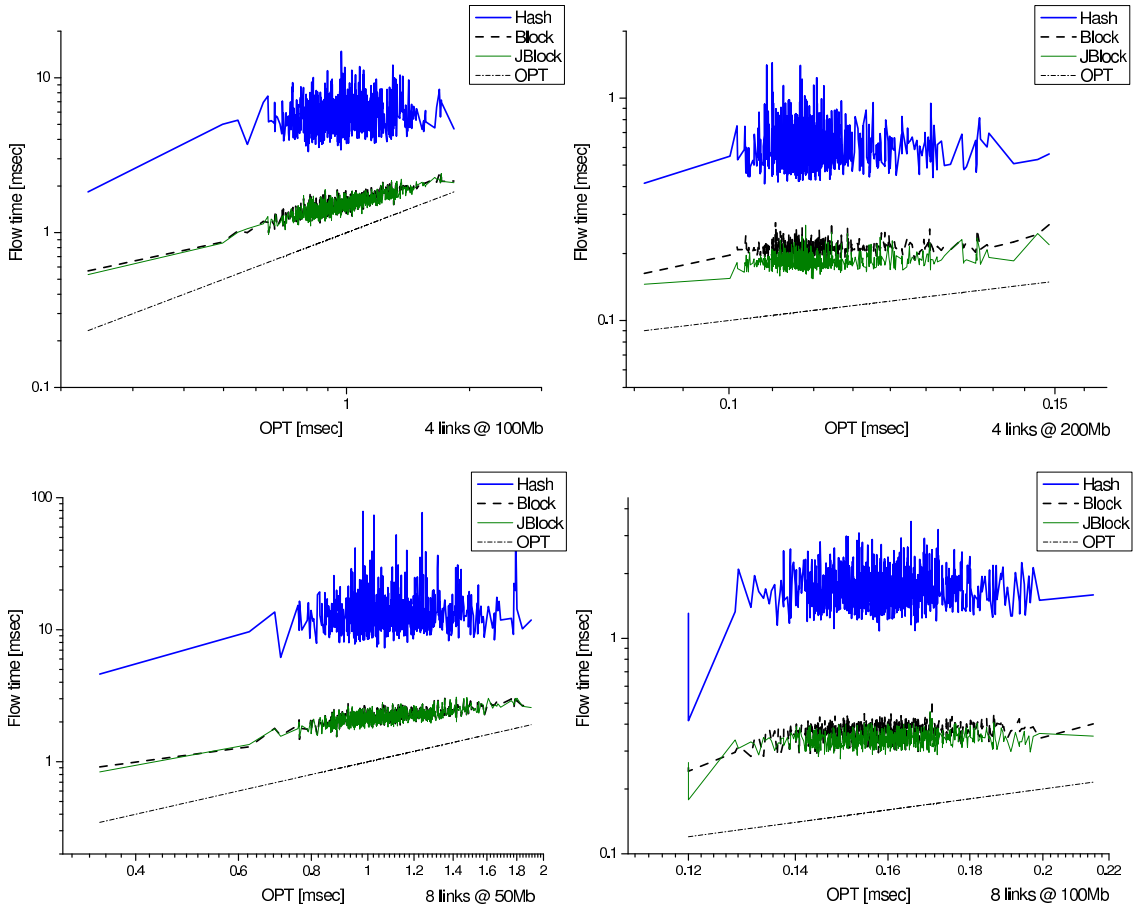

Figure 8: Relative performance of the scheduling algorithms to OPT, on traces collected every Thursday.

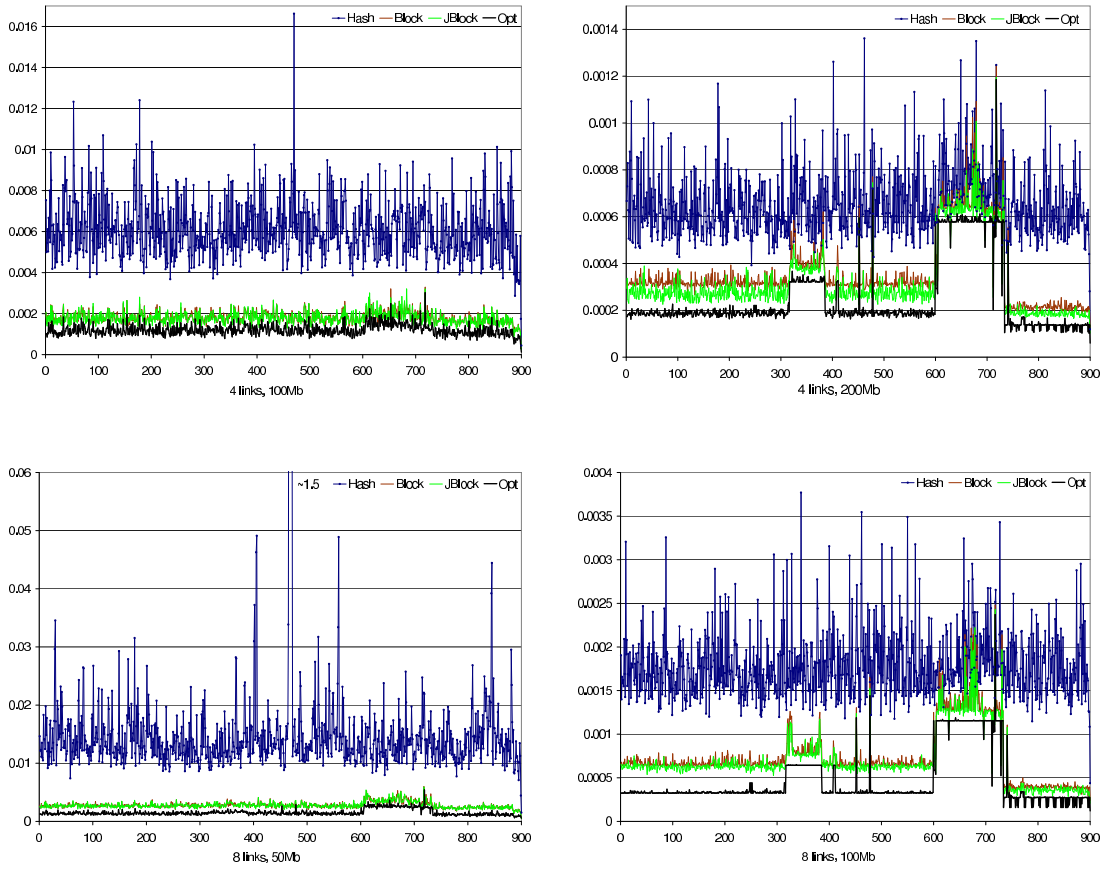

Figure 9: Results of experiments on sample traces collected on Thursdays. The Y-axis shows flow times in seconds, the X-axis time in seconds.