# Run, Fatboy, Run:
# Applying the Reduction to Uniprocessor Algorithm to Other Wide Resources

Andrew Shewmaker Carlos Maltzahn Katia Obraczka
Scott Brandt
*University of California Santa Cruz*

## Abstract

The RUN (Reduction to UNiprocessor) [18, 19, 13] algorithm was first described by Regnier, et al. as a novel and elegant solution to real-time multiprocessor scheduling. The first practical implementation of RUN [3] created by Compagnin, et. al., both verified the simulation results and showed that it can be efficiently implemented on top of standard operating system primitives. While RUN is now the proven best solution for scheduling fixed rate tasks on multiprocessors, it can also be applied to other resources. This technical report briefly describes RUN and how it could be used in any situation involving an array of multiple resources where some form of preemptions and migrations are allowed (although must be minimized). It also describes how buffers can be sanity checked in a system where a RUN-scheduled resource is consuming data from another RUN-scheduled resource.

## 1 Introduction

The RUN algorithm takes advantage of two features of highly loaded systems. First, a busy system has little idle time, so it makes more sense to solve the dual schedule (i.e. when tasks aren't running). The critical nature of idle time was first noticed by Levin, et al. when they created a theory explaining all previous optimal [1] multiprocessor algorithms [12, 11]. Second, a highly loaded system will generally have many small tasks that can be packed together and treated as one task. Both steps simplify the problem at hand, and when they are combined recursively they produce a reduction tree that partitions the scheduling problem amongst the packings and bounds the interactions between packings.

In a system with $N$ processes and $M$ processors where each process requires a fixed share of a processor, packing shrinks the size of $N$ and taking the dual of the system reduces the size of $M$ whenever $N < 2M$ By alternating packing and dual operations, Regnier, et al. showed that they were able to reduce the difficult multiprocessor problem down to a simple uniprocessor problem. The approach is revolutionary because it is simple and provably more efficient in terms of context switches and migrations than any previous approach (e.g the family of proportionate fairness and deadline partitioning scheduling algorithms).

In the remainder of this technical report, section 2 describes general improvements to RUN that will be necessary for practical deployment. Following that, section 3 describes applications to network hardware queues, queuing disciplines, and route management. Section 4 explores scenarios where RUN might be applicable to storage. Finally, section 5 describes how buffers between resources managed by scheduling algorithms like RUN can be semi-automatically sanity checking and debugging.

## 2 Heuristics and Refinements

RUN schedules are valid regardless of task-to-resource assignment strategy, packing scheme (e.g. Worst Fit, Best Fit, Harmonic Period), or single resource scheduling algorithm. This means that RUN can easily be made to support a policy best suited for the purpose at hand: power efficiency, performance, simplicity, NUMA support, etc.

### 2.1 Resource Assignment

The creators of RUN were primarily concerned with producing the set of tasks that should run at any given time. Their task-to processor assignment scheme is simple:

1. leave an executing task on its current processor
2. assign idle tasks to their last-used processor

[1] Optimal in the sense that an algorithm will produce a valid schedule for any task set that is feasible.

3. assign remaining tasks arbitrarily

It might also be worth the effort to keep track of the set of each tasks' *m* previously used resources. That way, if a task must migrate repeatedly, then it is more likely to migrate within a subset of the resources rather than amongst all of the resources within its subsystem. In other words, this heuristic should help on NUMA systems where the cost of migration becomes much larger between NUMA nodes.

## 2.2 Bin Packing

Below are brief descriptions of some different packing heuristics and when they might be most useful.

**Slack Packing** Increases the number of independent partions (decreasing migrations) by adding idle tasks at the end of the first packing (regardless of primary packing heuristic). Of course, this is less applicable as load increases.

**Worst Fit Decreasing Rates** Optimizes spreading large tasks amongst resources, so it would be suitable for less highly loaded situations or when it is more important to use each resource rather than using fewer resources. For instance, this would benefit parallel applications.

**Best Fit Decreasing Rates** Minimizes the number of packings and reduction levels (minimizes preemptions and migrations), so it would be suitable for highly loaded situations or when it is more important to use fewer resources than using all resources.

**Harmonic Periods, Decreasing Rates** Minimizes a server task's job release and deadline events (decreasing preemption and migration events) by first packing by decreasing rate and secondarily preferring packs with similar periods.

These heuristics have been evaluated in terms of performance (i.e. preemptions and migrations). Further work would be necessary to determine which would be appropriate for a power saving policy. It might be that it would be necessary to switch between Worst and Best Fit, as the former might allow all processors to finish more quickly where the latter could keep some cores off entirely.

Certainly other heuristics exist, but among these Slack Packing with Best Fit Decreasing Rates is both simple to implement and one of the best performing heuristics. The Harmonic Periods, Decreasing rates heuristic can achieve 4-5% fewer premptions and migrations, but at the cost of more complexity, at least with the implemention tested by Levin [13]. His heuristic minimizes the Least Common Multiple of the pack's periods. However, it should be simpler to sort tasks by their periods, and then perform Best Fit Decreasing Rate packing using the subset of tasks with periods that are equal or a multiple.

## 2.3 Single Resource Scheduling

RUN was designed to use Earliest Deadline First (EDF) scheduling within packings, but any optimal uniprocessor scheduling algorithm will work. In particular, since RUN already uses knowledge of rates and periods, it would make sense to use the Rate Based Earliest Deadline (RBED) generalization of EDF since it enables integrated scheduling of hard real-time, soft real-time, and best effort tasks [2]. Implementing RUN with EDF is easier than RBED since EDF only uses the tasks' deadline information.

Note that the LITMUS-RT implementation of RUN [3] does allow best effort tasks to use the processor when no real-time tasks are scheduled. While this is useful, RUN with RBED would be simpler than a hierarchical scheduler such as that. Also, RBED can support a broad range of quality of service. For instance, some tasks may only need best effort rates, but do require deadlines to be met.

Implementing RBED requires calculating best effort rates from their individual and combined weights, which affects the reduction phase of RUN. Some minor additional work will be necessary to combine best effort packing with RUN's slack packing heuristic, where idle tasks are used to fill out packings in order to encourage effective partitioning on systems that aren't fully loaded. Best effort slack packing would ensure that the system is work conserving (i.e. that all slack can be used).

RBED defines a Best Effort reserve to ensure that real-time tasks don't use all the available resources. When implementing RBED within RUN, it might be best to have a per resource reserve to encourage work conservation on each. Whether the reserve is global or per resource, it should be made tunable for use by the system administrator.

The theoretical underpinnings of RBED are based on the concept of Resource Allocation and Dispatching (RAD) reservations, which are (rate, period) tuples that obsolete priority classes and previously defined rate-limit specifications. Prior non-realtime scheduling methods possess a limited number of relative, coarse-grained classes (priorities), require rates to be strictly satisfisfied for any measured interval (e.g. Token Bucket Filters), have common periods between all tasks, or have a fixed linear mapping between periods to priorities. RAD reservations enable arbitrarily fine-grained Quality of Service (QoS), possess meanings that stay consistent in a dynamic environment, and allow straightforward reasoning about composing end-to-end QoS.

## 2.4 Dynamic Tasks

The description of RUN, as well as its implementation in simulators and in practice, assumes the entire task set is known a priori and that it doesn't change throughout the life of the system. Certainly this doesn't reflect reality and should be addressed. Furthermore, the work required by the scheduler to adapt to these changes needs to be minimized.

If a task leaves the system, then it can be trivially swapped with an equal idle task without disturbing the schedule (i.e. online slack packing). However, the change in available utilization means that best effort tasks should recalculate their rates and be redistributed. Perhaps it would be best to only do this work for the affected subsystem, even if that results in best effort tasks being treated somewhat unfairly across the entire system.

If a feasible task enters the system and has a rate less than or equal to an idle task, then it can be inserted with little effort. However, if the new task has a rate greater than any one idle task, then RUN must create a new reduction tree. Of course, it would best to maintain the previous reduction tree as much as possible in order to minimize one-time migration penalties. In fact, there is an additional constraint in online packing when compared to offline packing: in addition to task utilizations summing to less than or equal to one, the sums of their remaining budget must be less than the time left until the furthest deadline of the packed tasks. This is always a constraint, but one that it is obviously satisfied when the system is offline.

Best effort tasks make RUN's job both easier and more difficult for dynamic task sets. On the one hand, best effort tasks are flexible and can be packed around the real-time tasks to create proper subsystems. On the other hand, the rates of best effort tasks should be recalculated every time any task enters or leaves the system, although perhaps this should be done a lazily so that the work gets batched even if it negativley impacts fairness. If the task wants more of a guarantee, then it should specify more than a weight.

## 2.5 Sporadic Tasks

Defining all tasks as fixed-rate results in overprovisioning resources for a sporadic task. We refer to this unused utilization as dynamic slack. RUN does not, as of yet, support the sporadic task model, so there is naturally interest in scheduling algorithms that do. It is an open question whether or not RUN can be directly modified to support sporadic tasks—the original creators of it have, in fact, proposed the Quasi Partitioned Scheduling (QPS) algorithm as an alternative [14].

However, if RUN supports both best effort tasks and dynamic task sets as described above, then it doesn't matter if a sporadic task is defined as a fixed-rate task. The dynamic slack can be used by best effort tasks packed with sporadic tasks.

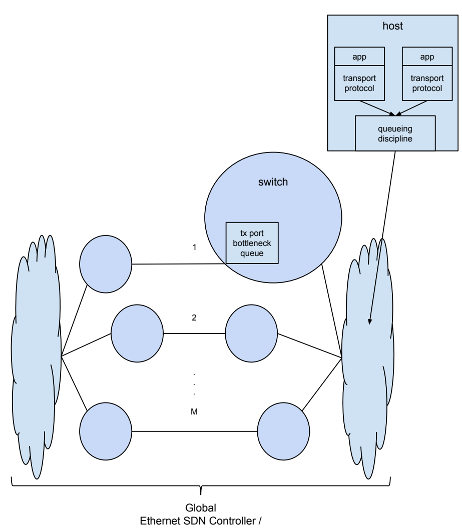
## 2.6 Affinity

Some work needs to be done to make RUN support resource affinity. This is common in the case of processes in a NUMA system that want to be as close to an attached device (e.g. network card or GPU) as possible. Affinity can be thought of as a partial pre-specification of the packings and placements. Pinning a task to a single resource should be easy to take into account, and regular recurring sets (i.e. a NUMA nodes) should also be straightforward to support. However, arbitrary affinity sets may be unworkable since they might over-constrain the packing problem. At any rate, affinity support in RUN requires further investigation.

# 3 Networking

Providing QoS on networks is complex because several independent resources must be managed in concert: transmission and reception queues on the communicating hosts and the transmission queues on the switches. As figure 1 shows, flow is affected by the route it takes, the bottlenecks on that route, and the manner in which a host sends its data.

Figure 1: Network QoS Layers




Even with traffic shaping and a global routing algorithm ensuring that routes aren't overloaded, a bottleneck queue might still build up and drop packets. The bottleneck would have to be able to handle the worst case simultaneous burst from every flow on that route [10]. Furthermore, even lossless networks such as Infiniband suffer from congestion in the form of the Parking Lot Problem [4, 5, 21]. In practice, Infiniband's congestion control isn't enabled because it must be tuned to the specific traffic patterns of the system. If the traffic changes, then overall throughput can be badly hampered.

Therefore, in a traditional networks, the transport protocol still needs to adapt to congestion in order to minimize bottleneck queue usage. Alternatively, if every part of the network fabric implemented a scheduling algorithm such as RUN, congestion wouldn't exist. Alizadeh, et al. showed how EDF might be implemented efficiently in pfabric [1]. It will be interesting to explore whether RUN with RBED might also be implemented efficiently for packet switching.

### 3.1 Host Hardware Queues and Qdiscs

Modern network interface hardware often possesses multiple queues, and Linux has supported them since the 2.6.27 kernel. That support currently allows an administrator several options, including pinning hardware queues to cores or NUMA nodes. While that minimizes context switches and maximizes cache use, it suffers from head-of-line blocking. Alternatively, a round-robin scheduler can be used. While being fair and avoiding head-of-line blocking, round-robin necessarily hurts efficiency. This presents another opportunity for the RUN algorithm. It has provably low numbers of migrations, and can prevent head-of-line blocking while enforcing QoS.

In addition to the multiqueue support already mentioned, a network classifier control group can tag packets to be handled by specific software queueing disciplines (qdiscs). Some of the existing qdiscs are Token Bucket Filters, Stochastic Fair Queueing, Fair Queuing Controlled Delay (FQ_codel), Random Early Detection, and Proportional Integral controller Enhanced. Each of these is an attempt to mitigate congestion or reduce buffer bloat in the network. Most of them concentrate on providing fairness, some provide coarse-grained QoS with priority classes.

Only one qdisc, the Hierarchical Fair Service Curve, claims to support real-time traffic. Configuring a hierarchy of qdiscs to classify and shape traffic is not trivial, and in general must be fine tuned to the network. For example, FQ_codel, a "knobless" qdisc still requires some sort of rate-limiting qdisc working in conjunction with it and is not intended for datacenter networks.

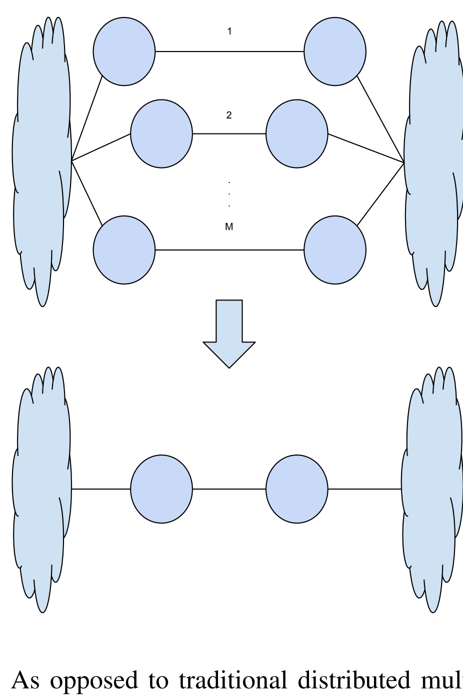
Since RUN can schedule flows according to QoS constraints across multiple hardware queues, it should be much simpler to configure. It would need to be able to classify packets according to flow, if not individual connections. And ideally it would work in concert with a control group designed to tag packets with $rate, period)$ information, in addition to the existing network priority control group.

By creating a RUN qdisc, not only will packets transmitted by Linux hosts be scheduled according to modern real-time theory, but it could lead to a RUN-based network fabric. The Open Virtual Switch (OVS) module in the Linux kernel is intended to be used for both virtual machine networks as well as the operating system on hardware switches, and OVS uses the existing Linux qdiscs to enforce its QoS. The efficacy of RUN can first be tested using Mininet [9], and then on real hardware.

### 3.2 Routes

The edges of networks present an interesting opportunity for RUN. Whether it is a global WAN gateway where bandwidth is extremely limited and precious or the high performance interconnect between a supercomputer and a parallel filesystem, flows should maximize the utilization of the available routes while preventing congestion and data loss. And in cases where there are multiple links or paths that immediately reconverge on the other side, as in figure 2, RUN can be applied.

Figure 2: Reduce to Unipath




As opposed to traditional distributed multipath routing approaches described by Hopps [6], RUN would be used by a SDN (Software Defined Network) Controller or a Subnet Manager (in the case of Infiniband) to assign routes to flows after they have passed the standard RAD admission control test: Would the new flow's rate cause the total flow rate to exceed capacity?

To be clear, RUN would not be discovering the topology of the network. It is scheduling the routes given to it to manage. Also, it would take further work to make RUN take considerations other than a path's cost (e.g. security) into account.

Can RUN be used in the case where the multiple routes aren't immediately recombined as Jain, et al. addressed with B4 [7]? Perhaps, but there are two big concerns. First, how would one partition the graph so that at least the left hand side of the routes looks identical from RUN's perspective? In other words, if the graph is simply bisected, then RUN assumes that any route will work and could overload the left side of the graph. It appears RUN would have to execute recursively on the graph from the edges inward.

That brings up the second concern: when RUN schedules a task it may migrate between several of the resources. At that point, the single fixed-rate task becomes a sporadic task on each of those resources. Treating them each as fixed-rate will quickly exhaust the resources even though best effort tasks could suck up the dynamic slack. An additional concern arises from the best effort tasks. Just because they can use up the dynamic slack at one hop doesn't mean the next hop can handle it.

# 4 Storage

This section is more brief than section 3 because it is not the current focus of the authors' research. However, we share our thoughts concerning RUN as applied to storage below. In general, RUN isn't a clear win for scheduling arrays of storage devices since content is not replicated everywhere. However, RUN could be useful in some scenarios, but would have to fold in lessons learned from Fahrrad and other real-time storage work done at UC Santa Cruz [16, 17, 8].

## 4.1 Multiqueue

Linux is in the process of gaining multiqueue support for storage, just as it did for networking. The block layer gained support first, and it is being followed by the SCSI and Device Manager subsystems. At this point, hardware drivers like NVMe are just beginning to exploit the block device support. From discussions on the mailing list, it appears that the I/O scheduler may be last to gain multiqueue support, and it might be an entirely new "deadline-ish" scheduler rather than a modification to the standard CFQ scheduler.

## 4.2 Reading or Modifying Existing Non-replicated Data

If data isn't replicated across all devices, RUN can schedule reads or modifications to a file or object if the task specification (i.e. RAD reservation) includes affinity information.

## 4.3 Writing New Data

On a distributed system, when the question arises where to store new data, RUN would not be constrained by task affinities and be able to freely manage performance after other considerations, such as available space are answered.

## 4.4 Reading from Replicas

Parallel file systems often replicate data between multiple servers. Usually one server is considered the primary, but it can become overloaded and want to balance its client load with its replicas. As long as consistency among replicas is maintained (trivially true for read-only access), then RUN might be used to schedule use of the replicas. If the distributed storage system uses per server replicas, RUN should work well.

Unfortunately, with regard to Ceph's per-object replicas [20], RUN may not be a good fit. The problem is that different objects won't necessarily share the same set of storage backends. Instead of having many independent RUN schedulers in control of distinct subsets of resources, you would need one instance of RUN scheduling all resources and complicated affinity sets. So RUN might work in this case, but it wouldn't be as parallelized and the affinity of disjoint sets makes the packing problem much harder. It may even constrain the bin packing problem enough that a good packing cannot be found. This problem deserves further thought.

# 5 Buffer Analysis

If RAD schedulers are operating according to their design, then performance is guaranteed. But the RAD model also enables sanity checking on the buffers between schedulers. A producer-consumer model, RAD-Flows [15], derives equations 1 and 2 describing the amount of buffer space $B_{max}$ and time $T_{max}$ is the amount of time it should take for the entire buffer to be rewritten for a well behaved producer/consumer pair of two interacting RAD $(rate, period)$ reservations $(r_p, p_p)$ and $(r_c, p_c)$.

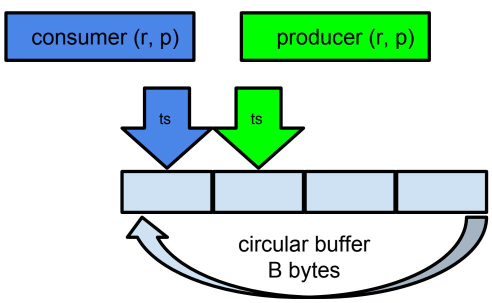
$$T_{max} = \begin{cases} 2p_c & \text{if } p_p \leq p_c \\ 3p_p & \text{if } p_p > p_c \end{cases} \quad (2)$$

$$B_{max} = \begin{cases} 2\left(\left\lceil \frac{p_c}{p_p} \right\rceil + 1\right) r_p p_p - r_p p_p & \text{if } p_p \leq p_c \\ 2 r_p p_p + \max\left(0, r_p p_p - \left(\left\lfloor \frac{p_p}{p_c} \right\rfloor - 1\right) r_c p_c\right) & \text{if } p_p > p_c \end{cases} \quad (1)$$

Given this knowledge, If an application suffers from overflow or underflow, RAD-Flow buffers can always point you toward the problem. You can also guard against the unlikely situation where all RAD schedulers in a chain are misbehaving by producing and consuming too quickly.

The following examples assume that a single circular buffer, as shown in figure 3 can be efficiently accessed simultaneously by the producer and consumer, and a timestamp is recorded whenever there is an attempt to move a pointer. Since we know the amount of time it takes to rewrite a RAD-Flow buffer, the simple circular buffer is sufficient to illustrate the general approach for other buffer data structures.

Figure 3: Circular RAD Buffer




The examples apply to both blocking and non-blocking producers. In the blocking case, overflow doesnt result in lost data and RAD allows us to determine whether the producer is blocking because it is attempting to write too quickly or whether the consumer caused the block by reading too slowly. Non-blocking producers will lose overflowing data and the same tests identify whether the producer or consumer bears responsibility.

If the producer pointer circles around to the consumer pointer (buffer overflow), then there are three possibilities:

1. the producer is sending faster than its reservation
2. the consumer is too slow
3. both 1 and 2

Since the producer has overtaken the consumer, we know that it has rewritten the entire buffer from the consumer pointer on. It must have written to the consumers location before the current value of the consumer timestamp. Because the buffer was sized according to the RAD reservations, we know the producers pointer should not arrive at the consumer pointers location before $ts_c + T_{max}$. Equation 3 uses that information to determine which party is to blame for overflow, and figures 4 and 5 give examples of both cases.

$$\text{producer} = \begin{cases} \text{fast} & \text{if } ts_p - ts_c < T_{max} \\ \text{slow} & \text{if } ts_c - ts_p \geq T_{max} \end{cases} \quad (3)$$

Figure 4: Overflowing RAD Buffer

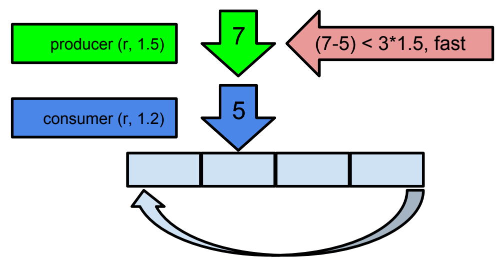
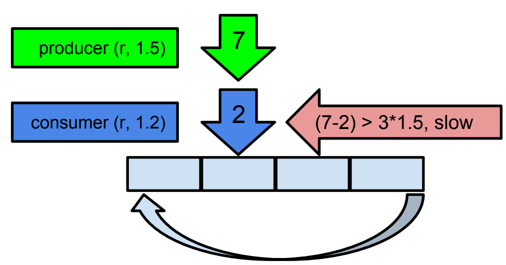



Figure 5: Underflowing RAD Buffer




Similarly, a buffer is underflowing when the consumer pointer circles to the producer pointer. Equation 4 is a mirror to equation 3.

$$\text{consumer} = \begin{cases} \text{fast} & \text{if } ts_c - ts_p < T_{max} \\ \text{slow} & \text{if } ts_p - ts_c \geq T_{max} \end{cases} \quad (4)$$

If both the producer and consumer are misbehaving, then overflow will be blamed on the producer and underflow will be blamed on the consumer. Once their issues are fixed, the buffer will continue to overflow or underflow, but the remaining bad actor will be blamed.

With a chain of reservations, an overflowing upstream consumer might be the victim of a slow downstream consumer. So, if there are several overflowing buffers in a row connecting a chain of RAD reservations, then the blame falls on the furthest downstream consumer. Similarly, the blame for a chain of underflowing buffers percolates up to the furthest upstream producer.

There is another mode of misbehavior that is more difficult to detect. If the producer and consumer are speed-matched but operating too fast or too slowly, then they wont overflow or underflow. However, as long as one part of a chain is behaving correctly, it will point in the direction of bad behavior. The only behavior dangerous to the system as a whole is when all producers and consumers in a chain are too fast. This can be guarded against with a pair of timestamps associated with the beginning of the buffer to track the last time it was produced or consumed (pick one). Whenever the beginning is accessed, the current time is compared to the last time and $T_{max}$, see equation 5.

$$\text{both fast if } ts_{now} - ts_{head} < T_{max} \tag{5}$$

In practice, comparisons will need to tolerate some small room for error to account for scheduling quanta and small indeterminate overheads in timekeeping, etc.

The final case of misbehavior is when every producer and consumer in the chain are too slow, but that would only happen when the ultimate producer is slow. In other words, it will only happen when an application is using a fraction of its reservation. This is not a danger to the system and best-effort applications can benefit from the dynamic slack. References

## Conclusion

This tech report only briefly explores how the Reduction to Uniprocessor algorithm can be applied to other resources. RUN is an elegant algorithm with immense power, and it should enable comprehensive QoS across many layers of resources. In particular, the Radon Network QoS project will be implementing and evaluating RUN in the near future. In addition, RAD buffer theory provides the ability to automatically debug misbehavior.

## References


[1] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 435–446. ACM, 2013.

[2] Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, December 2003.

[3] Enrico Mezzetti Davide Compagnin and Tullio Vardanega. Putting run into practice: implementation and evaluation. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014.

[4] Ernst Gunnar Gran, Magne Eimot, S-A Reinemo, Tor Skeie, Olav Lysne, Lars Paul Huse, and Gilad Shainer. First experiences with congestion control in infiniband hardware. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[5] Ernst Gunnar Gran, Eitan Zahavi, S-A Reinemo, Tor Skeie, Gilad Shainer, and Olav Lysne. On the relation between congestion control, switch arbitration and fairness. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 342–351. IEEE, 2011.

[6] Christian E Hopps and Dave Thaler. Multipath issues in unicast and multicast next-hop selection. 2000.

[7] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14. ACM, 2013.

[8] Tim Kaldewey, Theodore M Wong, Richard Golding, Anna Povzner, Scott Brandt, and Carlos Maltzahn. Virtualizing disk performance. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 319–330. IEEE, 2008.

[9] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[10] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[11] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model

for understanding optimal multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 3–13. IEEE, 2010.

[12] Greg Levin, Caitlin Sadowski, Ian Pye, and Scott Brandt. Sns: a simple model for understanding optimal hard real-time multi-processor scheduling. *Univ. of California, Tech. Rep. UCSCSOE-11-09*, 2009.

[13] Greg M. Levin. *Old And New Approaches To Optimal Real-time Multiprocessor Scheduling*. PhD thesis, University of California Santa Cruz, 2013.

[14] Lima-George Massa, Ernesto, Paul Regnier, Greg Levin, and Scott Brandt. Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014.

[15] Roberto Pineiro, Kleoni Ioannidou, Scott A Brandt, and Carlos Maltzahn. Rad-flows: Buffering for predictable communication. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 23–33. IEEE, 2011.

[16] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 13–25. ACM, 2008.

[17] Anna Povzner, Darren Sawyer, and Scott Brandt. Horizon: efficient deadline-driven disk i/o management for distributed storage systems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 1–12. ACM, 2010.

[18] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 104–115. IEEE, 2011.

[19] Paul D. E. Regnier. *Optimal Multiprocessor Real-time Scheduling Via Reduction To Uniprocessor*. PhD thesis, UFBA-UEFS-UNIFACS, 2012.

[20] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[21] Philip Williams. Congestion in infiniband networks. 2007.