Enhanced EDF Scheduling Algorithms for Orchestrating Network-wide Active Measurements *

Prasad Calyam
OARnet and The Ohio State University
Columbus, OH
pcalyam@oar.net

Chang-Gun Lee†, Phani Kumar Arava, Dima Krymskiy
The Ohio Sate University
Columbus, OH
cglee@ece.osu.edu
{phani,krymskiy}@cse.ohio-state.edu

Abstract

Monitoring network status such as end-to-end delay, jitter, and available bandwidth is important to support QoS-sensitive applications and timely detection of network anomalies like Denial of Service attacks. For this purpose, Internet Service Providers (ISPs) have started to instrument their networks with Network Measurement Infrastructures (NMIs) that periodically run active measurement tasks using measurement servers located at strategic points in their networks. However, one problem that most network engineers have overlooked is the measurement conflict problem*. Since active measurement tasks actively inject test packets to collect measurements along network paths, running multiple active measurements at the same time over the same path could result in misleading reports of network performance. We call this phenomenon a measurement conflict. Our recent observation of such measurement conflict motivates us to form a measurement task scheduling problem of meeting periodicity requirements, where real-time scheduling algorithms can play a role. The scheduling problem, however, is not exactly same as any of the existing scheduling problems in the real-time literature, because the problem involves multiple measurement servers running multiple measurement tasks whose conflict dependency propagates along the chains of paths. For this problem, we propose to use an EDF (Earliest Deadline First) heuristic but allowing "Concurrent Executions" if possible, to construct an offline schedule for a given measurement task set. Also, we propose a novel mechanism to flexibly use the offline schedule for minimizing the response time of dynamic on-demand measurement jobs. Further, we implement and deploy our scheduling algorithms in a real working NMI for monitoring Internet 2 Abilene network.*

*This work has been supported in part by The Ohio Board of Regents.
†The corresponding author is Chang-Gun Lee.

1 Introduction

It has become a norm for Internet Service Providers (ISPs) to instrument their networks with Network Measurement Infrastructures (NMIs) [6, 7, 8, 9] for continuous monitoring and estimation of network-wide status. For this, they use active measurement techniques that actively inject probing packets to collect useful measurements such as end-to-end delay, jitter, loss, bandwidth, etc. The active measurement techniques range from traditional simple tools such as Ping and Traceroute for measuring round-trip delay and topology to more advanced tools such as H.323 Beacon [1], Multicast Beacon [2], Iperf [3], Pathchar [4] and Pathload [5] that use sophisticated packet probing techniques. The measurement data obtained from the above tools can be used for network status estimation such as Network Weather Forecasting [11] and in turn useful for QoS-sensitive applications such as videoconferencing and for timely detection of network anomalies like Denial of Service attacks.

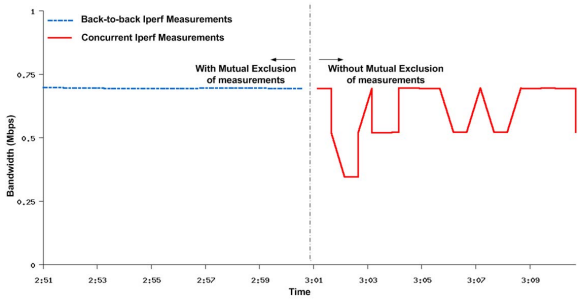


Figure 1. Iperf test results with and without mutual exclusion of measurements

For the correctness of the network status estimation, the periodicity of measurements is important. Thus, most NMIs periodically run measurement tools using *measurement servers* located at strategic points. However, one

problem we recently observed is the *measurement conflict problem*, which has been overlooked by most network engineers. Since active measurement tools consume non-negligible amount of network resources for injecting probing packets, if two or more measurement tasks run concurrently over the same path, they will collide with each other and thus could produce misleading reports.

Our experiment in Figure 1 illustrates the measurement conflict problem. In the experiment, Iperf jobs measure the available bandwidth between two measurement servers connected by a LAN Testbed with a total of 1500 Kbps bandwidth. The background traffic is an H.323 videoconferencing session at 768 Kbps dialing speed and thus the remaining bandwidth should be approximately 732 Kbps. When Iperf jobs run back-to-back with mutual exclusion (shown in the left-half of Figure 1), their measurements are in agreement with our expectation. However, concurrently running multiple Iperf jobs without mutual exclusion (shown in the right-half of Figure 1) causes misrepresentation of remaining bandwidth, merely due to conflicts of multiple Iperf jobs.

This observation motivates a scheduling problem of measurement tasks for orchestrating them to prevent conflicts while still satisfying their periodicity requirements. The nature of the problem is similar to real-time scheduling even though the time granularity of periods (order of minutes) is much coarser than that of the classical real-time systems. The measurement scheduling problem, however, is not the same as any of real-time scheduling problems in the literature in the following two senses: First, more than one measurement jobs can be scheduled at the same time on the same server as long as they can produce the correct measurement data. Second, each measurement job affects multiple measurement servers and their connection paths, thus propagating its conflict dependency with others all across the measurement server topology.

Our goal in this paper is to apply the scheduling techniques developed in the real-time community to the above measurement task scheduling problem with necessary modifications. More specifically, the contributions of this paper can be summarized as follows:

- We propose an offline scheduling algorithm based on the EDF principle but allowing concurrent execution if possible, which can significantly improve the schedulability of a given measurement task set,
- We propose an online mechanism that can steal the maximum possible slack from the offline calculated schedule to serve on-demand measurement requests as soon as possible without violating the periodicity requirements of all the given measurement tasks,
- We actually implement an NMI equipped with the proposed scheduling mechanisms to measure the real-working network, Internet 2 Abilene network.

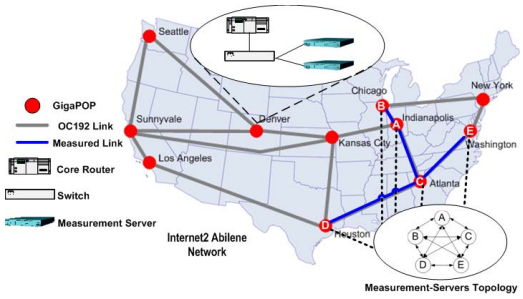


Figure 2. Measurement-Servers Topology constructed for a set of network paths

The rest of this paper is organized as follows: The next section formally defines the measurement task scheduling problem. Section 3 presents our algorithms to schedule offline and online measurement tasks. Section 4 shows our experimental results from both simulations and actual implementation. Section 5 summarizes the related work. Finally, Section 6 concludes the paper.

2 Problem Description and Terminology

An ISP deploys measurement servers at strategic points to continuously estimate the network-wide status. The measurement servers are attached to core routers as shown in Figure 2 to measure the paths to other servers. The paths to be measured are specified by the measurement topology, which can be formally defined by a graph $G = (N, E)$ where N is the set of measurement servers and E is the set of edges between a pair of two servers. Each edge represents a path between two servers to be measured. Figure 2 shows an example complete graph that consists of measurement servers $N = \{A, B, C, D, E\}$ on the Internet 2 Abilene network and edges for all pairs of A, B, C, D, E. This measurement server topology implies that the ISP wants to measure every path in between each pair of A, B, C, D, E.

Another input is the set of measurement tasks. Each measurement task τ_i is specified to measure a path from a source server src_i to a destination server dst_i using an active measurement tool $tool_i$. The measurement should be periodically repeated with period p_i. The execution time of a single measurement instance, called a measurement job, is denoted by e_i. Then, a measurement task can be represented using the similar notion of a real-time periodic task as follows:

$$\tau_i = (src_i, dst_i, tool_i, p_i, e_i).$$

The set of all offline specified measurement tasks is denoted by

$$\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}.$$

In addition to such offline specified measurement tasks, there can be an on-demand measurement request to quickly

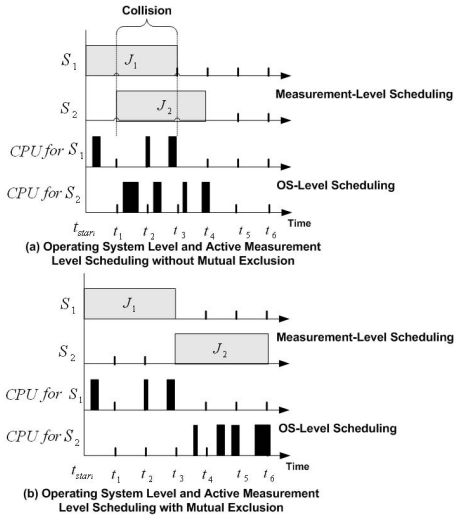


Figure 3. Comparison of scheduling active measurement tasks and operating system tasks

measure a path for admission control of a dynamically arriving QoS-sensitive session or for dynamic selection of an alternative path adapting to network status changes. Such an on-demand measurement request is denoted by

$$J_k = (src_k, dst_k, tool_k, e_k).$$

Our problem is to schedule the offline specified periodic measurement tasks and on-demand measurement requests on top of a given measurement server topology. Here, we have to clearly differentiate the measurement-level scheduling from the traditional OS-level scheduling as depicted in Figure 3. A measurement job is a single execution of an active measurement tool. During the execution, the tool can inject a probe packet and can be suspended until it receives a reply. When it receives the reply, it resumes to accumulate the reply and inject the next probing packet and so on. The duration of such a single execution can last a few minutes to have an accurate measure. Although there can be multiple suspends and resumes in OS-level scheduling, the required duration of a single measurement can be given as a constant number e_i in the measurement-level scheduling. The problem of measurement-level scheduling is to determine the times when we start a measurement tool and when we stop it, not when the OS threads are context switched to others.

For such a measurement-level scheduling problem, one important constraint is mutual exclusive schedule of two measurement jobs if they potentially collide. Since active measurement tools can be CPU intensive for sophisticated calculation and/or channel intensive for injecting a large amount of probing packets, running two measurement tools for an overlapped period over the same measurement server or the same channel can cause a measurement conflict and result in misleading reports by the two tools. Figure 3(a) shows an example of such conflict of two tools; one measures from S_1 to S_2 and the other measures from S_2 to S_1 by injecting a large amount of probing packets into the channels between S_1 and S_2. Figure 3(b) shows a mutual exclusive schedule of two jobs to avoid such conflict. Note that if two tools do not conflict, for example, if they are neither CPU intensive nor channel intensive, they can be scheduled concurrently without any problem on the same channel and even on the same server.

In addition to the mutual exclusion constraint to prevent the measurement conflict, one additional constraint of the measurement-level scheduling problem is the Measurement Level Agreement (*MLA*). Utilizing excessive network resources just for active measurements is not appropriate since it will largely degrade the regular user traffic performance. Thus, we need a regulation on the measurement traffic. Since an end-to-end measurement could involve analyzing data along network paths of multiple ISPs, we can envisage "measurement federations" in which many ISPs participate in inter-domain measurements based on MLAs for reaping the mutual benefits of performing end-to-end path measurements. MLAs could specify that only a certain percentage (1 - 5)% or only a certain number of bits per second (1-2) Mbps of the network bandwidth in ISP backbones could be used for measurement traffic, which can ensure the actual application traffic is not seriously affected by measurement traffic[1]. We use the notation ψ to denote the MLA specification in an NMI. In the measurement-level scheduling problem, the maximum concurrent measurement jobs over the same channel are constrained by ψ.

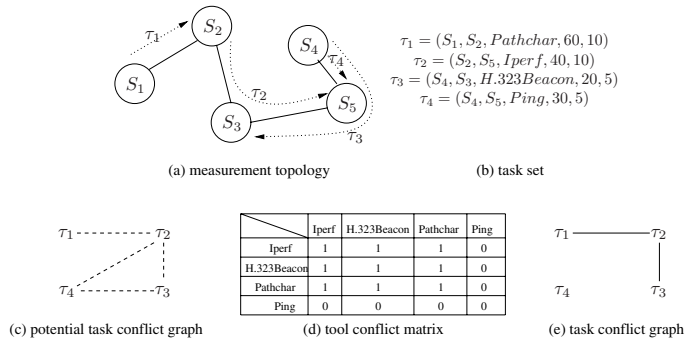
From the above inputs and constraints, the measurement-level scheduling problem can be formally described as follows:

Problem: Given measurement topology $G = (N, E)$ and offline specified measurement task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, find the schedule of measurement jobs such that all deadlines (equal to periods) can be met while preventing conflicts and adhering to the MLA constraint ψ. For an on-demand measurement request J_k, schedule it as early as possible without violating deadlines of tasks in Γ, conflict constraint, and MLA constraint.

3 Measurement Scheduling Algorithms

In this section, we first present an offline scheduling algorithm to construct a schedule table for a given set of periodic measurement tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Then, we present

[1]Since most active measurement tools have options to specify packet sizes and bandwidth usage of a measurement test, simple calculations can be used to determine how much of a network's bandwidth will be used by a given set of active measurements, over a certain period of time.



	Iperf	H.323Beacon	Pathchar	Ping
Iperf	1	1	1	0
H.323Beacon	1	1	1	0
Pathchar	1	1	1	0
Ping	0	0	0	0

Figure 4. Task conflict graph

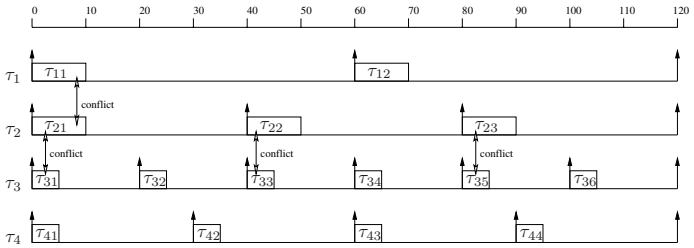


Figure 5. No orchestrated schedule

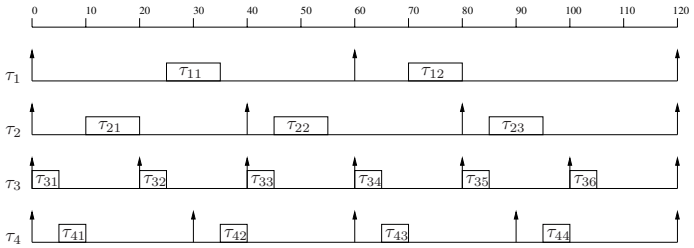


Figure 6. Orchestration based on single processor non-preemptive EDF schedule

an online algorithm to schedule an on-demand measurement request J_k without missing deadlines of periodic tasks.

3.1 Offline Scheduling Algorithm

In our measurement scheduling framework, a central regulator collects all specifications of periodic measurement tasks and builds a schedule table that determines times when measurement jobs can start and stop at each server. To build such a table, the first step is to make a *task conflict graph* by combining the measurement topology G and the task set Γ. Figure 4 shows an example problem. For the given task set, we examine each pair of two tasks τ_i and τ_j to see if they share the same source server, destination server or part of the paths between source and destination servers. If so, the two tasks may "potentially" conflict if scheduled concurrently. In Figure 4, τ_1 and τ_2 share S_2 and thus we add a potential dependency edge between them in the potential task conflict graph. τ_2 and τ_3 share the path and thus a dependency edge is added. On the other hand, τ_1 does not share any network resource with τ_3 and thus no edge is added. Even if two tasks share network resources, they may not actually conflict depending on the active measurement tools used. Based on our empirical study, we could determine which two tools conflict if they run concurrently. The result is summarized by the tool conflict matrix in Figure 4. For example, Iperf and Pathchar conflict if they run concurrently on the same server since both intensively use server and channel resources for active measurement. On the other hand, Ping just injects small probing packets and hence does not conflict with any other tools. Considering the tool conflict matrix, the potential task conflict graph can be converted to the final task conflict graph as in Figure 4(e). The edge between two tasks in the task conflict graph means that they should be scheduled in a mutual exclusive manner, otherwise a conflict happens resulting in misleading reports.

Now, we can consider only the task conflict graph to compute the offline schedule. One obvious solution is to start a measurement job at the source server at its release time without considering mutual exclusion and MLA constraints. Figure 5 shows such a schedule for the problem given in Figure 4. The schedule, however, causes a number of conflicts that result in misleading report of the actual network performance. Another approach is to run only a single measurement job at any time instant using a non-preemptive EDF scheduling algorithm. Figure 6 shows such a schedule for the same problem. It can completely prevent the conflicts. However, it does not allow concurrent execution of multiple jobs even if they do not conflict, which degrades the schedulability.

We aim at finding a schedule in between the two extremes such that conflicts are completely prevented while maximizing the concurrent execution whenever possible. For this, we propose the EDF-CE (i.e., EDF with Concurrent Execution)

algorithm that schedules measurement jobs in the EDF order while allowing concurrent execution if jobs do not conflict. The algorithm is formally described in the following:

EDF-CE: For the given task conflict graph, find the measurement schedule during a hyperperiod

---

Input: task set Γ and task conflict graph
Output: start time st_{ij} and finish time ft_{ij} for each job τ_{ij} in a hyperperiod
begin procedure
1. Initialize rt_list with the ordered list of all release times in a hyperperiod
2. Initialize ft_list = {} /* ordered list of finish times*/
3. Initialize $pending\_job\_queue$ = {}
4. **do**
5. $time$ = get the next scheduling time point from rt_list and ft_list
6. add all newly released jobs at $time$ to $pending\_job\_queue$ in EDF order
7. **for** each job τ_{ij} in $pending\_job\_queue$ in EDF order
8. **if** τ_{ij} does not conflict with any of already scheduled jobs **and**
9. scheduling τ_{ij} at $time$ does not violate MLA constraint ψ
10. $st_{ij} = time$ and $ft_{ij} = time + e_i$
11. **if** ft_{ij} is later than the deadline of τ_{ij}
12. return error /* infeasible task set */
13. **end if**
14. remove τ_{ij} from $pending\_job\_queue$
15. add ft_{ij} to ft_list in order
16. **end if**
17. **end for**
18. **until** $time == hyperperiod$
end procedure

The EDF-CE algorithm maintains the ordered list of release times rt_list and the ordered list of finish times ft_list. Line 1 initializes rt_list with all release times in a hyperperiod. In Figure 7, the release times are 0, 20, 30, 40, 60, 80, 90, 100, and 120. Line 2 initializes ft_list as empty since no job is scheduled yet. In addition, the algorithm maintains a $pending\_job\_queue$ that holds all jobs released but not scheduled, in the EDF order. Line 3 initializes it as empty. The **do-until** loop from Line 4 to Line 18 progresses the virtual time variable $time$ until a hyperperiod while determining the schedule at all scheduling time points, i.e., release times and finish times. Line 5 moves $time$ to the next scheduling time point. Then, Line 6 adds all newly released jobs to the $pending\_job\_queue$. The **for** loop from Line 7 to Line 17 examines the pending jobs in the EDF order and determines whether they can start at $time$ without causing any conflict and without violating MLA ψ (see Lines 8 and 9). If so, the job τ_{ij}'s start time st_{ij} is determined as $time$, its finish time is determined as $time + e_i$ in Line 10, and it is removed from the $pending\_job\_queue$ in Line 14. The

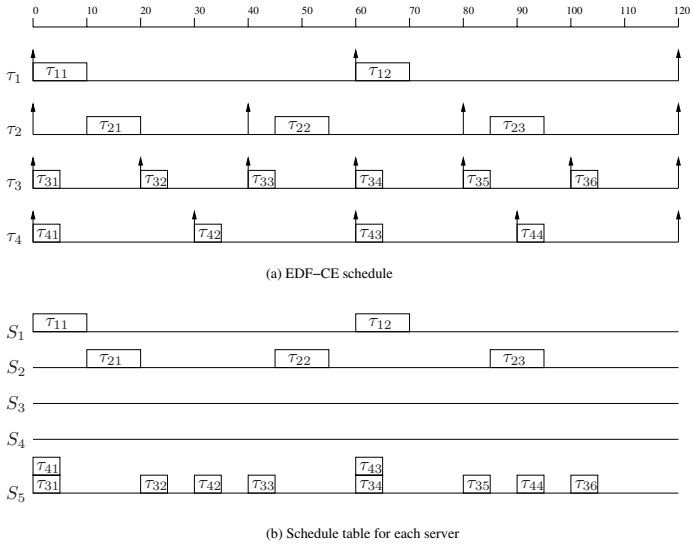


Figure 7. EDF-CE Schedule

finish time ft_{ij} is added to ft_list. Otherwise, the job is kept in the $pending\_job\_queue$ to be considered at the next scheduling time point. Note that the algorithm tries to concurrently start as many jobs as possible in the EDF order at $time$ as long as they neither conflict nor violate the MLA. Figure 7(a) shows such EDF-CE schedule for the same problem of Figure 4. At time 0, τ_{31} is scheduled first since it has the earliest deadline. The second earliest job τ_{41} can also be concurrently scheduled even though it runs on the same server S_4 as τ_{31}, since τ_3 and τ_4 have no conflict edge in the task conflict graph of Figure 4(e). The third earliest deadline job τ_{21}, however, cannot start at the same time since it conflicts with τ_{31}. On the other hand, τ_{11} can start since it does not conflict with the overlapping jobs τ_{31} and τ_{41}—no edge between τ_1 and τ_3, τ_4. The pending job τ_{21} is re-examined at the next scheduling point, which is the finish time 5 of τ_{31} and τ_{41}. At this time, τ_{21} conflicts with τ_{11} and thus cannot be scheduled. τ_{21} can be eventually scheduled at time 10. All the rest of the timeline is similarly filled in.

Once we find the EDF-CE schedule, we can convert it to the measurement schedule table of each server considering the source server of each job. Figure 7(b) shows the schedule tables of all the five servers. Note that τ_{31} and τ_{41} start at the same time on the same server S_5 since they do not conflict in the measurement level, which clearly differs from the traditional real-time job scheduling problems. Such constructed schedule tables are transferred to corresponding servers so that they can start and stop the planned measurement jobs.

3.2 On-line Scheduling of On-Demand Measurement Requests

At the run time while each server is executing periodic measurement tasks according to the pre-computed schedule table, a network engineer can request an on-demand mea-

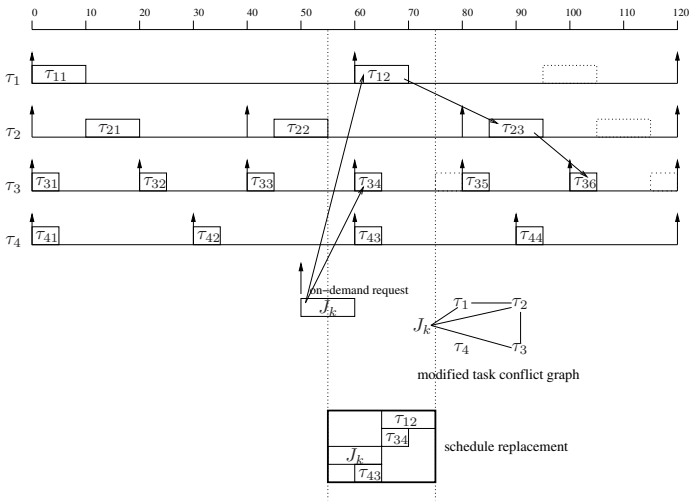


Figure 8. Recursive pushing for maximum slack calculation

surement J_k. We assume that such a request is received by the central regulator for the sake of simplicity. However, using a distributed agreement mechanism [12, 13], the on-demand request scheduling algorithm described below can be realized in a decentralized way without limitation.

Upon the arrival of an on-demand request $J_k = (src_k, dst_k, tool_k, e_k)$, our goal is to serve it as soon as possible without missing any deadlines of periodic measurement tasks. For this, we propose a method to calculate the maximum slack using a *recursive pushing* mechanism. The basic idea can be best illustrated by Figure 8 that shows the same EDF-CE schedule as above. Suppose that an on-demand request $J_k = (S_2, S_3, Iperf, 10)$ arrives at time 50. It conflicts with τ_1, τ_2, and τ_3 as shown by the modified task graph. The central regulator cannot allow J_k to start at time 50 since it conflicts with τ_{22} and τ_{22} cannot be preempted once started. Thus, the central regulator calculates the maximum slack from time 55 that can be continuously used for J_k. For this, the central regulator calls **push**(τ_{12}) and **push**(τ_{34}) to determine how much τ_{12} and τ_{34} can be pushed to make the maximum slack for J_k. The **push** operation is recursive. To determine the maximum **push** of τ_{12}, we first have to know the maximum **push** of the dependent job τ_{23}. Thus, **push**(τ_{12}) recursively calls **push**(τ_{23}). On the other hand, τ_{34} does not conflict with any other job while being pushed up to its deadline $d_{34} = 80$. Such job is called a *terminal job*, and its new pushed finish time new_ft_{34} can be simply given by its deadline d_{34} and new start time new_st_{34} is $new\_ft_{34} - e_3$. The **push** operation is formally defined as follows:

---

push: return the new start time of input jobs after maximum push

---

Input: τ_{ij}

Output: new start time after maximum push new_st_{ij}
begin procedure
1. **if** τ_{ij} has no conflicting jobs scheduled up to d_{ij}
/* terminal node */
2. new finish time $new\_ft_{ij} = d_{ij}$
3. new start time $new\_st_{ij} = new\_ft_{ij} - e_i$
4. **else** /* not terminal node */
5. new finish time $new\_ft_{ij} = d_{ij}$
6. **for** each conflicting task $\tau_{i'j'}$
7. new_ft_{ij} = min (new_ft_{ij}, **push**($\tau_{i'j'}$))
8. **end if**
9. new start time $new\_st_{ij} = new\_ft_{ij} - e_i$
10. **end if**
11. return new_st_{ij}
end procedure

This algorithm returns the new start time new_st_{ij} after maximally pushing τ_{ij}. If τ_{ij} is a terminal node, Lines 2 and 3 can simply calculate the new_st_{ij} from the deadline. Otherwise, Lines 6, 7, and 8 recursively call **push** for all dependent jobs to figure out the minimum new start time of all dependent jobs, which in turn gives the minimum new finish time new_ft_{ij} of τ_{ij}. With new_ft_{ij}, Line 9 calculates the new start time as $new\_st_{ij} = new\_ft_{ij} - e_i$. Finally, Line 11 returns new_st_{ij}.

Considering new_st_{ij}, we can calculate the maximum slack that can be used for the on-demand request J_k starting from time t. If the maximum slack is larger than the required execution time e_k, the central regulator sets time t as the start time of J_k and push dependent periodic jobs as needed. The piece of schedule affected by J_k (see "schedule replacement" in Figure 8) is transferred to the corresponding servers so that they can temporarily use the updated schedule piece instead of the original schedule, to accommodate J_k. If the maximum slack at time t is not larger than e_k, the central scheduler examines the next scheduling point to recalculate the maximum slack and so on, until it finds enough slack. In practice, it takes time to calculate the maximum slack and transfer the updated schedule piece to the servers. Such delay is order of milliseconds as will be shown Section 4. Thus, our algorithm can treat the maximum delay as a lead time and can start calculating the maximum slack from the current time plus the maximum delay. This way, the updated schedule piece can be used only after it is received by all the servers in a synchronized way.

This recursive push algorithm allows almost immediate service of on-demand requests most of time. As a result, the average response time of on-demand requests can be minimized without missing deadlines of periodic measurement tasks.

4 Experimental Results

In this section, we evaluate the performance of our measurement scheduling algorithms. We first perform simulations with synthetic measurement tasks to show the maxi-

mum schedulability by the EDF-CE algorithm and the average response times of on-demand requests by the recursive pushing algorithm. Then, we present performance evaluation results on an actual Internet testbed.

4.1 Performance Evaluation Results using Synthetic Tasks

Our synthetic task set is comprised of four periodic active measurement tasks τ_1, τ_2, τ_3 and τ_4. The period p_i of each task τ_i is randomly generated from [1000, 10000]. The execution time e_i of each task τ_i is randomly generated from [100, 999]. The task conflict graph of the four tasks is also randomly created using a parameter called a *conflict factor*. The conflict factor represents the probability that there is a conflict edge between any two tasks. Therefore, when the conflict factor is 1, the task conflict graph is fully connected. If the conflict factor is 0, there is no edge between tasks.

For each sample task set and task conflict graph, we use the maximum schedulable utilization $\sum_{i=1}^{4} e_i/p_i$ as the performance metric. We determine the maximum schedulable utilization by gradually increasing execution times e_i until the scheduling algorithms fail to construct a feasible schedule.

We compare three scheduling algorithms:

- **No-orchestration** that schedules measurement jobs at their release times without considering measurement conflicts,
- **EDF** that schedules only one measurement job at a time using the non-preemptive EDF algorithm just like a single processor EDF scheduling, and
- **EDF-CE** that is proposed in this paper.

Figure 9 shows the maximum utilization as increasing the conflict factor. Here, we assume a large MLA ψ and thus it is not a bottleneck when finding the schedule. Each plotted point in the figure is the average of 1000 random sample task sets. EDF's maximum utilization is constantly bounded under 100% regardless of the conflict factor since it does not allow concurrent execution even if possible. On the other hand, our EDF-CE algorithm can maximally utilize the concurrent execution whenever possible. When the conflict factor is zero, EDF-CE allows concurrent execution of all four tasks. This is similar to scheduling the four tasks on four independent processors. Thus, the maximum utilization reaches up to 400%. As the conflict factor increases, the maximum utilization gradually decreases. When the conflict factor is 1, i.e., when all four tasks conflict each other, EDF-CE automatically degenerates to the single processor EDF and hence gives the maximum utilization of 100%. The result shows that EDF-CE is leveraging the "maximal but only possible" concurrent execution by explicitly considering the conflict dependency among tasks. The no-orchestration approach always gives the maximum utilization of 400% since all four

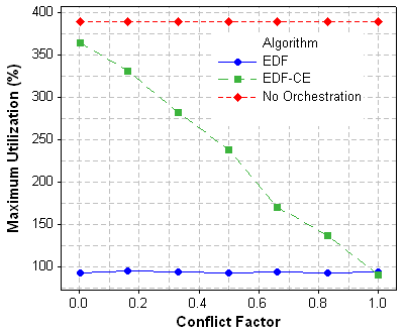


Figure 9. Maximum utilization by three scheduling algorithms

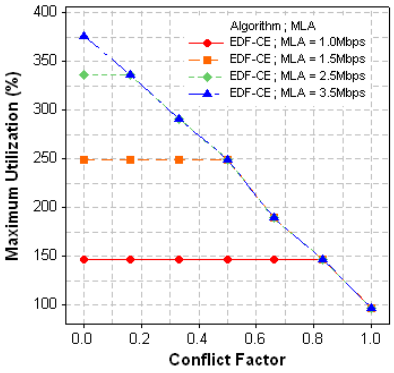


Figure 10. Effect of MLA ψ and conflict factor to EDF-CE

tasks can be concurrently executed ignoring the conflict dependency. This, however, causes many conflicts as will be shown in Section 4.2 resulting in many misleading reports of actual network performance.

Figure 10 illustrates how the maximum utilization of EDF-CE is bounded by the MLA constraint ψ and conflict factor. As expected, higher values of ψ accommodate a larger number of concurrent jobs and hence produce higher maximum utilization. For a ψ value, the maximum utilization is constant up to a certain point of the conflict factor and then starts decreasing. Such a trend explains that ψ is the bottleneck when the conflict factor is small, whereas the conflict dependency becomes the bottleneck when the conflict factor is large.

To study the performance of the "recursive push" algorithm for handling on-demand measurement requests, we simulate random arrivals of on-demand jobs and schedule them over the offline EDF-CE schedule. The offline specified task set consists of four periodic tasks as before, and their execution times and periods are randomly generated from [1 minute, 10 minutes] and [20 minutes, 200 minutes], respectively. The execution times and inter arrival times of

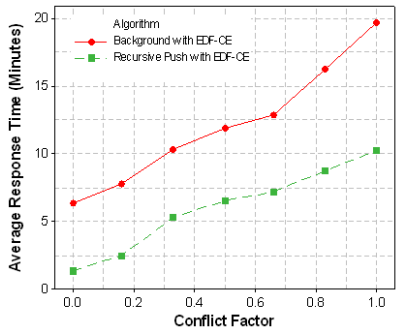


Figure 11. Average response time of on-demand jobs

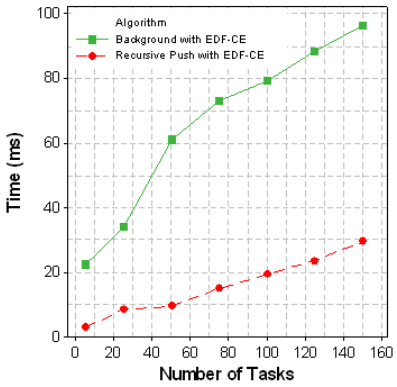


Figure 12. Online schedule overhead for on-demand jobs

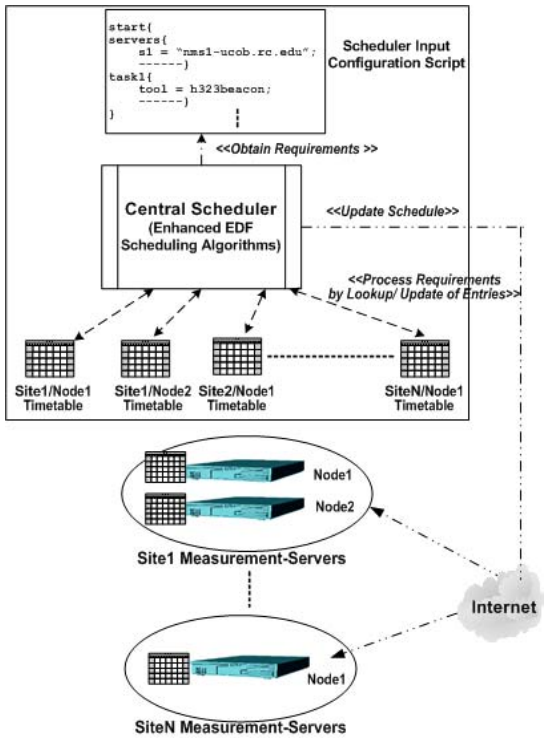


Figure 13. Structure of Measurements Scheduling Framework

on-demand jobs are also randomly generated from [1 minute, 10 minutes] and [20 minutes, 200 minutes], respectively. The performance metric is the average response time, i.e., time lag between the arrival time and the completion time, for 1000 on-demand jobs. We compare our recursive push algorithm with a background approach that schedules an on-demand job in the earliest gap present in the offline EDF-CE schedule within which the on-demand job can execute to completion. Figure 11 shows that our recursive push algorithm can significantly improve the responsiveness for on-demand measurement requests. Note that the average response time in both the background and recursive push cases increases as the conflict factor increases. This is because a higher conflict dependency among tasks reduces the concurrent execution of jobs and thus reduces the gaps available to schedule the on-demand jobs.

To estimate the overhead of online scheduling, we measure the algorithm running time for each on-demand job on 2.4 GHz Pentium 4 Linux PC. Figure 12 shows the average times as increasing the number of periodic tasks, i.e., the problem size, while fixing the conflict factor as 0.8. Even for a large number of periodic tasks with a high conflict factor, our recursive push algorithm can find the slack and calculate the updated schedule within tens of milliseconds. This is a negligible delay comparing with typical measurement task execution times in the order of minutes.

4.2 Performance Evaluation Results on an Internet Testbed

We have actually implemented and deployed our scheduling algorithms in a NMI that is being used to monitor network paths on the Internet 2 Abilene network backbone. The scheduling framework consists of a "Scripting Language Interface" and a central scheduler as shown in Figure 13. The scripting language interface provides a generic and automated way to input measurement specifications such as measurement server topology, periodic measurement tasks, and MLAs. These specifications are interpreted by the central scheduler to construct schedule timetables for the measurement servers. The constructed schedule timetables are transferred to the corresponding servers to initiate the measurement jobs at the planned times.

Our Internet testbed has five sites each of which are equipped with two measurement servers as shown in Figure 14(a). To collect the actual measurement data, we run five periodic measurement tasks as shown Figure 14(b). The resulting task conflict graph is shown in Figure 14(c).

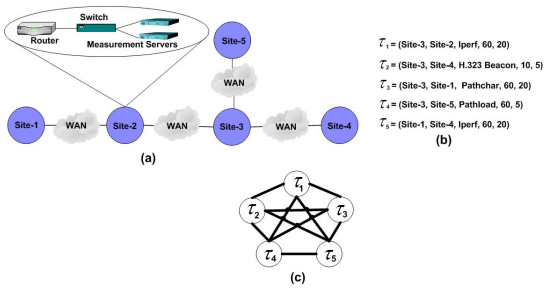


Figure 14. Internet Testbed Setup

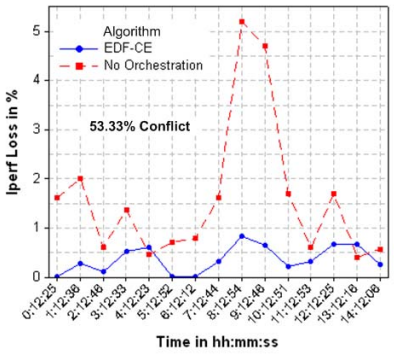


Figure 15. Iperf Loss measurements between Site-2 and Site-3

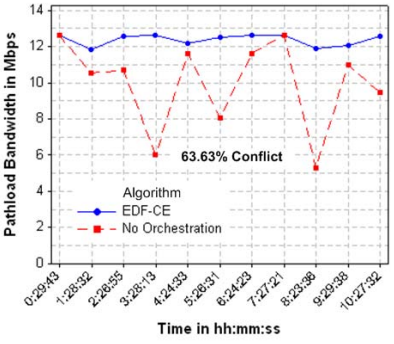


Figure 16. Pathload Bandwidth measurements between Site-3 and Site-5

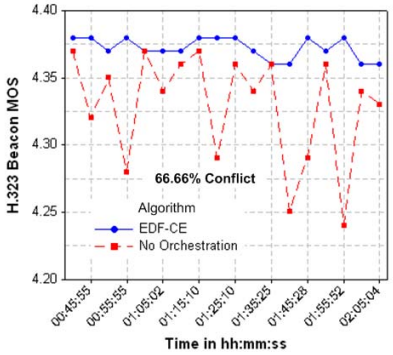


Figure 17. H.323 Beacon MOS measurements between Site-3 and Site-4

Figure 15 shows Iperf packet loss reports[2] measured between Site-2 and Site-3 by task τ_1. The proposed EDF-CE guarantees zero conflict with other tasks and the reported data correctly represents the inherent fluctuation of actual packet loss status between Site-2 and Site-3. In contrast, the no-orchestration method results in 53.33% conflicts of τ_1 with other tasks. As a result, the measurement reports abnormally fluctuate, which does not necessarily mean that the actual network status has such fluctuation.

Figure 16 shows the Pathload Bandwidth reports[3] measured between Site-3 and Site-5 by τ_4. Figure 17 shows the H.323 Beacon MOS reports[4] measured between Site-3 and Site-4 by τ_2. From these two figures, we can make the similar observation as in Figure 15 justifying the importance of measurement orchestration for the correct estimation of network status.

5 Related Work

Recent studies [6, 7] have developed Network Measurement Infrastructures using active measurement techniques to monitor and estimate the network status. They however have not paid much attention to the measurement conflict problem. They commonly create cron jobs that initiate active measurements without any considerations for avoiding measurement conflicts. Such approaches result in erroneous measurement results that do not accurately reflect the actual

[2]Loss measurements reported by Iperf correspond to the number of packets lost while transferring UDP packets similar to a typical UDP application flow along a path between two measurement sites. Iperf uses packet flooding methodology [3].

[3]Bandwidth measurements reported by Pathload correspond to the "available bandwidth" for a path, without affecting the rest of the traffic along the path between two measurement sites. Pathload uses Self-Loading Periodic Streams (SLOPS) methodology [5].

[4]MOS measurements reported by the H.323 Beacon are generated by VoIP traffic emulation and are useful in evaluating network capability to support Voice and Video over IP (VVoIP) applications. The MOS values are reported on a quality scale of 1 to 5; 1-3 range being poor, 3-4 range being acceptable and 4-5 range being good [1]. MOS values close to 4.41 are desirable for high-quality VVoIP.

network conditions. The NMIs in [8, 9] either use round-robin or resource broker methods for avoiding the measurement conflicts. However, they do not perform well in terms of schedulability and thus they fail to schedule many measurement tasks initiated between many measurement servers.

The Network Weather Service (NWS) [11] uses time-series models to forecast the network performance. These models require a consistent periodicity of active measurements in addition to accurate network status information. To meet these requirements, a token-passing mechanism was proposed [10] that allows only a single server in possession of a token to initiate measurements to all the other measurement servers in the NMI . However, it cannot leverage possible concurrent execution of multiple measurement jobs and hence limits the schedulability.

Our scheduling framework is similar to the joint-scheduling algorithms in [14, 15], in the sense that they construct an offline schedule for static tasks and steal unused slots to schedule aperiodic jobs. However, their scheduling problem is inherently a computation job scheduling problem on processors and thus they cannot address the nature of measurement scheduling problem such as concurrent execution of multiple jobs on the same server and task conflict among multiple servers across the communication channels.

6 Conclusion and Future Work

In this paper, we identify the measurement conflict problem, which results in misleading measurements of network status when multiple conflicting measurement tools are executing at the same time on the same server or path. From the observation, we formulate the measurement scheduling problem as a real-time scheduling problem.

For the optimal schedulability of periodic measurement tasks, we use the EDF principle, which has been proven to be optimal in single processor preemptive scheduling and perform well in general settings. Our significant enhancement is to leverage concurrent execution, which clearly differentiates the measurement scheduling problem from the classical real-time scheduling problems. Our enhanced EDF algorithm called EDF-CE allows concurrent execution of multiple measurement jobs not only on the isolated servers and paths but also on the same server and path, as long as they do not conflict—no misleading reports. This significantly improves the schedulability and thus allows us to run measurements more frequently or save much time for on-demand requests.

We also propose an online scheduling algorithm to serve on-demand measurement requests as early as possible. The online algorithm can steal the maximum slack without violating any periodic deadlines and thus can almost immediately schedule the on-demand requests. Therefore, the response times of on-demand requests can be significantly reduced.

All the proposed scheduling algorithms have actually been implemented and deployed on Internet 2 Abilene network. The actual experimental results demonstrate the pertinence and trustworthiness of our proposed scheduling algorithms.

In the future, we plan to implement our measurement scheduling framework in a decentralized way by using a distributed synchronization method. We believe that having a decentralized framework can improve the flexibility in deploying our algorithms on wider-basis in multidomain "measurement federations" on Internet backbones.

References

[1] P. Calyam, W. Mandrawa, M. Sridharan, A. Khan, P. Schopis, "H.323 Beacon: An H.323 application related end-to-end performance troubleshooting tool", ACM SIGCOMM NetTs, 2004.

[2] Multicast Beacon - http://dast.nlanr.net/Projects/Beacon

[3] A. Tirumala, L. Cottrell, T. Dunigan, "Measuring end-to-end bandwidth with Iperf using Web100 ", Passive and Active Measurement Workshop, 2003.

[4] A. Downey, "Using Pathchar to estimate Internet link characteristics", ACM SIGCOMM, 1999.

[5] C. Dovrolis, P. Ramanathan, D. Morre, "Packet Dispersion Techniques and Capacity Estimation", IEEE/ACM Transactions on Networking, 2004.

[6] L. Cottrell, C. Logg, I. Mei, "Experiences and results from a new high performance network and application monitoring toolkit", Passive and Active Measurement Workshop, 2003.

[7] M. Luckie, A. McGregor, "IPMP: IP Measurement Protocol", Passive and Active Measurement Workshop, 2002.

[8] P. Calyam, D. Krymskiy, M. Sridharan, P. Schopis, "TBI: End-to-end network performance measurement testbed for empirical bottleneck detection", IEEE TRIDENTCOM, 2005.

[9] E. Boyd, J. Boote, S. Shalunov, M. Zekauskas, "The Internet2 E2E piPES Project: An interoperable federation of measurement domains for performance debugging", Internet2 Technical Report, 2004.

[10] B. Gaidioz, R. Wolski, B. Tourancheau, "Synchronizing Network Probes to avoid Measurement Intrusiveness with the Network Weather Service", IEEE High-performance Distributed Computing Conference, 2000.

[11] R. Wolski, N. Spring, J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", Future Generation Computer Systems, 1999.

[12] M. H. Park and M. Kim, "A distributed synchronization scheme for fair multi-process handshakes", Information Processing Letter, Vol.34, No.3, pp.131-138, 1990.

[13] A. Fujii, M. Nakamura, and Y. Nemoto, "A fast sequential distributed synchronization protocol", Systems and Computers in Japan, Vol. 29, No. 12, pp.11-18, 1999.

[14] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems", Proc. of the IEEE Real-Time Systems Symposium, Dec. 1995.

[15] D. Isovic and G. Fohler, "Handling Sporadic Tasks in Off-line Scheduled Distributed Real-Time Systems", Proc. of Euromicro Conference on Real-Time Systems, July 1999.