Hybrid EDF Packet Scheduling for Real-Time Distributed Systems

Tao Qian, Frank Mueller
North Carolina State University, USA
mueller@cs.ncsu.edu

Yufeng Xin
RENCI, UNC-CH, USA
yxin@renci.org

***Abstract*—When multiple computational resource elements collaborate to handle events in a cyber-physical system, scheduling algorithms on these resource elements and the communication delay between them contribute to the overall system utilization and schedulability. Employing earliest deadline first (EDF) scheduling in real-time cyber-physical systems has many challenges. First, the network layer of a resource has to interrupt and notify the scheduler about the deadlines of arrived messages. The randomness of interruption makes context switch costs unpredictable. Second, lack of globally synchronized clocks across resources renders event deadlines derived from local clocks and piggybacked in messages meaningless. Third, communication delay variances in a network increase the unpredictability of the system, e.g., when multiple resources transmit message bursts simultaneously. We address these challenges in this work. First, we combine EDF scheduling with periodic message transmission tasks. Then, we implement an EDF-based packet scheduler, which transmits packets considering event deadlines. Third, we employ bandwidth limitations on the transmission links of resources to decrease network contention and network delay variance. We have implemented our hybrid EDF scheduler in a real-time distributed storage system. We evaluate it on a cluster of nodes in a switched network environment resembling a distributed cyber-physical system to demonstrate the real-time capability of our scheduler.**

I. Introduction

When computational resource elements in a cyber-physical system are involved in the handling of events, understanding and controlling the scheduling algorithms on these resource elements become an essential part in order to make them collaborate to meet the deadlines of events. Our previous work has deployed a real-time distributed hash table in which each node employs a cyclic executive to schedule jobs that process data request events with probabilistic deadlines, which are resolved from a hash table [21]. However, cyclic executives are static schedulers with limited support for real-time systems where jobs have to be prioritized according to their urgency. Compared to static schedulers, EDF always schedules the job with earliest deadline first and has a utilization bound of 100% on a preemptive uniprocessor system [15], [14]. In theory, two major assumptions have to be fulfilled in real-time systems in order to achieve this utilization bound. First, the scheduler has to know the deadlines of all jobs in the system, so it can always schedule the job with the shortest deadline. Second, the cost of a context switch between different jobs is ignorable. To satisfy the second assumption, not only the cost of a single context switch, but also the number of context switches must be ignorable. This constrains the number of job releases in a period of time. Employing EDF schedulers in such a distributed system raises significant challenges considering the difficulty of meeting these assumptions in a cyber-physical system in which computational resource elements (i.e., nodes) communicate with each other by message passing.

First, to handle an event in a cyber-physical system, messages of the event are passed among nodes to trigger jobs on each node so that the event can be eventually processed. When messages arrive at a particular node, they are queued in the operating system network buffer until the job scheduler explicitly receives these messages and releases corresponding jobs to process them. The delays between the arrival and reception of messages provide a limitation on the capability of the job scheduler in terms of awareness of the deadlines of all current jobs on the node. One possible solution to address this problem is to let the operating system interrupt the job scheduler whenever a new message arrives at the node. In this way, the scheduler can always accept new messages at their arrival time so that it knows the deadlines of all current jobs. However, this may result in an unpredictable cost of switching between the job execution context and the interruption handling context that receives messages.

To address this problem, we combine a periodic message receiving task with the EDF scheduler. This receiving task accepts messages from the system network buffer and releases corresponding jobs into job waiting queues of the EDF scheduler. Considering the fact that message jobs can only be released when the receiving task is executing, this design makes the scheduler partial-EDF because of the aforementioned delays. However, it increases the predictability of the system in terms of temporal properties including period, worst case execution time, and relative deadline of the receiving task, which also makes context switch cost more predictable. In addition, our partial-EDF adopts a periodic message sending task to regulate the messages sent by events so that the inter-transmission time of messages has a lower bound. We have theoretically studied the impact of the temporal properties of these transmission tasks on the schedulability of partial-EDF.

The second challenge is relevant to the deadlines carried in event messages. Since clocks of nodes in cyber-physical systems are not synchronized globally considering the cost of global clock synchronization [11], deadlines carried in messages sent by different nodes lose their significance when these messages arrive at a node. To address this problem, our scheduler maintains a time-difference table on each node consisting of the clock difference between the senders and this receiving node. Hence, the deadlines carried in messages can be adjusted to the local time based on the information in this time-difference table. Thus, the EDF order of jobs can be maintained. These time-difference tables are built based on the

Network Time Protocol (NTP) [17], and they are periodically updated so that clock drifts over time can be tolerated.

The third challenge is to optimize the underlying network that the cyber-physical system utilizes to transmit event messages between nodes to provide a bounded network delay. We assume that even in a real-time distributed system where all nodes release distributed jobs periodically, the number of corresponding messages received by the receiver nodes cannot be guaranteed to be periodic due to the inconsistent delay of the underlying network. Since the jobs that handle these messages could send messages to further nodes (when distributed jobs require more than two nodes to handle them), the variance in the numbers of messages can potentially increase the burstiness of the network traffic, which decreases the predictability of network delay. Our evaluation in Section IV-C proves this hypothesis. We address this challenge in two ways. First, we propose an EDF-based packet scheduler that works on the system network layer on end nodes to transmit packets in EDF order. Second, since network congestion increases the variance of the network delay especially in communication-intensive cyber-physical systems (considering the fact that multiple nodes may drop messages onto the network simultaneously and that bursty messages may increase the packet queuing delay on intermediate network devices, which may require retransmission of message by end nodes), we employ a bandwidth sharing policy on end nodes to control the data burstiness in order to bound the worst-case network delay of message transmission. This is essentially a network resource sharing problem. In terms of resource sharing among parallel computation units, past research has focused on shaping the resource access by all units into well defined patterns. These units collaborate to reduce the variance of resource access costs. E.g., MemGuard [25] shapes memory accesses by each core in modern multi-core platforms in order to share memory bandwidth in a controlled way so that the memory access latency is bounded. D-NoC [4] shapes the data transferred by each node on the chip with a (σ, ρ) regulator [3] to provide a guaranteed latency of data transferred to the processor. We have implemented our bandwidth sharing policy based on a traffic control mechanism integrated into Linux to regulate the data dropped onto the network on each distributed node.

To demonstrate the feasibility and capability of our partial-EDF job scheduler, the EDF-based packet scheduler and the bandwidth sharing policy, , we integrated them into our previous real-time distributed hash table (DHT) that only provides probabilistic deadlines [21]. The new real-time distributed system still utilizes a DHT [18] to manage participating storage nodes and to forward messages between these nodes so that data requests are met. However, each participating node employs our partial-EDF job scheduler (instead of the cyclic executive in our previous paper) to schedule data request jobs at the application level. Furthermore, each node employs our EDF-based packet scheduler to send packets in EDF order at the system network level. In addition, all participating nodes follow our bandwidth sharing policy to increase the predictability of network delay. We evaluated our system on a cluster of nodes in a switched network environment.

In summary, the contributions of this work are: (1) We combine periodic message transmission tasks with an EDF scheduler to increase the predictability of real-time distributed systems; we study the impact of period, relative deadline, and WCET of transmission tasks on the EDF job scheduler. (2) We propose an EDF-based packet scheduler running at the operating system level to make the network layer aware of message deadlines. (3) We utilize a Linux traffic control facility to implement our bandwidth sharing policy to decrease the variance of network delays. (4) We implement our comprehensive EDF scheduler and bandwidth sharing policy in a real-time distributed storage system to demonstrate the feasibility and capability of our work.

The rest of the paper is organized as follows. Section II presents the design of our partial-EDF job scheduler, packet scheduler, and traffic control policy. Section III details our real-time distributed storage system implementation. Section IV discusses the evaluation results. Section V contrasts our work with related work. Section VI presents the conclusion and ongoing research.

II. Design

This section first presents the design of our partial earliest deadline first scheduler (partial-EDF) that employs periodic tasks to transmit messages and release jobs, and schedules these jobs in EDF order. It then presents the time-difference table maintained on each node so that the absolute deadlines carried in messages can be adjusted to local clock time. Then, this section presents the design of our EDF packet scheduler and bandwidth sharing policy at the network transport layer.

A. Partial-EDF Job Scheduler

In distributed real-time systems, messages that contain the deadlines of the event are passed among nodes so that the event can be eventually processed. When a message arrives at the operating system network buffer, the scheduler has to acknowledge the arrival of the message so that it can release a corresponding job, which inherits the absolute deadline of the message and can be executed later to process the message. However, when the sender transmits messages periodically, the scheduler on the receiving side cannot guarantee that the release of jobs to process these messages be periodic considering the variance of network delays between the sender and receiver, which causes deviations in message arrival times.

One possible solution is to let the operating system notify the scheduler when a message arrives. For example, a scheduler implemented on a Linux system may utilize a thread to accept messages. This thread is waiting on the network socket until the kernel notifies the thread of message arrivals. The scheduler, once interrupted, releases sporadic jobs to process messages. In this way, the scheduler can process the deadlines of messages immediately when they arrive at the node. Let us call this scheduler *global-EDF* if it executes the job in an EDF order, considering the fact that this scheduler has the knowledge of all deadlines of the corresponding jobs in the system so that it can always execute jobs with the shortest deadlines first. The drawback of the solution is that it is hard to provide a reliable service. Since the operating system may interrupt the scheduler as well as the job executed at a sporadic time, context switches between the scheduler and the interrupt handler may reduce the predictability of the system.

To eradicate the randomness of context switches, we combine the EDF scheduler with a static schedule table, which includes temporal information of a periodic message sending task and a periodic message receiving task. The sending task is to guarantee inter-transmission time of messages has a lower bound. We will describe its details when we present the schedulability test of this EDF scheduler. The receiving task, once executed, accepts messages from the operating system network buffer, processes deadlines contained in these messages, and releases corresponding jobs into the EDF job queue. The receiving task introduces delays between message arrivals and receptions, since messages that have arrived at a node have to wait in the network buffer until the next execution of the transmission task. As a result, the scheduler only has the deadline information of the jobs in the job queue (but no deadline information of the messages waiting in the network buffer). Thus, we name this scheduler the *partial-EDF* job scheduler.

Fig. 1 illustrates the effects of the partial-EDF job scheduler on a node that receives and processes messages of 3 distributed tasks. The green blocks on the first line (τ') represent the execution of the periodic receiving jobs while blocks on the other lines (τ_1, τ_2, τ_3) represent the execution of the corresponding jobs for these tasks, respectively. Four messages arrive at time 2, 3, 6, and 8. Since the corresponding jobs of the messages can only be released the next time a transmission job is executing, jobs $\tau_{1,1}, \tau_{1,2}, \tau_{2,1}, \tau_{3,1}$ are released at time 7, 7, 7, and 13, as shown in the figure. The numbers in the blocks indicate the absolute deadlines of the jobs inherited from messages. As the figure shows, both messages, $\tau_{2,1}$ and $\tau_{3,1}$, have arrived at this node at time 8. However, $\tau_{2,1}$ is executed first even though $\tau_{3,1}$ has a tighter deadline since the message of $\tau_{3,1}$ is still waiting in the network buffer and the partial-EDF scheduler has no knowledge of its arrival at time 8.

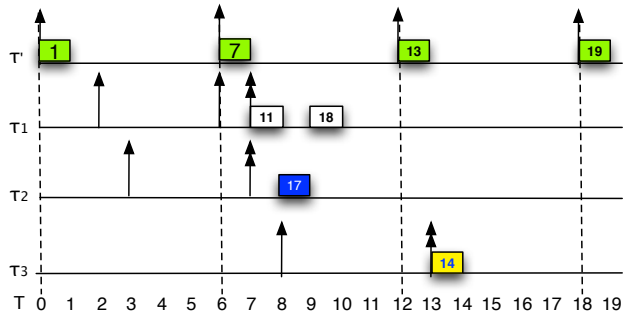

Fig. 1: Partial-EDF Job Scheduler Example (1)

The system density-based schedulability test [14] for global-EDF scheduling is not sufficient for our partial-EDF. As an example, assume the absolute deadline of $\tau_{3,1}$ was 10 in Fig. 1. The global-EDF scheduler can execute all jobs before their deadlines ($\tau_{3,1}$ is scheduled at time 8) as the system density is no more than 100%. However, under partial-EDF, $\tau_{3,1}$ cannot execute before time 10 because of the delay between the message arrival at time 8 and its reception within τ' at time 12.

To derive the schedulability test for our partial-EDF scheduler, we consider a set τ of N tasks that are scheduled in a distributed system of M nodes. A local task $\tau_l = (T_l, C_l, D_l)$ is characterized by its period T_l, worst case execution time (WCET) C_l, and relative deadline D_l. A distributed task $\tau_i = (T_i, C_i, D_i) \xrightarrow{s_i, r_i} (T'_i, C'_i)$ is characterized by a node s_i that releases jobs of the task with period T_i, relative deadline D_i, and WCET C_i of the jobs. The message sending task on node s_i regulates the message traffic of task τ_i so that the minimum inter-transmission time is T'_i, although the EDF scheduler could reduce the time between the executions of consecutive jobs of task τ_i. However, we require that $T_i \geq T'_i$. The receiving task on the receiver r_i releases jobs with WCET C'_i to process these messages. In addition, we use $\tau^s_i = (T^s_i, C^s_i, D^s_i)$ to denote the sending task on the i^{th} node, and we use $\tau^r_i = (T^r_i, C^r_i, D^r_i)$ to denote the receiving task.

Fig. 2 depicts an example that has one distributed task τ_1 and two nodes, (i.e., $N = 1$, $M = 2$). The number in a block represents the absolute deadline of that job. The red blocks on the first line represent the execution of the jobs of τ_1 on the first node in EDF order. $\tau_1 = (3, 1, 21) \xrightarrow{1,2} (3, 1)$ indicates that the message sending task on the first node $\tau^s_1(3, 1, 1)$ guarantees that the inter-transmission time of messages of τ_1 is at least 3. The second receiving job (shown as a block with absolute deadline 11) on the second node $\tau^r_2(5, 1, 1)$ accepts the first three messages of τ_1 and releases three jobs to process them, respectively.

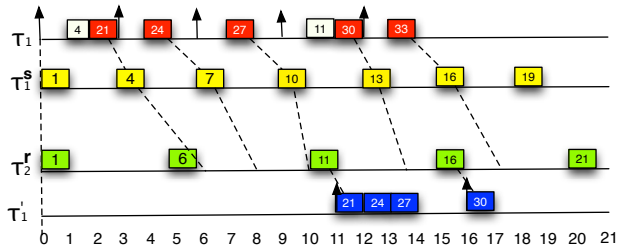

Fig. 2: Partial-EDF Job Scheduler Example (2)

A sufficient schedulability test for our partial-EDF is to prove that a task set always has a system utilization of no more than 100% during any time interval. Let us consider the worst case. Task τ_i generates a flow of periodic messages (period T'_i) from s_i to r_i. Equation 1 expresses the arrival time of the j^{th} message of task τ_i on the receiver r_i, where $\delta_{s_i,r_i,j}$ is the network delay between the sender and receiver when this message is transmitted and O_{s_i,r_i} is the clock time difference between the sender and receiver. Equations 2 and 3 express the constraints of the number of messages k_i of task τ_i that can be queued in the network buffer together with the j^{th} message of the same task before their reception at the receiver r_i (Equation 3 is derived from the left-most terms of Equation 2). In this worst case, messages j, $j+1$, ..., $j+k_i-1$ all arrive at the receiver during $xT^r_{r_i}$, the start time of the x^{th} receiving job on r_i, and $(x+1)T^r_{r_i}$, the start time of the $(x+1)^{th}$ receiving job (assuming messages of the same event are transmitted by the underlying network in FIFO order).

$$A(i,j) = jT'_i + \delta_{s_i,r_i,j} + O_{s_i,r_i} \tag{1}$$

$$xT^r_{r_i} < A(i,j) \leq A(i, j+k_i-1) \leq (x+1)T^r_{r_i} \tag{2}$$

$$x = \lfloor \frac{A(i,j)}{T^r_{r_i}} \rfloor \tag{3}$$

Equation 4 is derived from the right-most terms of Equation 2 substituted with Equation 1. It presents the upper bound

of the number of messages k_i. Let $\delta^-_{s_i,r_i}$ be the best case network delay between the sender and receiver, and $\delta^+_{s_i,r_i}$ be the worst case network delay. Equation 5 presents the number of messages for the i^{th} task that can be queued in the operating system network buffer at the receiver in the worst case, derived from substituting x in Equation 4 according to Equations 3 and 1.

$$k_i \leq \frac{(x+1)T^r_{r_i} - \delta_{s_i,r_i,j+k_i-1} - O_{s_i,r_i}}{T'_i} - j + 1 \quad (4)$$

$$k_i = 1 + \lfloor \frac{T^r_{r_i}}{T'_i} + \frac{\lfloor \frac{jT'_i + \delta^+_{s_i,r_i} + O_{s_i,r_i}}{T^r_{r_i}} \rfloor T^r_{r_i} - jT'_i - \delta^-_{s_i,r_i} - O_{s_i,r_i}}{T'_i} \rfloor$$

$$\leq 1 + \lfloor \frac{T^r_{r_i}}{T'_i} + \frac{\delta^+_{s_i,r_i} - \delta^-_{s_i,r_i}}{T'_i} \rfloor$$

$$with\ \delta_{s_i,r_i,j} \leq \delta^+_{s_i,r_i}\ and\ \delta_{s_i,r_i,j+k_i-1} \geq \delta^-_{s_i,r_i}. \quad (5)$$

The intuition of Equation 5 is that in order to maximize the number of queued messages for an event between two receiving jobs, the first message queued for that event has to arrive with the worst network delay and the last message queued before the receiving job has to arrive with the best network delay.

The first three jobs of τ_1 in Fig. 2 illustrate this worst case. Assume the network delay between the two nodes is in the range $[1, 3]$. The message of the first job $\tau_{1,1}$ arrives after the first receiving job of the second node is invoked because of the large network delay (i.e., 3). After that, $\tau_{1,2}$ and $\tau_{1,3}$ arrive, which results in the worst-case number of message arrivals from τ_1 (i.e., k_1 = 3). Equation 5 with $T^r_{r_i}$ = 5, T'_i = 3, $\delta^+_{s_i,r_i}$ = 3, and $\delta^-_{s_i,r_i}$ = 1 derives exactly this worst case.

When the s^{th} queued message is eventually accepted by the $(x+1)^{th}$ receiving job, the scheduler releases a job to process the message. The relative deadline of the job should be the difference of the absolute deadline carried in the message adjusted by the clock time difference, $(j+s)T_i + D_i + O_{s_i,r_i}$, and the time when the job enters the EDF queue, $(x+1)T^r_{r_i} + C^r_{r_i}$. Equation 6 shows the relative deadline, $D'_{i,s}$, of this job (substitute x from Equations 3 and 1). A negative job deadline indicates that the absolute deadline carried in the message is shorter than the worst-case time ($T^r_{r_i} + C^r_{r_i} + \delta^+_{s_i,r_i}$), i.e., the time required for the partial-EDF on the receiver to release a corresponding job to process the message. In this case, this message cannot be processed before its deadline.

$$D'_{i,s} = ((j+s)T_i + D_i + O_{s_i,r_i}) - ((x+1)T^r_{r_i} + C^r_{r_i})$$
$$\geq ((j+s)T'_i + D_i + O_{s_i,r_i}) - ((x+1)T^r_{r_i} + C^r_{r_i})$$
$$\geq (sT'_i + D_i) - (T^r_{r_i} + C^r_{r_i} + \delta^+_{s_i,r_i}),$$
$$where\ \delta_{s_i,r_i,s} \leq \delta^+_{s_i,r_i}\ and\ 0 \leq s \leq k_i - 1. \quad (6)$$

If these jobs can be executed before the $(x+2)^{th}$ receiving job, the task set is schedulable. Equation 7 defines the total density of periodic tasks on node m (the density of local periodic tasks are included). Equation 8 defines the total density of jobs that process messages sent to node m in this case. $L(i,m)$, $S(i,m)$, and $R(i,m)$ are indicator functions. $L(i,m) = 1$ iff τ_i is a local periodic task on node m. $S(i,m) = 1$ iff node m is the sender of distributed task τ_i. $R(i,m) = 1$ iff node m is the receiver of distributed task τ_i. Otherwise, $L(i,m)$, $S(i,m)$ and $R(i,m)$ are 0. The density test on node m for this case is expressed in Equation 9. This density test has to pass for all M nodes.

$$\rho_m = \sum_{i=1}^{N} (L(i,m) + S(i,m)) \frac{C_i}{min\{T_i, D_i\}} + (\frac{C^r_m}{T^r_m} + \frac{C^s_m}{T^s_m})$$

$$L(i,m) = \begin{cases} 1, & if\ \tau_i\ is\ local\ on\ m, \\ 0, & otherwise. \end{cases}$$

$$S(i,m) = \begin{cases} 1, & if s_i = m, \\ 0, & otherwise. \end{cases} \quad (7)$$

$$\rho'_m = \sum_{i=1}^{N} R(i,m) \sum_{s=0}^{k_i-1} \frac{C'_i}{min\{T^r_m, D'_{i,s}\}}$$

$$R(i,m) = \begin{cases} 1, & if\ r_i = m, \\ 0, & otherwise. \end{cases} \quad (8)$$

$$\rho_m + \rho'_m \leq 1 \quad (9)$$

Our schedulability test accuracy depends on the variance of network delays as shown in Equation 5. We will introduce our bandwidth sharing policy in Section II-C, which reduces contention and network delay variance. In addition, absolute deadlines carried in messages have to be adjusted at the receiver to compensate for pairwise time differences across nodes. Otherwise, our partial-EDF cannot schedule corresponding jobs in EDF order. We will introduce the time-difference table maintained on each node in Section II-B to address this problem.

B. Time-difference Table

Our scheduler depends on the task deadlines carried in the messages to schedule the jobs to process these messages in EDF order. Since the messages received on a node could come from different senders and the absolute deadlines carried in these messages are derived from the local clock of the senders (i.e., different reference frames), the receiving node must have adequate knowledge of the time difference to maintain the EDF order of job executions. Global clock synchronization could solve this problem but is costly [11]. Instead of clock synchronization, our scheduler maintains a time-difference table on each node. This table contains the difference between the clocks on nodes that send messages to each other. When one node receives messages from another node, it uses the time-difference table to adjust the absolute deadlines carried in the messages, so that the deadlines in all messages are adjusted to the local clock of this receiver.

We utilize the algorithm of the Network Time Protocol (NTP) [17] to derive the time difference between each pair of sender and receiver. We assume the sender s and receiver r of a task τ to be known a priori. In our system, the receiver r transmits a synchronization message to s and s transmits a feedback message to r. On the basis of the timestamp exchanged via this pair of messages, the receiver r can derive the round trip delay $\delta_{s,r}$ and clock offset $O'_{s,r}$ The constraint of the true time offset for the sender relative to the receiver, $O_{s,r}$, is expressed in Equation 10. Following the NTP design, the data-filtering algorithm is performed on multiple pairs of $(\delta_{s,r}, O'_{s,r})$ collected from timestamps exchanges to derive the three time offset. Each node in our distributed system has a

periodic task to transmit time synchronization messages and to calculate its local time-difference table dynamically.

$$\delta_{s,r} - \frac{O'_{s,r}}{2} \leq O_{s,r} \leq \delta_{s,r} + \frac{O'_{s,r}}{2} \tag{10}$$

With the time-difference table, the absolute deadline carried in messages can be adjusted to the local time on the receiver (i.e., $D'_r = D_s + O_{s,r}$).

C. EDF Packet Scheduler and Bandwidth Sharing

We utilize a traffic control mechanism provided by Linux to regulate the message traffic that is transmitted via the network in order to reduce network contention and variance of network delay. By limiting the bandwidth per node and by reducing the burstiness of the traffic, network contention can be reduced. In addition, we extend Linux traffic control by implementing a deadline-driven queue discipline, the *EDF packet scheduler*, to compensate for the shortcoming of our partial-EDF scheduler that it may transmit the messages for tasks with longer deadlines earlier than the messages for tasks with shorter deadlines.

Let us assume that each node in the distributed system is connected by a single full-duplex physical link to a switch and the network interface works in work-conserving mode. A node transmits messages to other nodes via the same link. We configure the network interface to use the Hierarchical Token Bucket (HTB) [7] algorithm to transmit messages. HTB has two buckets: The *real-time bucket* is utilized to transmit packets of messages that serve real-time tasks and the *background traffic bucket* is utilized to transmit background traffic. This design requires that the IPs and ports of receiving nodes for real-time tasks are known a priori, so that their messages can be differentiated from other traffic and put into the real-time bucket. We configure HTB to have a (σ, ρ) regulator [3] for each bucket. This regulator ensures that the total size of packets transmitted from that bucket during the time interval $[s, e]$ is bounded by $\sigma + \rho * (s - e)$, where σ determines the burstiness of the traffic, and ρ determines the upper bound of the long-term average rate.

In order to configure HTB so that it benefits real-time systems, we propose several policies (1) to prioritize buckets, (2) to schedule packets in the real-time bucket, and (3) to share bandwidth among nodes. First, the real-time bucket has a higher priority than that of the background traffic bucket. Task packets are transmitted before background traffic packets when both buckets are legitimate to send packets according to the (σ, ρ) rule.

Second, the real-time bucket employs an EDF packet scheduler to schedule packets. Since the Linux traffic control layer does not understand task deadlines embedded in data messages, which are encapsulated by the application layer, we extend the data structure for IP packets in Linux by adding new fields to store timestamps and extend the *setsockopt* system call by supplying these timestamps. These extensions provide the capability of specifying message deadlines for real-time tasks (applications). Then, our EDF packet scheduler utilizes the message deadlines to transmit packets in EDF order. By employing the EDF packet scheduler, we compensate for shortcomings of our partial-EDF job scheduler. As mentioned in Section II-A, because of the delay between message arrivals and receptions introduced by the receiving task in our partial-EDF scheduler, messages that arrive at a node but have not been accepted into the job queue may be processed later than the jobs already in the queue, even if these messages have shorter deadlines. As a result, messages carrying short deadlines are put into the real-time bucket later than packets with longer deadlines. By considering the deadlines carried in the messages at the transport layer, our EDF packet scheduler compensates for this shortcoming by rescheduling the messages. In contrast, the background traffic bucket simply employs a FIFO queue to transmit background traffic.

Third, we consider the underlying network utilized by nodes as a whole system and propose two strategies of bandwidth sharing among nodes. Let w_m be the proportion of bandwidth for node m and M be the number of nodes. The first strategy is fair-share (i.e., $w_m = \frac{1}{M}$). This strategy is simple to understand but does not consider the different transmission demands of nodes. Consider a scenario where two periodic tasks on node 1 transmit messages to node 2, of which only one periodic task transmits messages back to node 1. Node 1 could benefit from a larger fraction of bandwidth since it has a larger transmission demand. Thus, the second strategy shares bandwidth proportionally considering the number of tasks on each node and the periods of these tasks. Let us assume the task sets on nodes were known a priori so that the denominator in Equation 11 represents the aggregate of transmission demands on all nodes and the numerator the transmission demand of node m.

$$w_m = \frac{\sum_{i=1}^{N} S(i,m)\frac{1}{T'_i}}{\sum_{i=1}^{N} \frac{1}{T'_i}} \tag{11}$$

The experimental evaluation results in Section IV show that our EDF packet scheduler can decrease the deadline miss rate of messages significantly and the bandwidth sharing policy can decrease the network delay deviation in a local cluster of nodes. In Section III, we will elaborate on the implementation of our real-time distributed storage system, which employs the hybrid EDF scheduler to schedule jobs and packets.

III. Real-Time Distributed Storage System

In this section, we summarize our prior contributions [21], i.e., we present the features of our real-time distributed storage system and its message routing algorithm, which provides the basis of our novel contributions in this work. We then present our integration of a hybrid EDF scheduler to the RT-DHT.

A. DHT

As a storage system, our RT-DHT provides *put(key, data)* and *get(key)* API services to upper layer applications. This RT-DHT manages a group of storage nodes connected over the networks and implements a consistent hashing algorithm [10] to map a key onto a node that stores the data. When a *put* or *get* request is sent to any node in the system, a *lookup(key)* message is forwarded among these nodes following a particular forwarding path until it arrives at the node to which this key is mapped.

We adopt the Chord algorithm [18] to determine the forwarding paths for requests. Chord organizes storage nodes in a ring-shaped network overlay [18], [10]. It uses a base

hash function, such as SHA-1, to generate an identifier for each node by hashing the node's information, e.g., IP address and port. It also uses the same hash function to generate an identifier for the key of a request. The purpose of using the same hash function is to generate the identifiers of nodes and keys in the same domain in which ordering is defined. If node A is the first node after node B clockwise along the ring, A is called the predecessor of B, and B is called the successor of A. Similarly, a key also has a successor node. For example, Fig. 3 depicts 9 storage nodes (green) that are mapped onto the Chord ring (labels in squares are their identifiers), which has 5-bit identifiers for nodes and keys. *N14* is the successor of *N12* and the predecessor of *N20*. The successor node of *K14* is *N15*. The Chord algorithm defines the successor node of a key as its destination node, which stores the corresponding data of the key.

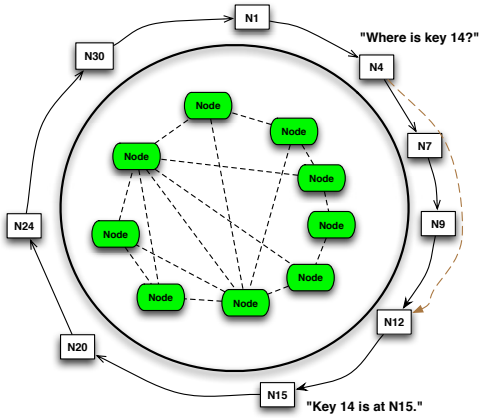

Fig. 3: Chord ring example

Chord maintains a so-called finger table on each node, which acts as a soft routing table to decide the next node to forward lookup messages to. Each entry in that table is a tuple of (start, interval, forward node), as shown in Table I of *N4*'s finger table. Each entry in the table represents one routing rule: the next hop for a given key is the forward node in that entry if the entry interval includes the key. For example, the next hop for *K14* is *N12* as *14* is in *[12, 20)*. In general, for a node with identifier k, its finger table has $logL$ entries, where L is the number of bits of identifiers generated by the hash function. The start of the i^{th} finger table entry is $k + 2^{i-1}$, the interval is $[k + 2^{i-1}, k + 2^i)$, and the forward node is the successor node of key $k + 2^{i-1}$. Because of the way a finger table is constructed, each message forwarding reduces the distance to its destination node to at least half that of the previous distance on the ring. The number of intermediate nodes per request is at most $logM$ with high probability [18], where M is the number of nodes in the system.

Chord provides high efficiency and deterministic message forwarding in the stable state (i.e., no node failures, node joins, and network failures). This is important to our RT-DHT since based on this property, we can apply the bandwidth sharing policy and perform a schedulability test on a given task set to determine whether the deadlines of the requests in the task set can be met by our partial-EDF scheduler.

B. Hybrid EDF Scheduler Integration

We have implemented our RT-DHT in Linux. Each storage node in the system employs a partial-EDF job scheduler using a single thread. As described in Section II-A, the static schedule table includes the temporal information of periodic tasks. Based on this temporal information, the scheduler sets up a periodic timer. When the signal handler (at the application level) is triggered by the timer interrupt, the scheduler, by executing the handler, releases the jobs of periodic tasks into a job queue according to the table. After that, the scheduler executes the jobs waiting in the queue in EDF order.

TABLE I: Finger table for *N4*

#	start	interval	forward node
1	5	[5, 6)	N7
2	6	[6, 8)	N7
3	8	[8, 12)	N9
4	12	[12, 20)	N12
5	20	[20, 4)	N20

This work extends the Linux system by providing support for the RT-DHT storage node for message deadlines and for transmitting messages in EDF order (see Section II-C). Below are the most significant changes we made to Linux to support this functionality.

(1) We added a new field, *deadline*, of type *ktime_t* in the kernel data structure *sock* so that deadline of a socket provided by the application can be stored.

(2) We extended the kernel function *sock_setsockopt* and added a new option $SO\_DEADLINE$ so that the application can utilize $setsockopt$ (i.e., the corresponding user mode function) to configure the message deadlines when the application attempts to transmit messages. *sock_setsockopt* keeps the value of the deadline in the *deadline* field of the *sock* structure.

(3) When the application transmits a message, the kernel creates instance(s) of the *sk_buff* structure to store the data of the message. We added a new field, *deadline*, in *sk_buff* so that the *deadline* of a socket can be saved when the kernel creates the data structure. After this, the deadline of the message in *sk_buff* is passed down to the transport layer.

(4) We implemented an EDF packet scheduler, which provides the standard interfaces of Linux traffic control queue disciplines [7]. The EDF packet scheduler utilizes a prioritized queue to maintain the instances of *sk_buff* in a min-heap data structure. We utilize the *cb* field, the control buffer in *sk_buff*, to implement a linked list-based min-heap. This linked list-based implementation does not have a limit on the number of messages that can be queued in the min-heap, which an array-based implementation of min-heap would have.

We combine Linux HTB with our EDF packet scheduler and associate the hierarchical queue with the network interface that connects the storage node with the DHT overlay as described in Section II-C. Since the EDF packet scheduler implements the standard interfaces of queue disciplines, the storage system can utilize the Linux traffic control command (i.e., *tc*) to configure the queue when the system boots up.

Section II-A presented the schedulability test for the partial-EDF job scheduler. However, more than two storage nodes may be involved to serve the same data request of a distributed task in the RT-DHT. For example, to serve the request that looks up the data for key 15 on $N4$ in Fig. 3, $N4$ first executes a job to transmit the request message to $N12$ following the finger table. Then, $N12$ executes a corresponding

job to process the message and transmits another message to $N15$ via its sending task. $N15$ obtains the data from its storage and transmits a feedback message back to $N4$. We adjust the task set in the schedulability test to accommodate this as follows.

Let us use τ to represent the original task set, in which the message forwarding path for τ_i is $P_i = (n_1, n_2, ..., n_{m_i})$, where m_i is the length of the path. Let us denote $\tau_i = (T_i, C_i, D_i) \xrightarrow{n_1,n_2} (T_i', C_i')$ to represent task τ_i that initializes data requests on node n_1 periodically. We first replace τ_i with τ_i^1 as shown in Equation 12. The partial-EDF scheduler on n_1 schedules τ_i^1 with its new absolute deadline $\frac{D_i}{m_i}$. We share the end-to-end deadline D_i evenly among m_i nodes to simplify the representation of our schedulability test model. Past work has provided sophisticated protocols for the subtask deadline assignment problem (SDA) [9], [22], [23], which can be adopted by our schedulability model. When the receiving job on n_2 releases the corresponding job to process the message of τ_i, it increases the deadline of the job by $\frac{D_i}{m_i}$. Replacing τ_i by τ_i^1 only changes Equation 7 in Section II-A if $m_i = 2$ (i.e., $D_i \rightarrow \frac{D_i}{2}$).

$$\tau_i^1 = (T_i, C_i, \frac{D_i}{m_i}) \xrightarrow{n_1,n_2} (T_i', C_i') \tag{12}$$

On node n_2, the jobs created by τ_i^1 transmit messages to n_3. To reflect this new task on n_3 in the schedulability test, we define a virtual task $\tau_i^2 = (*, 0, \frac{D_i}{m_i}) \xrightarrow{n_2,n_3} (T_i', C_i')$. The density of the jobs (on sender n_2) of τ_i^2 is already considered in Equation 8 since they are the very jobs that process the received messages of τ_i^1. The density of the jobs (on receiver n_3) of τ_i^2 is considered in Equation 8 for node n_3. This task is virtual since the scheduler on n_2 does not actually schedule any jobs of the task. We set its WCET to 0 so that it can be integrated into Equation 7. As indicated in Equations 4 to 8, the density of the jobs on receiver n_3 does not depend on the period of τ_i^2. Thus, the period of τ_i^2 is set to *. In general, Equation 13 expresses the virtual task τ_i^j for each node n_j on the path.

$$\tau_i^j = (*, 0, \frac{D_i}{m_i}) \xrightarrow{n_j,n_{j+1}} (T_i', C_i'), where\ 2 \leq j < m_i. \tag{13}$$

Finally, we perform the schedulability test on the new task set, which is the union of τ and all virtual task, with Equations 5 - 9 by adjusting Equation 6 ($D_i \rightarrow \frac{2D_i}{m_i}$) and Equation 7 ($D_i \rightarrow \frac{D_i}{m_i}$).

To simplify the presentation of these equations, we assume that the jobs to process messages of the same task have the same WCET even when they are executed on different storage nodes (i.e., C_i' is inherited by the virtual task). However, the schedulability test does not depend on inheritance of WCETs.

IV. Experimental Results and Analysis

In this section, we evaluate our hybrid EDF scheduler on a local cluster. The experiments are conducted on 8 nodes, each of which features a 2-way SMP with AMD Opteron 6128 (Magny Core) processors and 8 cores per socket (16 cores per node). Each node has 32GB DRAM and Gigabit Ethernet (utilized in this study). These nodes are connected via a single network switch. Each node runs a modified version of Linux 2.6.32, which includes the EDF packet scheduler.

In the first experiment, we evaluate the contribution of the bandwidth sharing policy in terms of decreases in network delay variance. Then, we evaluate the impact on task deadline miss rate when employing our EDF packet scheduler. Next, we evaluate the partial-EDF job scheduler by running different task sets on the RT-DHT.

A. Bandwidth Sharing Results

In this experiment, we use the network benchmark, Sockperf, to measure the application-to-application network delays in ping-pong mode. We utilize 8 nodes, each of which runs a Sockperf server and a client. Each Sockperf client periodically transmits messages as TCP packets to the servers and a server sends a feedback packet back to the client immediately. The clients calculate network delays based on the time of sending the message and receiving the feedback. Sockperf clients can only transmit a fixed number of messages (specified with *burst* option) within a period of time. However, a real-time application could have different message transmission demands in different frames. We extend the Sockperf implementation to transmit varying numbers of messages during different time periods. In our experiment, the varying numbers are drawn round robin from a population of bursts derived from *burst* option, and the destination (server) of the message is chosen from the 8 servers in a round robin fashion. As a result, the number of messages transmitted (i.e., burstiness) in different frames can be different. We compare the network delay variance with and without bandwidth sharing policy. For the experiments of bandwidth sharing, we configure the network interface of server and client to send messages into two token buckets: one for the client to transmit messages and the other one for the server to transmit feedback messages. A third token bucket is added to transmit unclassified (background) traffic including the packets of other applications running on the same node.

Fig. 4 depicts the network delay interval on one node (x-axis) over the network delay (round-trip delay) in milliseconds (y-axis). The blue clusters depict the network delays for the bandwidth sharing policy. The interval of blue clusters extends from 0.14 to 3.69 (i.e., a range of 3.55) altogether, which is larger than 0.97, the range without bandwidth sharing policy. However, the large variance is not only due to network delays. Since the token bucket implementation of Linux is driven by kernel timer events, once a message is pending in the queue (i.e., not transmitted by the network interface immediately due to the bandwidth sharing policy), the message has to wait at least one timer interrupt interval until the next time the network interface considers to transmit messages. The timer interrupt interval is $1ms$ (i.e., HZ = 1000). Considering the cost of the token bucket implementation, we divide the measured application-to-application delays into different clusters and calculate the network delay variance in each cluster. As depicted in Fig. 4, each blue cluster represents a cluster of network delays with the variance marked on the x-axis. The range of all blue clusters is less than the 0.97 range of the red cluster, which illustrates the benefit of our bandwidth sharing policy.

Fig. 5 depicts the network delays for all 8 nodes after the token bucket costs are removed in the measured data. We observe that the bandwidth sharing policy reduces network delay variance on all nodes. The cost of bandwidth sharing

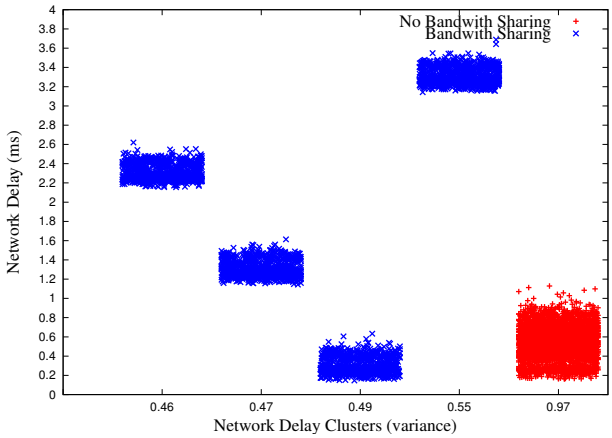

Fig. 4: Network delay comparison (1) for Delay Clusters (ms)

contributes significantly to application-to-application network delay since we utilize a Gigabit network switch and NICs in our experiment. However, we believe that the proportion of the cost will reduce in complex network topologies, where cumulative network queuing delays on intermediate network devices could change significantly without bandwidth sharing policy, i.e., our reference without bandwidth sharing in the experiments is the best case in a distributed system.

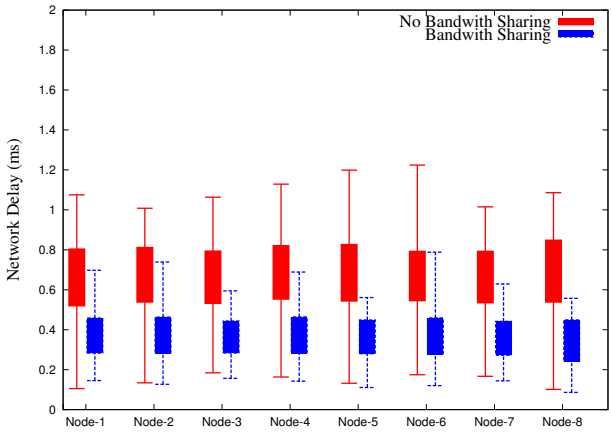

Fig. 5: Network delay comparison (2)

The challenge of applying bandwidth sharing is to determine the limit of the packet transmission rate according to the transmission demand of the applications and the transmission capability of the underlying network. If the transmission rate is too large, packets are transmitted into the network immediately and network contention could be significant resulting in large network delay variance. If the rate is too small, packets could wait in the bucket for a long time resulting in large application-to-application delays. In Section II-C, we proposed to share the total rate proportionally according to the demand of an individual node. However, we did not provide a theoretic method to determine the total rate. In the experiment for Fig. 4 and 5, we configure Sockperf to simulate a $10ms$ frame size (set option mps to 100) and in each frame, the number of messages are chosen from $[5, 10]$ in a round robin fashion (set option $burst$ to 10 for an the interval of $[\frac{burst}{2}, burst]$). The message size is 400 bytes. Then, we tune the rate experimentally and select $2.45Mbps$ as the client transmission rate and $19.60Mbps$ as the server transmission rate.

B. EDF Packet Scheduler Results

In this section, we extend the network benchmark, Sockperf, so that messages sent by Sockperf clients carry deadline information. Sockperf clients specify the deadline (by using the $setsockopt$ system call) before sending a message and Sockperf servers immediately send a feedback message with a specified message deadline. The feedback message inherits the deadline of the original message sent by the client. A Sockperf client compares the arrival time of a feedback message and the deadline carried in that message to determine if this message has met its deadline. In each experiment, we first run Sockperf without the EDF packet scheduler on 8 nodes (one client and one server on each node), then we run Sockperf in playback mode, which transmits the same sequence of messages with the EDF packet scheduler installed on the same nodes. We compare their deadline miss rates.

The workload is controlled by Sockperf options $(mps, burst)$. The number of messages transmitted in a frame by a client is chosen from an interval $[\frac{burst}{2}, burst]$ in a round robin fashion. Each message is 400 bytes. In addition, the deadline of a message is the current time when the message is created, increased by a random value uniformly drawn from the interval $[D, 2D]$, where D is a configurable parameter in different experiments.

Fig. 6 depicts the deadline miss rates on one node under different workloads and deadline setups. Line *0.8-NoEDF* depicts the change of deadline miss rates when the workload changes from $mps = 100, burst = 40$ to $mps = 5000, burst = 200$. D is $0.8ms$, and no EDF packet scheduler is attached to the network interface of the nodes. The line indicates that before workload $(4400, 40)$, the deadline miss rate is stable (about 5%). After that, the deadline miss rate increases significantly when the workload increases. This is explained as follows: When the workload is small, the network interface can transmit messages immediately into the network. In this case, the deadline misses are caused by the network delay. When the workload increases to large values (i.e., larger than $(4400, 40)$ in our experiment), message packets start to be queued in the network buffer, which increases their transmission time. Thus, deadline misses increase significantly. In addition, the probability of a message being queued depends on the burst size, since a larger burst size suggests that the Sockperf client attempts to transmit more messages in a frame.

As a comparison, line *0.8-EDF* depicts the change of deadline miss rates under the same setup, except that each node adopts an EDF packet scheduler. We observe a similar trend of deadline miss rates that change with the workload as for line *0.8-NoEDF*. However, due to the capability of re-scheduling packets, the EDF packet scheduler decreases the deadline miss rates when the network interface is dominated by bursty data. Lines *1.0-NoEDF* and *1.0-EDF* depict the comparison of deadline miss rates when D is $1ms$, which suggest a similar trend as that of $D = 0.8ms$.

C. Partial-EDF Job Scheduler Results

We next experiment with our RT-DHT storage system to measure the value of k_i defined in Equation 5, which is the worst number of messages that can arrive at one node for task

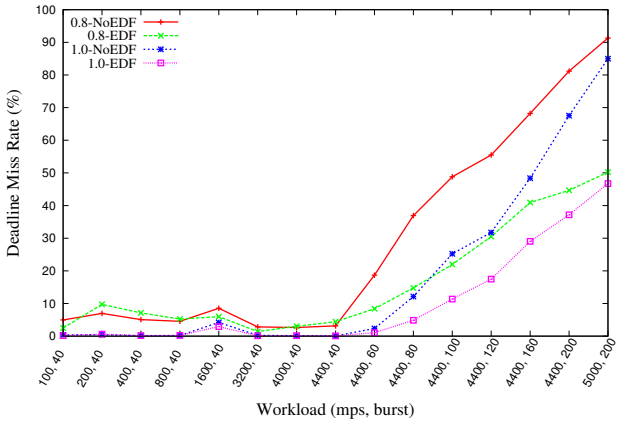

Fig. 6: Deadline miss rate comparison

τ_i, since k_i is the basis of the schedulability test of the partial-EDF scheduler. Then, we compare the measurement with the value calculated by our formula. To simplify the analysis, we predefine the identifiers of nodes and keys on the chord ring instead of using a hash function to calculate these identifiers. We utilize 8 storage nodes in this experiment.

In the first experiment, we run a single distributed put-request task $\tau_1 = (10, 1, 100) \xrightarrow{0,1} (5, 1)$ on node 0, which only involves nodes 0 and 1 (all times are expressed in ms). The partial EDF scheduler on node 0 releases jobs of τ_1 with a period of $10ms$ and the sending task guarantees that the inter-transmission time for the message of τ_1 is at least $5ms$. The frame size of node 0 is $5ms$ (we require that $T_0 \geq T_0'$). In addition, a local periodic task $\tau_2 = (60, 50, 60)$ is scheduled on node 0. We busy wait to make the real execution time of a task match its WCET. The receiving task on node 1 has a period of $10ms$. We observe that the worst-case number of message arrivals on node 1 is 3, which matches the value calculated by Equation 5 (i.e., $k_1 = 1 + \lfloor \frac{10}{5} + \frac{\delta^+ - \delta^-}{5} \rfloor$), since both δ^+ and δ^- are small in our local cluster compared to the message sending period ($5ms$). However, we cannot simply decrease the period of τ_1 to the level of network delay considering the minimal time slices of Linux ($1ms$ in our experiment). Instead, we implement a traffic control utility to simulate large network delays. We add a $10 \pm 5ms$ network delay to node 0, which results in a delay range of $[5.25ms, 15.89ms]$. (The delay range is measured via the ping-pong mode of Sockperf. It changes over time but the changes are in $\pm 0.5ms$). With this setup, the worst-case number of message arrivals is 4 (i.e., smaller than the upper bound 5 calculated from Equation 5).

Then, we update τ_1 so it releases *put* requests that require nodes 0, 3 and 4. In this case, the message of τ_1 is first forwarded to node 3 and from there forwarded to node 4. Both nodes 3 and 4 have a $10ms$ message receiving period and a $5ms$ message sending period. We observe the same worst-case number of message arrivals, i.e., 4.

However, the temporal properties of the receiving task in our partial-EDF scheduler have to be tuned to suit different workloads. For example, the WCET of the receiving task determines the most number of messages it can copy from the kernel. In one experiment, each node has four distributed tasks and each of them involves three nodes. The periods of these tasks are $40ms$ each. The receiving task of $1ms$ WCET cannot always copy all messages from the kernel and release the corresponding jobs to process these messages. This leaves as an open question: How should one theoretically determine the proper WCET of the receiving task as well as the sending task?

V. Related work

Past research has put forth methodologies for adjusting the interrupt-handling mechanisms of operating systems to provide predictable interrupt-handling mechanisms for real-time application tasks [26], [5], [13]. These approaches improves the predictability of the kernel-level interrupt handling, while our partial-EDF job scheduler focuses on improving the predictability of the interrupt handling at user level. Compared to other structured user-level real-time schedulers that provide predictable interrupt management [19], our partial-EDF job scheduler simply adopts a static scheduling mechanism (i.e., cyclic executive [2]) with an EDF scheduler. This simplification allows us to use a density-based schedulability test on a task set by refining the temporal properties of tasks in the table (for example, the timer interrupt handler, the message sending task, and the message receiving task in our system). Thus, these tasks can be modeled as periodic tasks in the density-based schedulability test. This also allows us to perform a schedulability test on a set of distributed tasks, in which we consider the temporal dependency among tasks on different distributed nodes.

Resource sharing significantly affects the predictability of real-time systems where multiple real-time tasks may attempt to access the shared resource simultaneously. Past work has focused on shaping the resource access patterns to provide predictable response times to concurrent real-time tasks [25], [4], [24], [12]. For example, bursty memory accesses were delayed so that the number of memory accesses by a task meets a bound during a period of time. Shaping is also utilized to improve the schedulability of fixed-priority real-time distributed systems that contain periodic tasks with a release jitter. [20]. We apply a similar methodology, which utilizes Linux traffic control mechanisms to shape the out-going traffic on distributed nodes in order to improve the predictability of the network delay (i.e., a decrease in variance of network delay). In contrast to past work [20], our experimental results have shown the benefit of applying shaping, which decreases the network delay variance by reducing network congestion in general in distributed systems, where tasks not only experience release jitter, but the number of tasks can also vary over time. However, these aforementioned resource-sharing mechanisms are *passive* since they only reduce the probability of resource contention instead of eradicating contention altogether by only controlling the resource access of end nodes. Real-time tasks may benefit more from *active* resource sharing mechanisms, which perform access control via resource controllers [1] or real-time state-ful communication channels [8]. These mechanisms require intermediate devices (e.g., network routers) to cooperate. Today, software-defined networks (SDN) [16] allow intermediate network devices (i.e., SDN capable switches and routers) to control traffic flows explicitly, which may reduce the network delay variance even for complex network topologies.

We implement an EDF packet scheduler that transmits

packets in EDF order at end nodes. Previous research has also proposed to transmit real-time traffic over entire networks [6]. However, this requires modifications to the IP packets and end nodes to establish stateful communication channels before transmitting data. In addition, network devices have to maintain channel states and be aware of the deadlines carried in IP packets.

VI. Conclusion

We have presented a hybrid EDF scheduler for distributed real-time systems. At the application level, our partial-EDF job scheduler utilizes periodic tasks to transmit messages over networks to decrease the impact of application-level interrupt handling on system predictability, while the scheduler schedules tasks in EDF order. We have proposed a density-based schedulability test on a task set for our partial-EDF job scheduler. In the transport layer, we extend the Linux network stack to support message deadlines and implement an EDF packet scheduler to transmit messages in EDF order. We also propose to utilize Linux traffic control mechanisms to decrease network delay variance, which may increase the predictability of distributed tasks. Our experimental results showed that the EDF packet scheduler can decrease the deadline miss rate and the traffic control mechanism can decrease network delay variance in a local cluster. In addition, we demonstrated the effectiveness of these techniques by integrating them into our RT-DHT storage system so that data requests can be served in EDF order.

Future work includes experimenting the traffic control mechanism in complex network topologies and performing fine-grained network resource control, for example, adopting software-defined network techniques to prioritize messages on intermediate network devices according to message deadlines. In doing so, we wish to study the impact of our hybrid EDF scheduler in a large-scale environment, where more nodes could process real-time distributed tasks simultaneously. In addition, we wish to find a better way to evaluate our partial-EDF job scheduler.

Acknowledgment

This work was supported in part by NSF grants 1329780, 1239246, 0905181 and 0958311. Aranya Chakrabortty helped to scope the problem in discussions.

References

[1] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256. ACM, 2007.

[2] T.P. Baker and Alan Shaw. The cyclic executive model and Ada. Technical Report 88-04-07, University of Washington, Department of Computer Science, Seattle, Washington, 1988.

[3] Rene L Cruz. A calculus for network delay. i. network elements in isolation. *Information Theory, IEEE Transactions on*, 37(1):114–131, 1991.

[4] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhies, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[5] Tullio Facchinetti, Giorgio Buttazzo, Mauro Marinoni, and Giacomo Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 98–105. IEEE, 2005.

[6] Hoai Hoang, Magnus Jonsson, Anders Kallerdahl, and Ulrik Hagström. Switched real-time ethernet with earliest deadline first scheduling-protocols and traffic handling. *Parallel and Distributed Computing Practices*, 5(1):105–115, 2002.

[7] Bert Hubert, Thomas Graf, Greg Maxwell, Remco van Mook, Martijn van Oosterhout, P Schroeder, Jasper Spaans, and Pedro Larroy. Linux advanced routing & traffic control. In *Ottawa Linux Symposium*, page 213, 2002.

[8] DD Kandhlur, Kang G Shin, and Domenico Ferrari. Real-time communication in multihop networks. *Parallel and Distributed Systems, IEEE Transactions on*, 5(10):1044–1056, 1994.

[9] Ben Kao and Hector Garcia-Molina. Deadline assignment in a distributed soft real-time system. *Parallel and Distributed Systems, IEEE Transactions on*, 8(12):1268–1274, 1997.

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[11] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, 100(8):933–940, 1987.

[12] S. K. Kweon and K. G. Shin. Achieving real-time communication over ethernet with adaptive traffic smoothing. In *in Proceedings of RTAS 2000*, pages 90–100, 2000.

[13] Luis E Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional pc hardware. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 14–23. IEEE, 2006.

[14] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[15] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[16] N. McKeown. Software-defined networking. INFOCOM keynote talk, 2009.

[17] David L Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[18] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[19] Gabriel Parmer and Richard West. Predictable interrupt management and scheduling in the composite component-based system. In *Real-Time Systems Symposium, 2008*, pages 232–243. IEEE, 2008.

[20] Linh TX Phan and Insup Lee. Improving schedulability of fixed-priority real-time systems using shapers. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 217–226. IEEE, 2013.

[21] T. Qian, F. Mueller, and Y. Xin. A real-time distributed hash table. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

[22] Juan Rivas, J Gutierrez, J Palencia, and M Gonzalez Harbour. Deadline assignment in edf schedulers for real-time distributed systems.

[23] Juan M Rivas, J Javier Gutiérrez, J Carlos Palencia, and M González Harbour. Optimized deadline assignment for tasks and messages in distributed real-time systems. In *Proceedings of the 8th International Conference on Embedded Systems and Applications, ESA*. Citeseer, 2010.

[24] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 49–60. IEEE, 2007.

[25] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.

[26] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *RTSS*, volume 6, pages 191–201, 2006.