# Scheduling Real-Time Systems with End-to-End Timing Constraints Using the Distributed Pinwheel Model *

Chih-wen Hsueh and Kwei-Jay Lin†

*Department of Information and Computer Science*

*†and Department of Electrical and Computer Engineering*

*University of California, Irvine*

*Irvine, CA, 92697*

March 10, 1997

## Abstract

Real-time distributed applications have timing constraints on tasks running on several processors. To design real-time systems with end-to-end performance requirements, we need to have algorithms to schedule and to coordinate tasks on different processor nodes. In this paper, an end-to-end scheduling approach based on the pinwheel scheduling model is presented for distributed real-time systems. We show how tasks on different nodes may be transformed to have periods consisting of only harmonic numbers. With harmonic periods, we can use a polynomial-time algorithm to find the start and finish times of each task on each node. Phase alignment algorithms are then applied to adjust the phases between schedules of neighboring nodes so that the overall end-to-end delay is minimized. Using the distributed pinwheel model, schedules on different nodes are synchronized to lessen the delays. For many real-time systems, this approach provides a predictable performance and a short end-to-end delay.

*This work was supported in part by contracts from the Office of Naval Research N00014-94-1-0034 and N00014-95-1-0262, Hughes Aircraft/UC MICRO 94-082 and 95-097, and the US Navy NSWCDD N60921-94-M-1261 and N60921-94-M-2714.

# 1 Introduction

Many real-time systems have complex configurations including multiprocessors, sensors and actuators. They also maintain a large set of fast-changing and short-lived data. One way to implement such systems is to divide each real-time computation into a pipeline of tasks, each task running on a separate processor in a distributed system. We call such a task sequence a *transaction* in this paper. For example, in many military avionics applications, a multi-sensor situation assessment task is invoked periodically to evaluate new threats. The assessment may have one or more input processing tasks collecting data from different sensor devices. These data are then sent to one or more processing and communication tasks so that the assessment task can decide if the response to a certain threat is necessary. The end-to-end deadline between a sensor event and the corresponding decision response is defined to be between 200 and 500 ms. Similar requirements exist in a flight control task which collects device data to detect any hardware/software fault, and must reconfigure the system within an end-to-end deadline.

Algorithms for allocating resources and scheduling tasks are important to the success of real-time systems with end-to-end performance requirements. A good algorithm will improve the schedulability, dependability and scalability of systems. Previous work on the problem has studied different algorithms using the priority-driven approach [1, 17, 18]. In this paper, an end-to-end scheduling approach based on the pinwheel scheduling algorithms [2, 11] is presented for distributed real-time systems. We extend one of the most powerful single-node pinwheel algorithms, **Sr** [7], to the **DSr** algorithm so that all nodes are processed at the same time. We then present a polynomial-time algorithm to produce the single-node schedule for each node after it is processed by **DSr** and consists of only harmonic distance constraints. After the schedules for individual nodes are produced, we then provide algorithms to adjust the phases (relative start time) between these nodes so that the overall end-to-end delay for transactions is minimized or reduced.

Our contribution in this paper can be summarized as follows:

1. We have designed a polynomial time algorithm to construct the single-node pinwheel schedule. No such algorithm has been published previously.

2. We have extended **Sr** [7] to **DSr** so that tasks on multiple nodes are transformed at the same time. We have also found a tight schedulability condition for **DSr**.

3. An algorithm for minimizing the total end-to-end delay has been presented. The problem of

minimizing the maximum end-to-end delay has also been studied.

Using the pinwheel approach, schedules on different nodes are better synchronized and the end-to-end delays are more predictable than the priority-driven approach. We believe that our work in this paper provides a useful and complete design methodology to many real-time systems that are statically structured, such as those in automated factory, automobile and avionics systems [12, 13].

The rest of this paper is organized as follows. Section 2 reviews the basic models of end-to-end scheduling and pinwheel scheduling. Section 3 presents the distributed pinwheel scheduling algorithm **DSr**. We discuss its complexity and the schedulability condition bound. Section 4 presents the single-node pinwheel scheduler. In Section 5, we investigate the algorithms that can be used to reduce the end-to-end delays by adjusting the phases of pinwheel schedules on different nodes. The paper is concluded in Section 6.

## 2 Scheduling models

In distributed real-time systems, transactions are decomposed into a set of tasks. The tasks are executed sequentially or concurrently on different processors. One of the most important timing constraints for a real-time transaction is its end-to-end delay which is defined by the time between when the transaction is initiated by the arrival of some input data and when the transaction has completed all corresponding processing on all nodes. The primary objectives of end-to-end scheduling for hard real-time systems are to ensure that (1) individual hard deadlines for tasks on each processor are met, and (2) the end-to-end deadlines for all transactions are met.

For real-time systems with independent tasks on a single processor, the most well-known scheduling algorithms are the Rate Monotonic and the Earliest Deadline First algorithms[14]. The performance of RM and the schedulability of tasks on multi-processor systems have been studied [4, 6]. Algorithms have been proposed for dynamic hard real-time distributed systems with shared resources [15, 19]. The issues of end-to-end timing constraints have also been studied [1, 5, 17, 18]. However, most previous work on end-to-end scheduling uses the RM or priority-driven approach. The algorithms proposed are usually complicated and the bound for the end-to-end response is loose. This is because the effective schedule (the shortest schedule pattern which can represent the entire schedule) generated using the priority-driven approach is usually long (exponential to input size) and irregular to be efficiently and effectively analyzed for a general improvement. Therefore, only the worst case is analyzed in general. In this section, we first discuss an end-to-end solution using the rate monotonic scheduling method and then review the pinwheel model.

### 2.1 End-to-end scheduling

In [17], the rate monotonic scheduling (RMS) method is used to meet the end-to-end performance requirement of a transaction running on three processors. It uses a partitioning approach to divide the end-to-end deadline into the individual deadline for each task. One of the problems with the RMS scheduling is that each periodic task is only guaranteed to complete at the end of the period. Even if a task uses only a short amount of time (e.g. 10ms) in each period (e.g. 200ms), its successor task may have to wait for the whole period before a result is available. Therefore, the total response time for a transaction with a sequence (pipeline) of tasks usually is defined to be the sum of all task periods in the sequence.

However, the statement is not always true. In fact, unless tasks are synchronized on different nodes (i.e. their periods start at some fixed relative times), a result produced on one node by a task may not be received in the next node until almost two periods later. This can be seen in Figure 1, where the output from $T_{i1}$ in the first period is too late to be used in the first period of $T_{i2}$, and again the output from the second period of $T_{i2}$ is too late as the input data for the third period of $T_{i3}$. In such situations, after a time interval that equals to the sum of all task periods, we cannot guarantee that an input data value has been reflected in the output produced by the last task. In fact, in the worst-case situations, the total end-to-end delay for a specific input data could be as long as almost twice the sum of all tasks' periods (Figure 1).

Another problem is that when an end-to-end deadline cannot be met, the design approach proposed in [17] uses a manual or ad hoc decision to improve it. In their example, the end-to-end requirement for the example control transaction is 785 ms, and, the sum of all tasks' periods is 800 ms. Therefore, one of the tasks is assigned a deadline shorter than its period. The decision as to which task to choose is left as a user's decision. Such an ad hoc design approach may be tedious and produce only sub-optimal results.

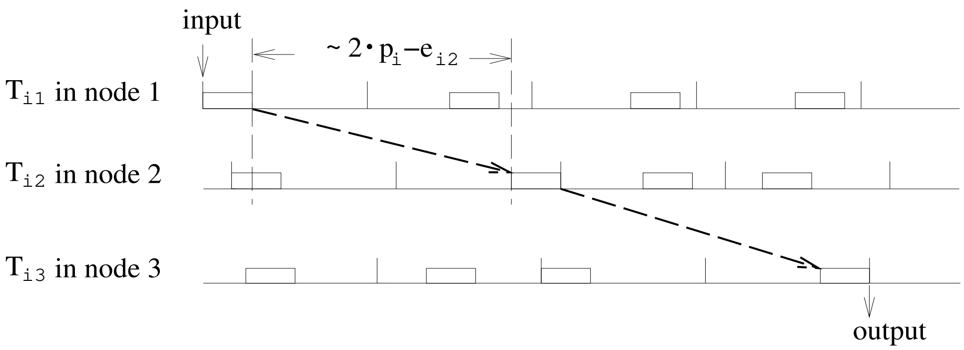


Figure 1: Worst case end-to-end delay for RMS

### 2.2 Pinwheel scheduling

In some real-time applications, tasks must be executed in a *distance-constrained* manner, rather than periodically. That is, the temporal distance between any two consecutive executions (or inter-execution) of a task should always be less than a certain amount of time. Such a real-time system is called a *Distance-Constrained Task System* (DCTS) [7]. For example, video systems need to ensure the inter-arrival time between video frames to be always less than 33 ms. This is different from receiving at least one frame in every 33 ms interval, which allows for an inter-arrival time of greater than 33 ms.

Formally, in a DCTS task set $\mathbf{T}$, every task $T_i$ consists of a sequence of jobs $J_i^1$, $J_i^2$, $J_i^3$,.... Task $T_i$ has constant execution time $e_i$ and distance constraint $c_i$. The difference between the finishing times of any two consecutive jobs of $T_i$ must be less than $c_i$. We assume that there is a precedence constraint between jobs of the same task, i.e., job $J_i^{j+1}$ cannot be started until job $J_i^j$ has finished. Moreover, jobs are preemptable.

Our research for the DCTS scheduling problem is based on the pinwheel scheduler **Sr** [7] which uses a period *specialization* technique to transform all task periods to harmonic numbers. In a DCTS, it is necessary that the transformed periods are less than or equal to the original periods. There are many solutions to transform the periods so that the distance constraints can be met. Pinwheel schedulers transform the periods into a set of special periods so that the total density (defined next) increase is minimized. The process was called specialization in [7] but we called it transformation in this paper.

Let $\mathbf{T}$ be a DCTS task set with distance constraint multiset $\mathbf{C}=\{c_1, c_2, ..., c_n\}$, $c_1 \le c_2 \le ... \le c_n$. We define $\tau(T_i) = e_i/c_i$ as the *density* of task $T_i$. The total density of $\mathbf{T}$ is $\tau(\mathbf{T}) = \sum_{i=1}^{n} e_i/c_i$. Let $\Phi_T(r)$ be the density of $\mathbf{T}$ when its distance constraint $\mathbf{C}$ is transformed with respect to $\{r\}$, i.e. every

$c_i$ in $\mathbf{C}$ is transformed to a number $b_i = r2^{\lfloor \lg(c_i/r) \rfloor}$. Note that $b_i \leq c_i$. **Sr** works as follows:

1. Set $l_i = c_i/2^{\lceil \lg(c_i/c_1) \rceil}$, for $1 \leq i \leq n$ (note that $c_1/2 < l_i \leq c_1$).

2. Sort $l_i$'s and remove all duplicates. Call the sorted result $\{k_1, ..., k_u\}$, where $k_1 < k_2 < ... < k_u$, $u \leq n$, the *special base* of $\mathbf{T}$.

3. Compute all $\Phi_T(k_i)$'s, for all $k_i$ in the special base of $\mathbf{T}$, and select the one with the minimum value. Use that $k_i$ as the $r$ in the transformation process to find $b_i$.

4. Schedule all tasks as periodic tasks using $b_i$'s as the periods.

For example, suppose a system has five tasks with distance constraints {9.2, 10.6, 10.7, 21.4, 23.4} respectively, and their execution times are {1.5, 2.0, 3.4, 1.4, 3.0} respectively. After applying **Sr**,

1. $\{l_1, l_2, \ldots, l_5\} = \{9.2, 5.3, 5.35, 5.35, 5.85\}$

2. $\{k_1, k_2, \ldots, k_4\} = \{5.3, 5.35, 5.85, 9.2\}$

3. $\{\Phi_T(k_1), \Phi_T(k_2), \ldots, \Phi_T(k_4)\} = \{1.0, 1.17, 1.42, 0.99\}$

Since $\Phi_T(k_4)$ is the minimum, we chose $r = k_4 = 9.2$ for period transformation. The five tasks will have new distance constraints {9.2, 9.2, 9.2, 18.4, 18.4}. These tasks will then be scheduled as periodic tasks using the new distance constraint as their periods. Since the periods are harmonic numbers, the execution schedule for each task will have no jitter and meet the original distance constraint. In Figure 2(b), the schedule for the task set when it is transformed with respect to $r = 5.3$ is shown. The rate monotonic schedule using the original distance constraints as periods is also shown in Figure 2(a). Note that all tasks in the pinwheel schedule has a constant inter-completion time.

It has been shown [7] that if $\tau(\mathbf{T}) \leq n(2^{1/n} - 1)$, $\mathbf{T}$ is schedulable using **Sr**. The pinwheel scheduling has also been applied to provide real-time network scheduling on DQDB and FieldBus [8, 9].

# 3 Distributed pinwheel scheduling algorithm

To use the pinwheel scheduling method in distributed systems, we must modify **Sr**. The reason is as follows. Although we can use **Sr** to find a pinwheel schedule for each node and schedule each node independently, tasks of the same transaction on different nodes may be transformed to have different

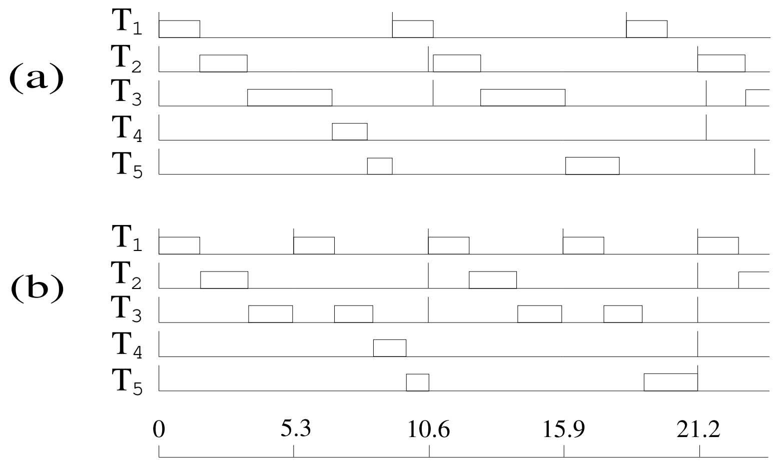


Figure 2: The (a)RMS and (b)Pinwheel schedule

distance constraints. This will make the transaction processing very difficult. It is necessary to have the same period for all tasks of the same transaction even on different nodes.

### 3.1 Scheduler DSr

Let **X**$= \{X_i\}$ be a set of transactions in a distributed system. Suppose the number of transactions is $n$, and the number of processor nodes is $m$. Let $T_{ij}$ be the task of $X_i$ executing on node $N_j$. $T_{ij}$ has the distance constraint $C_i$ and execution time $e_{ij}$. Without loss of generality, we assume that $\forall i$, $C_i \leq C_{i+1}$. In this paper, we assume that all transactions have the same (or uniform) node sequence, i.e. all transactions have the first task running on node $N_1$, the second task on node $N_2$, etc. We also assume the buffer size in each node is large enough to temporarily store all incoming data. Moreover, transactions with end-to-end delays have higher priorities than non-real-time transactions so that network delay can be neglected. Our approach can be applied to more complex node sequences such as ring, star, or tree structure with minor modifications. They are beyond the scope of this paper. Finally, it is non-trivial to predict the end-to-end delays of transactions with non-uniform node sequences using any existing approach.

Algorithm **DSr** works as follows. We first define $l_k = C_k/2^{\lceil \lg(C_k/C_1) \rceil}$. Let $\Phi_{N_j}(l_k)$ be the density of the task set in node $N_j$ when their distance constraints are transformed with respect to $\{l_k\}$, i.e.

$$\Phi_{N_j}(l_k) = \sum_{i=1}^{n} \frac{e_{ij}}{B_{ik}}$$

**Algorithm DSr**
Input: DCTS $\mathbf{X} = \{X_i \mid 1 \le i \le n\}$, where $X_i = \{T_{ij} = (e_{ij}, C_i) \mid 1 \le j \le m\}$, and $C_i \le C_{i+1}$.
Output: $\mathbf{X}' = \{T'_{ij} = (e_{ij}, b_i) \mid 1 \le i \le n, 1 \le j \le m\}$.

1. $L = \emptyset$;
2. **for** $k = 1$ **to** $n$ **do** $\{$
3. $\quad l_k = C_k / 2^{\lceil \lg(C_k/C_1) \rceil}$;
4. $\quad$ **for** $i = 1$ **to** $n$ **do** $B_{ik} = l_k \cdot 2^{\lfloor \lg(C_i/l_k) \rfloor}$;
5. $\quad$ **for** $j = 1$ **to** $m$ **do** $\Phi_{N_j}(l_k) = \sum_{i=1}^{n} e_{ij}/B_{ik}$;
6. $\quad$ **if** $\forall j, \Phi_{N_j}(l_k) \le 1, L = L \cup \{l_k\}$;
7. $\}$
8. find $r^*$ s.t. $\Phi_N(r^*) = \min_{r \in L} \sum_{j=1}^{m} \Phi_{N_j}(r)$;
9. **for** $i = 1$ **to** $n$ **do** $b_i = r^* \cdot 2^{\lfloor \lg(C_i/r^*) \rfloor}$;

Figure 3: **Algorithm DSr**

where $B_{ik} = l_k \cdot 2^{\lfloor \lg(C_i/l_k) \rfloor}$. And let $\Phi_N(l_k) = \sum_{j=1}^{m} \Phi_{N_j}(l_k)$. We compute the $\Phi_{N_j}(l_k)$'s for every node $N_j$ using every $l_k$ in $\{l_1, ..., l_n\}$. Any $l_k$ which causes any of the nodes to have a total density greater than 1 will be removed from consideration. We select the $l_k$ with the minimum value of $\Phi_N(l_k)$. This $l_k$ is selected as $r^*$ to convert the distances of all transactions. Figure 3 shows the **DSr** algorithm. Since line 5 of DSr involves a summation of $n$ numbers and it is nested in $n \times m$ loops, we have the following theorem:

**Theorem 1** *The time complexity for **DSr** with $n$ transactions in $m$ nodes is $O(mn^2)$.*

The time complexity can be improved to $O(n \lg n + mn)$. To achieve that, a technique as used in **Sr** [7] can be applied. It first sorts $\{l_1, ..., l_n\}$ into $\{k_1, ..., k_u\}$ with duplicates removed and computes $\Phi_{N_j}(k_u)$, then uses $\Phi_{N_j}(k_u)$ to derive $\Phi_{N_j}(k_{u-1})$, etc. For simplicity, we only present the simpler **DSr** algorithm in this paper.

An example system with five transactions executing on 2 nodes, $N_1$ and $N_2$ is now given. There are five tasks on each node. Distance constraints $C_1, ..., C_5$ are 9.2, 10.6, 10.7, 21.4, 23.4 respectively. On $N_1$, executing times $e_{11}$ to $e_{51}$ are 1.5, 2.0, 3.4, 1.4, 3.0. On $N_2$, executing times $e_{12}$ to $e_{52}$ are 1.2, 1.5, 3.3, 2.1, 4.7. So $\{l_1, l_2, ..., l_5\}$ is $\{9.2, 5.3, 5.35, 5.35, 5.85\}$. **DSr** chooses $r^* = 5.3$. After the conversion process, the five tasks are in three groups with periods $\{5.3\}$, $\{10.6, 10.6\}$, and $\{21.2, 21.2\}$. Their schedules are shown in Figure 4. The densities on both nodes after transformation are 1. Figure 4 shows the two schedules have a relative phase 2.1. This will be discussed in Section 5.

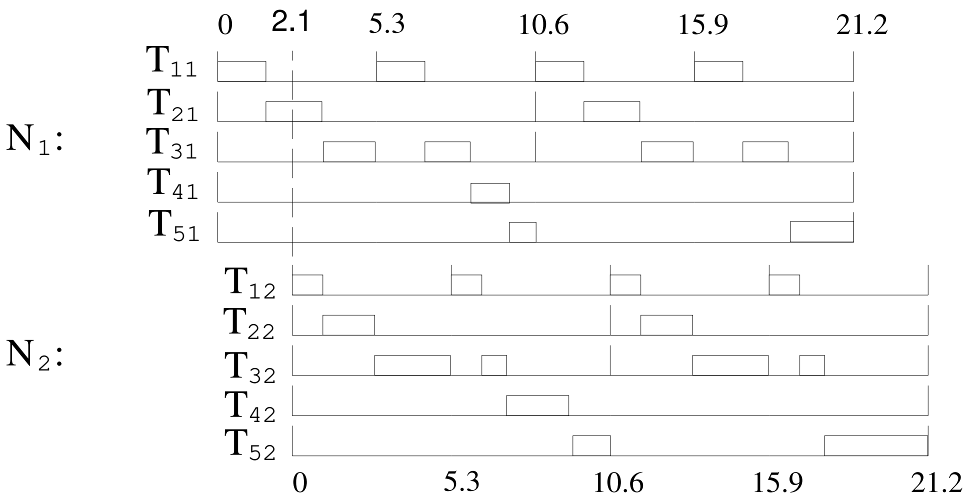



Figure 4: Pinwheel schedules for Nodes $N_1$ and $N_2$

### 3.2 Schedulability condition of DSr

It has been shown [7] that if $\tau(\mathbf{T}) \leq n(2^{1/n}-1)$, $\mathbf{T}$ is schedulable using **Sr**. For **DSr**, we also would like to find a schedulability bound. For each node $N_j$, we define $\tau_{N_j}(\mathbf{X}) = \sum_{i=1}^{n} e_{ij}/C_i$ as the total density of the task set of $\mathbf{X}$ on $N_j$. Since we assume the execution times of tasks in each node are independent, the maximum $\tau_{N_j}(\mathbf{X})$ to satisfy the sufficient condition should be the same. We will find the bound $\beta$ so that if , for all $N_j$, $\tau_{N_j}(\mathbf{X}) \leq \beta$, then $\mathbf{X}$ is schedulable.

To find the schedulability bound $\beta$, we need to derive the relationship between $\tau_{N_j}(\mathbf{X})$ and $\Phi_{N_j}(l_k)$. Given $B_{ik} = l_k \cdot 2^{\lfloor \lg(C_i/l_k) \rfloor}$ and $l_k = C_k/2^{\lceil \lg(C_k/C_1) \rceil}$, we will show later in Lemma 3,

$$\frac{\Phi_{N_j}(l_k)}{\tau_{N_j}(\mathbf{X})} \leq \begin{cases} C_n/B_{n1}, & \text{if } k=1; \\ C_{k-1}/B_{k-1,k}, & \text{otherwise.} \end{cases}$$

Therefore, we need to analyze the term $C_i/B_{ik}$, i.e.

$$C_i/B_{ik} = C_i/((C_k/2^{\lceil \lg(C_k/C_1) \rceil})2^{\lfloor \lg(C_i/(C_k/2^{\lceil \lg(C_k/C_1) \rceil})) \rfloor})$$

The expression will be simplified by variable substitutions in Lemma 2.

Let $x_i = \lfloor \lg C_i \rfloor$ and $y_i = \lg C_i - x_i$. Therefore, $C_i = 2^{x_i+y_i}$. By definition, we know $0 \leq y_i < 1$. W.l.o.g., let's assume $y_1 \leq \ldots \leq y_i \leq \ldots \leq y_n$. We first investigate the relationship between $C_i/B_{ik}$

and $C_j/B_{jk}$ for any $i$ and $j$.

**Lemma 2** *$C_i/B_{ik} \geq C_j/B_{jk}$ iff (a) $i \geq j \geq k$, (b) $k \geq i \geq j$, or (c) $j \geq k > i$.*

*Proof:* For all $i, j, k$, we know that $C_i, B_{ik}, C_j, B_{jk}$ are all greater than or equal to 0. So, the expression, $C_i/B_{ik} \geq C_j/B_{jk}$, is equivalent to

$$\lg C_i - \lg B_{ik} \geq \lg C_j - \lg B_{jk} \tag{1}$$

By substituting $C_i$, $C_j$, $C_k$ and $C_1$ with $2^{x_i+y_i}$, $2^{x_j+y_j}$,$2^{x_k+y_k}$ and $2^{x_1+y_1}$ respectively, we can get

$$y_i - y_j \geq \lfloor y_i - y_k \rfloor - \lfloor y_j - y_k \rfloor \tag{2}$$

Since $0 \leq y_i, y_j, y_k < 1$, we know $-1 < y_u - y_v < 1$ and $\lfloor y_u - y_v \rfloor = 0$ or $-1$, for $1 \leq u, v \leq n$. Because if $u \leq v$, $y_u \leq y_v$, (2) holds if and only if (a) $i \geq j \geq k$, (b) $k \geq i \geq j$, or (c) $j \geq k > i$ . So does $C_i/B_{ik} \geq C_j/B_{jk}$ if and only if one of the conditions is true. □

Now, we can derive the relationship between $\Phi_{N_j}(l_k)$ and $\tau_{N_j}(\mathbf{X})$.

**Lemma 3** *It is true that either $\Phi_{N_j}(l_1) \leq \tau_{N_j}(\mathbf{X}) \cdot C_n/B_{n1}$ or $\forall k, k \neq 1$ $\Phi_{N_j}(l_k) \leq \tau_{N_j}(\mathbf{X}) \cdot C_{k-1}/B_{k-1,k}$.*

*Proof:* Our proof is divided into two cases.

**Case 1:** for $k = 1$. By definition, $\Phi_{N_j}(l_1) = \sum_{i=1}^{n} e_{ij}/B_{i1}$, that is

$$\Phi_{N_j}(l_1) = e_{nj}/B_{n1} + \sum_{i=1}^{n-1} e_{ij}/B_{i1}.$$

Since $\tau_{N_j}(\mathbf{X}) = \sum_{i=1}^{n} e_{ij}/C_i$, by substituting $e_{nj}$ in $\Phi_{N_j}(l_1)$ with $C_n(\tau_{N_j}(\mathbf{X}) - \sum_{i=1}^{n-1} e_{ij}/C_i)$, we can get

$$\Phi_{N_j}(l_1) = \tau_{N_j}(\mathbf{X}) \cdot C_n/B_{n1} + \sum_{i=1}^{n-1} e_{ij}(C_i/B_{i1} - C_n/B_{n1})/C_i.$$

From condition (a) of Lemma 2, we know that for $1 \leq i \leq n-1$, $C_n/B_{n1} \geq C_i/B_{i1}$. So $\sum_{i=1}^{n-1} e_{ij}(C_i/B_{i1} - C_n/B_{n1})/C_i \leq 0$ and $\Phi_{N_j}(l_1) \leq \tau_{N_j}(\mathbf{X}) \cdot C_n/B_{n1}$.

**Case 2:** for $k \neq 1$. Again, by definition,

$$\Phi_{N_j}(l_k) = e_{k-1,j}/B_{k-1,k} + \sum_{1 \leq i \leq n, i \neq k-1} e_{ij}/B_{i1}.$$

by substituting $e_{k-1,j}$ in $\Phi_{N_j}(l_k)$ with $C_{k-1}(\tau_{N_j}(\mathbf{X}) - \sum_{1\le i\le n, i\ne k-1} e_{ij}/C_i)$, we can get

$$\Phi_{N_j}(l_k) = \tau_{N_j}(\mathbf{X}) \cdot C_{k-1}/B_{k-1,k} + \sum_{1\le i\le n, i\ne k-1} e_{ij}(C_i/B_{ik} - C_{k-1}/B_{k-1,k})/C_i$$

Again, by condition (b) and (c) of Lemma 2, we know for $1 \le i \le n, i \ne k-1$, $C_{k-1}/B_{k-1,k} \ge C_i/B_{ik}$, so $\sum_{1\le i\le n, i\ne k-1} e_{ij}(C_i/B_{ik} - C_{k-1}/B_{k-1,k})/C_i \le 0$ and $\Phi_{N_j}(l_k) \le \tau_{N_j}(\mathbf{X}) \cdot C_{k-1}/B_{k-1,k}$. □

The schedulability bound for **DSr** is shown as follows.

**Theorem 4** *Given a distance constraint transaction set* $\boldsymbol{X}$ *with* $n$ *transactions executing on* $m$ *nodes,* $\{N_1, \ldots, N_m\}$*,* ***DSr*** *is guaranteed to find a feasible schedule for* $\boldsymbol{X}$ *if* $\forall N_j$, $\tau_{N_j}(\boldsymbol{X}) \le \beta = 2^{1/n-1}$*. Moreover, the bound* $\beta$ *is tight.*

*Proof:* Given some $l_k$, if for all nodes $N_j$, $\Phi_{N_j}(l_k)$ is less or equal to 1, we know $\mathbf{X}$ is schedulable.

If $\mathbf{X}$ is transformed with respect to $l_1$, by Lemma 3, we know $\Phi_{N_j}(l_1) \le \tau_{N_j}(\mathbf{X}) \cdot C_n/B_{n1}$. If $\tau_{N_j}(\mathbf{X}) \cdot C_n/B_{n1} \le 1$, $\Phi_{N_j}(l_1)$ is less or equal to 1. So if $\beta \le B_{n1}/C_n$ (all $\tau_{N_j}(\mathbf{X}) \le \beta$), for all $N_j$, $\Phi_{N_j}(l_1) \le \tau_{N_j}(\mathbf{X}) \cdot C_n/B_{n1} \le 1$ and $\mathbf{X}$ is schedulable.

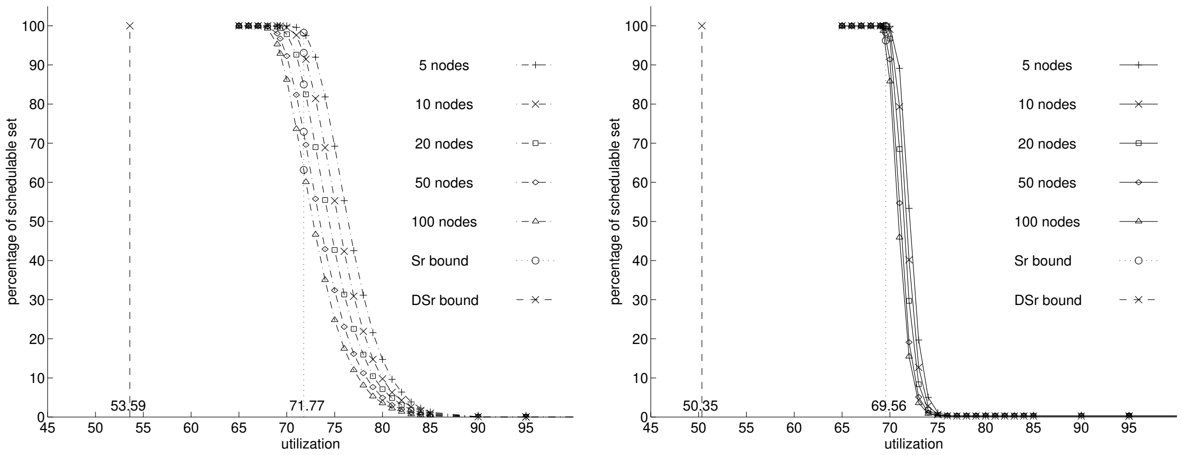
Similarly, for all $l_k, k \ne 1$, if $\beta \le B_{k-1,k}/C_{k-1}$, $\mathbf{X}$ is schedulable. These $n$ inequalities, which are the sufficient conditions for the schedulability of $\mathbf{X}$ using **DSr**, can be simplified, using a technique similar to that used in Lemma 2, to the following inequalities:

$$\begin{aligned}
\lg\beta &\le \lfloor y_n - y_1 \rfloor - (y_n - y_1) \\
\lg\beta &\le \lfloor y_1 - y_2 \rfloor - (y_1 - y_2) \\
&\ \ \vdots \\
\lg\beta &\le \lfloor y_{n-1} - y_n \rfloor - (y_{n-1} - y_n)
\end{aligned}$$

Now, we try to find the maximum $\beta$ such that at least one of these inequalities holds. By adding the above $n$ inequalities, we get

$$n\lg\beta \le \lfloor y_n - y_1 \rfloor + \sum_{i=1}^{n-1} \lfloor y_i - y_{i+1} \rfloor$$

Since $0 \le y_i \le y_{i+1} < 1$, we know $\lfloor y_n - y_1 \rfloor = 0$, and $\lfloor y_i - y_{i+1} \rfloor = 0$ or $-1$. The worst case is that none of $y_i$ is equal to $y_{i+1}$. In this case, $n\lg\beta \le -(n-1)$ or $\lg\beta \le 1/n - 1$. So the maximum $\beta$ such that if every $\tau_{N_j}(\mathbf{X}) \le \beta$, $\mathbf{X}$ is schedulable is $2^{1/n-1}$.




$(a)$ 10 *transactions* $(b)$ 100 *transactions*

Figure 5: Schedulable set using **DSr**

To show the bound is tight, $\forall i$, let $y_i = (i-1)/n$. The $n$ inequalities become $\lg \beta \leq 1/n - 1$. So if $\beta > 2^{1/n-1}$, none of the $n$ inequalities holds. Consequently, $\mathbf{X}$ may not be schedulable if $\beta > 2^{1/n-1}$. Thus, the bound $\beta$ is tight. $\square$

We now give an example system to show the bound is tight. The system has two transactions running on two nodes. $X_1$ has the distance constraint $C_1 = 100$ ms and $X_2$ has $C_2 = 141.42$ ms. $X_1$'s execution time $e_{11}$ is 0.01 ms and $e_{12}$ 70.7 ms. $X_2$'s $e_{21} =$ is 100 ms and $e_{22} = 0.03$ ms. The total density on both nodes is 0.7072, slightly higher than the schedulability bound $2^{1/2-1} = 0.7071$ (but smaller than the Sr schedulability bound, 0.8284). Using DSr, $l_1$ is 100 and $l_2$ is 70.71. If we choose $l_1$ as the $r^*$, the density on node 1 becomes $0.0001 + 1$, or 1.0001. On the other hand, if $l_2$ is chosen, the density on node 2 is $0.9999 + 0.0002$, or 1.0001. The system is not schedulable using **DSr**.

As $n$ goes to infinity, $\beta$ approaches a value of 0.5. That is, if the utilization in every node is less than 0.5, **DSr** is guaranteed to find a schedule for $\mathbf{X}$ in all nodes. To see the average behavior, we have conducted a set of simulations. In each simulation, we choose a $\beta$ value and generate 10000 data sets. Each simulation studies either 10, 20, 50 or 100 transactions and on either 5, 10, 20 , 50 or 100 nodes. The transaction utilization on each node is randomly generated but the total utilization is always less than $\beta$. In Figure 5 the schedulable percentages for 10 and 100 transactions are shown. In general, the more nodes there are in the system, the worse the schedulable percentage is. Note that the simulation result is closer to the schedulability bound of **Sr** (dotted line) than the bound of **DSr** (dash line). In our simulations, even when $\beta$ is 0.65 or higher, almost all randomly generated transaction sets are schedulable. However, since the **DSr** schedulability bound is tight, there is no guarantee for a feasible schedule if some nodes have a total utilization between $2^{1/n-1}$ and $n(2^{1/n}-1)$.

## 4 Producing the Single-node pinwheel schedule

A simple approach to build a scheduler for periodic tasks is to generate an LCM (least common multiple of all task periods) schedule off-line and, then execute the tasks according to the schedule cyclically. However, this is not desirable because the LCM schedule is of exponential length in general [11]. A better approach is to generate some parameters off-line and, then use them to construct a fast online scheduler(FOLS) which can generate the scheduling sequence in constant time [11]. Several constant-time FOLS's using $O(n)$ hardware have been presented for some classes of pinwheel problem instances [11, 2, 3]. For a DCTS with distance constraints of solely harmonic numbers, a modified RMS, called **DMS** (distance monotonic scheduling), can be used as a FOLS. DMS assigns the task with smaller distance constraint a higher priority and imposes proper separation constraints [7]. The main difference between DMS and RMS is that in RMS, tasks are assumed to arrive only periodically, while DMS assumes tasks are always ready for execution as soon as its current execution is finished.

Using the pinwheel approach to schedule a DCTS in a single node, the FOLS approach is proper. However, in order to shorten and to predict the end-to-end delay in a distributed system we need to construct the node schedules a priori so that we can adjust the relative phases between them (to be discussed in Section 5). Since the pinwheel LCM schedule with harmonic distance constraints is jitterless (using DMS), specifying the start time and the finishing time for each task in its first period is sufficient to represent the whole schedule. We now describe an algorithm that can produce the pinwheel schedule in polynomial time.

Using **DSr** defined in the last section, we can transform all distance constraints so that $\forall i,\ c_i|c_{i+1}$. Let $s_i$ be the start time of $J_i^1$ and $f_i$ be the finishing time of $J_i^1$. Since $c_i|c_{i+1}$, the length of the LCM schedule is $c_n$, the start time of $J_i^j$ is $s_i + (j-1)\cdot c_i$ and the finishing time of $J_i^j$ is $f_i + (j-1)\cdot c_i$. Therefore, the $s_i$'s and $f_i$'s are sufficient to represent the whole schedule.

Using DMS, $J_i^j$ can be started only after $J_i^{j-1}$ is finished. Therefore, normally, $s_i$ is equal to $f_{i-1}$.

**Algorithm** $SSr$
Input: DCTS $\mathbf{T}= \{T_i = (e_i, c_i) \mid 1 \leq i \leq n\}$ and $c_i | c_{i+1}, 1 \leq i < n$.
Output: $(s_i, f_i), 1 \leq i \leq n$.

**if** $\tau(\mathbf{T}) > 1$, $\mathbf{T}$ is un-schedulable;
$s_1 = 0, f_1 = e_1, slack_1 = c_1 - e_1$; // $slack_i$ is the slack time in any period of $T_i$ after
**for** $i = 2$ **to** $n$ **do** { // $T_1 \ldots T_i$ are scheduled.
  $s_i = NextIdle(f_{i-1}, i)$; // Set $s_i$ to the next idle time after $T_{i-1}$ is scheduled.
  $e' = e_i, f_i = 0$;
  **for** $k = i - 1$ **downto** 1 **do** {
    $f_i = f_i + c_k \cdot \lfloor e'/slack_k \rfloor$; // Set $f_i$ to the beginning of $\epsilon_k$-th period of $T_k$.
    $e'' = e' \bmod slack_k$; // $e'$ is the equivalent execution time in $\epsilon_{k+1}$-th period of $T_{k+1}$.
    **if** $e'' = 0$, { $e' = 0$; **break**; } // $e''$ is $e'$ left in $\epsilon_k$-th period of $T_k$.
    **else** $e' = e'' + e_k$;
  }
  $f_i = f_i + e'$;
  $slack_i = slack_{i-1} \cdot c_i / c_{i-1} - e_i$;
}

$NextIdle(t, i)$ // Returns the next idle time since time $t$ for $T_i$.
**if** $(t \bmod c_1) \neq 0$, **return** $t$;
**for** $k = i - 1$ **downto** 1 **do**
  **if** $(t \bmod c_k) = 0$, **return** $NextIdle(t + f_k, k)$;

Figure 6: **Algorithm** $SSr$ **for finding the single node schedule**

However, if there are jobs with a higher priority requested at $f_{i-1}$, $s_i$ must be delayed to the time after the execution of those higher-priority jobs. Moreover, whenever $s_i$ is delayed to the time when those higher-priority jobs requested at $f_{i-1}$ all finish, there may be some other higher-priority jobs requested at this time. This situation happens recursively. Since only the jobs with a higher priority than $J_i$ will further delay $s_i$, the number of recursions is bounded by $i$. In Figure 6, the search for $s_i$ is implemented by the $NextIdle()$ function. Its complexity is $O(n)$.

Before we show how to find $f_i$, we discuss a property of the slack time using **Sr**.

**Corollary 5** *If tasks $T_1 \ldots T_i$ are scheduled using **Sr**, the slack time for $T_i$ in each $T_i$ period is the same.*

The corollary follows directly from the proof of Theorem 1 in [7] where it has been shown that "For any task $T_i$, if it is running at time $t$ (using **Sr**), then it is also running at time $t + q \cdot c_i$ for any integer

$q$ such that $t + q \cdot c_i \geq 0$." Suppose $T_i$ finishes in the $\epsilon$-th $T_{i-1}$ period from the beginning of $T_i$'s period. Let $e''$ be the execution time of $T_i$ left in the $\epsilon$-th $T_{i-1}$ period. We can make the following observation.

**Corollary 6** *The finish time of $T_i$ is the same as the finish time of $T_{i-1}$ in the $\epsilon$-th $T_{i-1}$ period if $T_{i-1}$ has an execution time of $e'' + e_{i-1}$ in that period.*

Now, we are ready to show how to find $f_i$. Let $slack_i$ be the slack time in any period of $T_i$ after $T_1 \ldots T_i$ are scheduled. Before we schedule $T_i$, the time available to $T_i$ in a period of $T_i$ is $slack_{i-1} \cdot c_i / c_{i-1}$. From Corollary 5, we can get

$$slack_i = slack_{i-1} \cdot c_i / c_{i-1} - e_i.$$

We schedule $T_i$ as we fill $e_i$ into the slacks for $T_i$ in each of the $T_{i-1}$ periods. In each $T_{i-1}$ period, $T_i$ can execute for at most $slack_{i-1}$ amount of time. Therefore, $f_i$ must fall into the $\lceil e_i / slack_{i-1} \rceil$-th period of $T_{i-1}$. We thus know that for $T_i$ $\epsilon = \lceil e_i / slack_{i-1} \rceil$ and $e'' = e_i \bmod slack_{i-1}$. From Corollary 6, we try to find the finish time of $T_{i-1}$ with execution time $e' = e'' + e_{i-1}$ in the $\epsilon$-th period of $T_{i-1}$. To do that, we apply the same approach as we schedule $T_i$. After recursively applying this approach until finding the finishing time of $T_1$ in one of the periods of $T_1$, $f_i$ is finally decided.

In Figure 6, we show the algorithm for $\mathbf{T}$ with multiple distance constraints, called $SSr$. The algorithm works even when $e_i = 0$ or $c_i = c_{i+1}$ for some $i$.

**Theorem 7** *$SSr$ can find a pinwheel schedule and its time complexity in $O(n^2)$.*

The correctness part of Theorem 7 can be proved by induction. It is clear $SSr$ can schedule $T_i$ when $i = 1$. Suppose $SSr$ can schedule $T_i$ when $i = k$. When scheduling $T_{k+1}$, $slack_1 \ldots slack_k$ are all known. From Corollaries 5 and 6, we know the schedule for $T_{k+1}$ can be found, thus the theorem. It is also clear the complexity of $SSr$ is $O(n^2)$ since line 4 has a time complexity of $O(i)$ and it is in a loop of $n$ times. If we group tasks according to their period values and suppose there are $l$ different period values, the complexity to schedule $\mathbf{T}$ can be improved to $O(nl)$, with a slight modification of $SSr$.

In Figure 7, we show how to calculate $s_5$ and $f_5$ for the example of 5 tasks discussed in Section 2.2. After $T_1 \ldots T_4$ are scheduled, we know $slack_1 \ldots slack_5$ (the grey area in Figure 7 (a)). However, when $J_4^1$ finishes $J_1^3$, $J_2^2$, and $J_3^2$ are all ready to execute. $s_5$ needs to be delayed until $J_3^2$ finishes ($s_5 = f_4 + f_3$). In

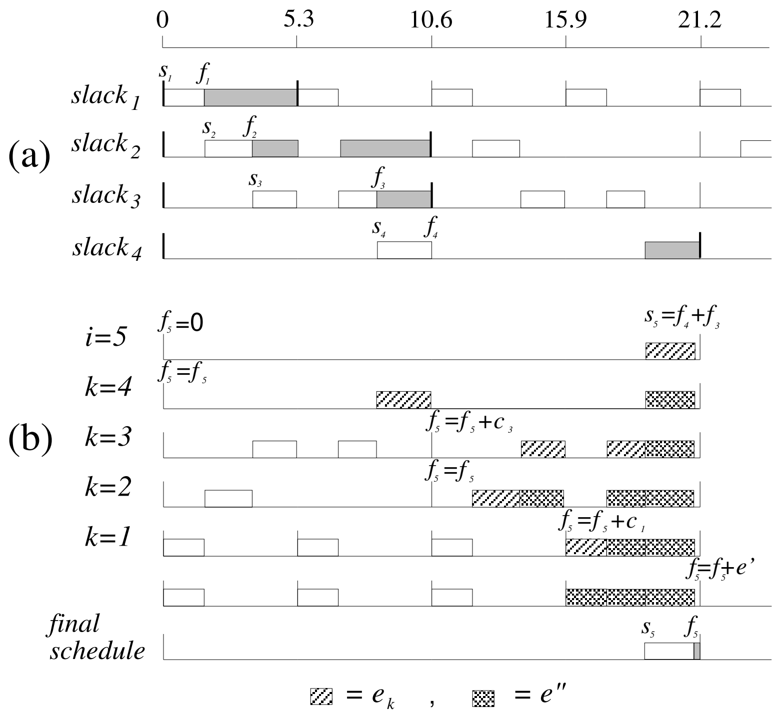



Figure 7: An example for calculating $s_5$ and $f_5$

Figure 7 (b), $f_5$ is initialized to 0. When line 7 is executed with $k = 4, 3, 2, 1$, $f_5$ is set to $0, c_3, c_3, c_3 + c_1$, and $c_3 + c_1 + e'$ respectively. $f_5$ thus can be found in linear steps.

# 5 Pinwheel phase alignment

Using **DSr**, the periods of all transactions are transformed into harmonic numbers. Every node in the system thus has a jitter-free pinwheel schedule. However, due to the data flow constraint, task $T_{i,j+1}$ on node $N_{j+1}$ cannot start execution until task $T_{ij}$ has finished execution on node $N_j$. On the other hand, task $T_{i,j+1}$ must wait until its pre-scheduled time instance on $N_{j+1}$'s pinwheel schedule to start execution. The time interval between the completion of $T_{ij}$ on node $N_j$ and the start time of $T_{i,j+1}$ on $N_{j+1}$ is defined as the *delay* $d_{ij}$. The end-to-end delay for transaction $X_i$ is $\sum_{j=1}^{n} d_{ij}$. The end-to-end response time for $X_i$ is $\sum_{j=1}^{n}(d_{ij} + e_{ij})$ plus the time when any of $X_i$'s being preempted.

The end-to-end delay for a transaction may be improved by adjusting the phases of the pinwheel schedules between nodes. For example, Figure 4 shows the pinwheel schedule on $N_1$ and $N_2$ when phase $h_1 = 2.1$. The delays $d_{11}, d_{21}, d_{31}, d_{41}$ and $d_{51}$ are 1.1, 0.3, 7.5, 0.0 and 11.9 respectively. If we change $h_1$ to 2.0, $d_{11}$ will become 1.0 but $d_{41}$ will jump to 21.1. In other words, a better phase for one transaction may be worse for another. If there is only one transaction with an end-to-end deadline, we can adjust the pinwheel schedules on all nodes so that the transaction has no delay. When there are more than one transactions with end-to-end deadlines, finding an ideal global alignment is non-trivial.

In this paper, we assume that all transactions have end-to-end deadlines. Two objective functions for global alignment are defined in this paper. One is to minimize the total delays received by all transactions in all nodes. Another is to minimize the maximum delay received by all transactions in the system.

## 5.1 Minimize the total delay

Given two pinwheel schedules for two neighboring nodes $N_j$ and $N_{j+1}$, we define the *phase* $h_j$ as the difference between the start times of $T_n$, which is the task with the longest period, on the two nodes. We choose the start time of $T_n$ as the reference point because there is only one unique start time of $T_n$ in any pinwheel schedule. It must be true that $0 \leq h_j < C_n$ since the schedule repeats every $C_n$. The delay of $T_i$ between $N_j$ and $N_{j+1}$, $d_{ij}$, is a periodic function of $h_j$ with period $C_i$. $d_{ij}$ can be defined as

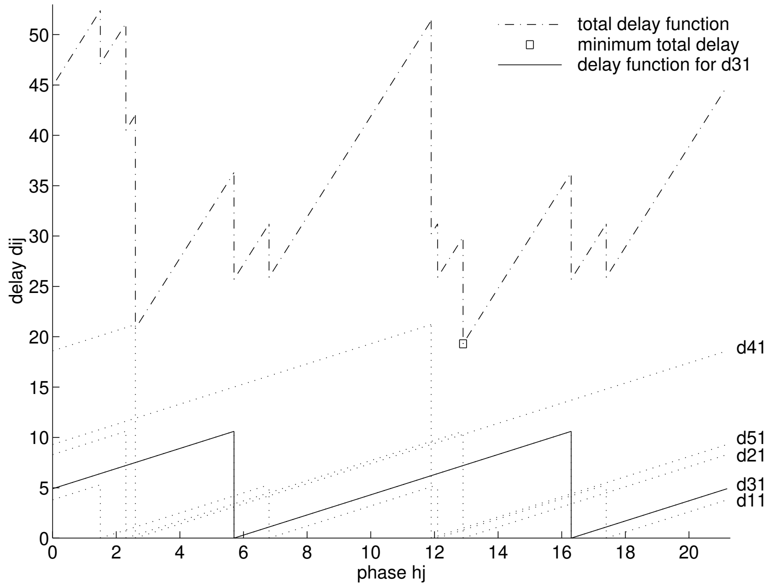


Figure 8: Delay functions and their sum

follows:

$$d_{ij}(h_j) = (h_j + \alpha_{ij}) \bmod C_i, \quad 0 \le h_j < C_n \tag{3}$$

where $\alpha_{ij}$ is the delay for $T_{ij}$ when $h_j = 0$. For the rest of this paper, we sometimes use $d_{ij}$ as a shorthand for $d_{ij}(h_j)$. The total delays of all tasks between nodes $N_j$ and $N_{j+1}$ is defined as $\omega_j(h_j) = \sum_i d_{ij}(h_j)$. $\omega_j$ has the following property.

**Theorem 8** *If the total delay $\omega_j(h_j)$ between nodes $N_j$ and $N_{j+1}$ has the minimum value when $h_j = t$, it must be true that $\exists\ i,\ d_{ij}(t) = 0$.*

*Proof:* Assume $h_j = t'$ is the optimal phase for $\omega_j$ and none of the $d_{ij}(t')$'s is equal to 0, $\forall i$. Suppose the minimum of the $d_{ij}(t')$'s is $d_{kj}(t')$. If we set the phase to $t' - d_{kj}(t')$, all $d_{ij}$'s will be reduced by $d_{kj}(t')$. So, at this new phase, $t' - d_{kj}(t')$, the total delay is less than that at the phase $t'$ by the amount $n \cdot d_{kj}(t')$, thus a contradiction. □

#### 5.1.1 Representing the delay function

Before we present the algorithm for finding the minimum total delay, we first show how to efficiently represent the delay function in our algorithm. The $d_{ij}$ function defined in Equation (3) is a set of discontinued line segments (e.g. $d_{31}$ in Figure 8). Between discontinued points, $d_{ij}$ increases linearly with $h_j$. At each discontinued point, $d_{ij}$ drops from $C_i$ to zero. When several $d_{ij}$ functions are combined, the resulting function is a *serrated line*. In a serrated line, the function value monotonically increases, with a constant slope, between discontinued points. In other words, each serrated line can be defined as a set of line segments with a constant slope. The delay functions and their total function for the example of Figure 4 is shown in Figure 8.

Suppose the total delay $\omega_j(h_j)$ has the minimum value when $h_j = t$. To find $t$, we need to inspect all line segments in the serrated line $\sum_i d_{ij}(h_j)$. From Theorem 8, we know that the minimum $\omega_j(h_j)$ occurs at $h_j = t$ when at least one of the $d_{ij}(t)$ is 0. By searching through all discontinued (or drop) points in $[0, C_n)$, we can find the $t$ value. Unfortunately, there are many drop points in $[0, C_n)$. For example, if $\forall i$, $1 \leq i < n$, $C_{i+1}/C_i = 2$, there will be $2^n - 1$ drop-points. We would like to reduce the number of drop points to be searched.

We use a special data structure to cut down the domain size of a serrated line while keeping all significant drop points (i.e. those with lower $\omega_j$ values). For example, to reduce the domain size of a serrated line $f(t)$ from 100 to 25, we define

$$f'(t) = \min\{f(t), f(t+25), f(t+50), f(t+75)\}$$

for $1 \leq t \leq 25$. Conceptually, the serrated line is cut into equal portions, and only the minimum values of relative points in the portions are kept. To keep track of the original $h_j$ value, each point in our representation has two domains: the current domain and the original domain. Throughout the algorithms, the serrated line will be cut many times and only a subset of the original $\omega_j(h_j)$ function is kept. However, each point in the final serrated line has its unique $h_j$ value in the original domain. The data structure SLINE is defined below.

*Definition:* An SLINE $S = \{s, w, (x, y), \{(p_1, a_1, p'_1), (p_2, a_2, p'_2), \ldots\}\}$ is defined by

1. $s$ is the *slope* of the line segments;
2. $w$ is the current domain size;
3. $(x, y)$ is the origin coordinates in the original domain;

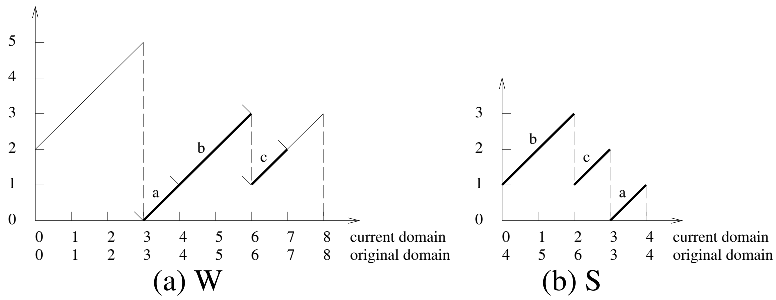



Figure 9: Cutting SLINE (S=CUT{W,4})

4. $(p_i, a_i, p'_i)$'s are a sequence of drop points. For each point, $p_i$ is its current domain value, $p'_i$ is its original domain value, and $a_i$ is the drop amount. The points are sorted on the $p_i$ value. □

Suppose $S$ is an SLINE with $v$ drop points and domain size $E_k$, and $F_1, \ldots, F_u$ are delay functions with period $E_w$. Three operations have been defined for SLINE: SUM, MAX and CUT.

1. SUM$\{S, F_1, \ldots, F_u\}$ is an SLINE that has the sum of all function values.
2. MAX$\{S, F_1, \ldots, F_u\}$ is an SLINE that has the maximum of all function values.
3. CUT$\{S, E_w\}$ is an SLINE with $S$'s domain size cut to $E_w$ but keeps the minimum values in $S$.

Figure 9 shows the CUT operation of SLINE. $W = \{1, 8, (0, 2), \{(3, 5, 3), (6, 2, 6)\}\}$, and $S = CUT\{W, 4\} = \{1, 4, (4, 1), \{(2, 2, 6), (3, 2, 3)\}\}$. Both SUM$\{S,F_1,\ldots,F_u\}$ and MAX$\{S,F_1,\ldots,F_u\}$ can be done in $O(v + u \lg u)$ time. The time complexity of CUT$\{S, E_w\}$ is $O(v \lg(E_k/E_w))$. The detailed definitions and algorithms for these SLINE operations are included in the Appendix.

### 5.1.2 Minimizing the delay between two nodes

Again let $\mathbf{X}$ be a set of $n$ transactions in a distributed system of $m$ processors. $T_{ij}$ is the task of $X_i$ on node $N_j$ with execution time $e_{ij}$. We group tasks according to their period values. Suppose there are $l$ different period values, tasks with the shortest period, $E_1$, are in group $G_1$ and tasks with the longest period, $E_l$, are in group $G_l$. $|G_k|$ is the number of tasks in group $G_k$ and $E_k$ is the period value of task group $G_k$. Therefore $\sum_{k=1}^{l} |G_k| = n$. For any task $T_{ij}$ in a group $G_k$, the value of $d_{ij}(h_j)$ is the same as the value of $d_{ij}(h_j + E_k)$. Moreover, the sum of all $d_{ij}(h_j)$'s for all tasks in $G_k$ is also a periodic

**Algorithm MS**

Input: $E_1, ..., E_l$ are the period lengths for $G_1, ..., G_l$;
$d_{ij}$ is the delay function for $X_i$ between $N_j$ and $N_{j+1}$;
$S$ is defined as an SLINE;

1. $S = \{0, E_l, (0,0), \emptyset\}$;
2. $S =$ **SUM**$\{S, F_l\}$ where $F_l = \{d_{ij} \mid d_{ij} \in G_l\}$; // $F_l$ is a collection of delay functions.
3. **for** $k = l - 1$ **downto** 1 **do** {
4. $S =$ **CUT**$\{S, E_k\}$ ;
5. $S =$ **SUM**$\{S, F_k\}$ where $F_k = \{d_{ij} \mid d_{ij} \in G_k\}$;
6. } //$S$ is the current subtotal function
7. return $t$ where $S(t)$ is the minimum.

Figure 10: Algorithm for the minimum total delay

function of $h_j$ with period $E_k$. We therefore may combine all delay functions of the same task group in one SLINE.

We would like to find the $h_j$ that minimizes $\sum_{i=1}^{n} d_{ij}(h_j)$. Instead of searching through every point in the domain $E_l$, we use the CUT operation to reduce the search domain but keep all potential $h_j$ values. In our algorithm, we add the delay functions (using SUM) one group at a time, until all groups are added, to form the subtotal delay function $S$. Before each SUM operation, we use the CUT operation to reduce the subtotal function $S$ to the next (smaller) domain size.

Figure 10 shows the **MS** algorithm for finding the minimum total delay. Let $\Omega_j^k(h_j)$ be the sum of all $d_{ij}(h_j)$'s in the groups $\{G_k, G_{k+1}, ..., G_l\}$. By definition, we know that $\Omega_j^k(h_j)$ is a periodic function with period $E_k$, since $E_k | E_i$, for $k + 1 \leq i \leq l$. In the **MS** algorithm $\Omega_j^k(h_j)$ is implemented as SLINE $S$ with domain size $E_k$. By applying CUT$(S, E_k)$, $S$ keeps all the points with the minimal $\Omega_j^k(h_j)$ value and also keeps track of their positions in the original domain.

### Algorithm complexity

Since the domain size of $S$ is $E_k$, each $d_{ij}$ in $G_k$ will only contribute one drop point in $S$. The number of new drop-points added will be at most $|G_k|$. Because $\sum_{k=1}^{l} |G_k|$ is $n$, the number of drop-points in $S$ will be no greater than $n$. The complexity of line 2 and line 5 is $O(\sum_{k=1}^{l}(n + |G_k| \lg |G_k|))$, the complexity of line 4 is $O(\sum_{k=1}^{l-1} n \lg(E_{k+1}/E_k))$, and the complexity of line 7 is $O(n)$. Since the $(E_{k+1}/E_k)$'s for a system are constants, the overall complexity of the algorithm is $O(n \lg n + nl)$.

**Algorithm MX1**
Input: $E_1, ..., E_l$ are the period lengths for $G_1, ..., G_l$;
$d_{ij}$ is the delay function for $X_i$ between $N_j$ and $N_{j+1}$;
$S$ is defined as an SLINE;

1. $S = \{0, E_l, (0,0), \emptyset\}$;
2. $S =$ **MAX**$\{S, F_l\}$ where $F_l = \{d_{ij} \mid d_{ij} \in G_l\}$; // $F_l$ is a collection of delay functions.
3. **for** $k = l-1$ **downto** 1 **do** {
4. $S =$ **CUT**$\{S, E_k\}$ ;
5. $S =$ **MAX**$\{S, F_k\}$ where $F_k = \{d_{ij} \mid d_{ij} \in G_k\}$;
6. } //$S$ is the current submaximum function
7. return $t$ where $S(t)$ is the minimum.

Figure 11: Algorithm for the minimax delay for 2 nodes

#### 5.1.3 Minimize the total end-to-end delay

For a sequence of nodes, the phases $h_j$'s between adjacent nodes can be adjusted independently. Therefore, the minimum total delay between any node pair can be achieved independently. To get the minimum total delay for all nodes, we only need to sum up the minimum total delay between any two consecutive pinwheels. That is $MS = \sum_{j=1}^{m-1} \omega_j$. The time complexity to find the minimum total delay of a $m$-node system is $O(mn \lg n + mnl)$.

### 5.2 Minimize the maximum delay

We now study the pinwheel alignment problem with the objective function of minimizing the maximum delay of any task in the system. Similar to Theorem 8, the minimized maximum, or minimax, delay between nodes $N_j$ and $N_{j+1}$ must occur at a phase value $h_j = t$ such that $\exists i$, $d_{ij}(t) = 0$. This is because if $h_j = t'$ and none of the $d_{ij}(t')$'s is equal to 0, we can find the minimum of the $d_{ij}(t')$'s, say $d_{kj}(t')$. By setting $h_j$ to $t' - d_{kj}(t')$, all $d_{ij}$'s will be reduced by the same amount $d_{kj}(t')$. In other words, at $h_j = t' - d_{kj}(t')$, the maximum delay is less than that at $h_j = t'$, a contradiction.

#### 5.2.1 Minimize the maximum delay between two nodes

The **MX1** algorithm (Figure 11) for finding the minimax delay between nodes $N_j$ and $N_{j+1}$ is similar to the algorithm for finding the minimum total delay. The only difference is that the MAX function is applied instead of the SUM function on line 2 and line 5 to get the submaximum function. Therefore,

**Algorithm MX2**
Input: $E_1, ..., E_l$ are the period lengths for $G_1, ..., G_l$;
$d_{ij}$ is the delay function for $X_i$ between $N_j$ and $N_{j+1}$;
$\sigma$ is the sum of delays in previous nodes.
$S$ is defined as an SLINE;

1. $S = \{0, E_l, (0,0), \emptyset\}$;
2. $S =$ **MAX**$\{S, F_l\}$ where $F_l = \{d_{ij} \mid d_{ij} \in G_l\}$; // $F_l$ is a collection of delay functions.
3. **for** $k = l - 1$ **downto** 1 **do** {
4. $S =$ **CUT**$\{S, E_k\}$ ;
5. $S =$ **MAX**$\{S, F_k\}$ where $F_k = \{d_{ij} + \sigma_i \mid d_{ij} \in G_k\}$;
6. } //$S$ is the current submaximum function
7. find $h_j$ where $S(h_j)$ is the minimum.

Figure 12: Algorithm for the modified minimax delay

the complexity is also $O(n \lg n + nl)$.

### 5.2.2 Minimize the maximum end-to-end delay

Unfortunately, even if we find the minimax delay between all adjacent nodes, the minimax end-to-end delay may not equal to the sum of all of them. This is because the best alignment may occur when all delays are evenly distributed among all transactions and among all nodes. In other words, we may need to search through all possible combinations of alignments between different nodes. Suppose $p = \sum_{i=1}^{n} C_n/C_i$, the total number of drop points in the delay function between two nodes, the number of total combinations is $p^{m-1}$.

In [10], we have studied different search strategies to minimize the maximum end-to-end delay of any transaction. Our approach is to use a modified delay function, which includes the actual delay between two nodes, and also the accumulated delays between all previously searched nodes. We call this modified delay function $\delta_j$, and the accumulated delays for $X_i$ in all previous nodes as $\sigma_i$. Different heuristics use different node sequence functions **seq()**'s to find the alignments and different history function **hist()** to compute $\sigma_i$. The algorithm for finding the best alignment between two nodes are shown in Figure 12. Its complexity is $O(n \lg n + nl)$. The algorithm for searching through all nodes are shown in Figure 13. From our simulation result[10], defining the **seq()** function by searching sequentially from node 1 to $m - 1$ and the history function as **hist()** $= d_{ij}(t) + \sigma_i$ usually can achieve a very good result.

**Algorithm HMX**

Input: **seq()** is the heuristic for the node sequence;
$\sigma$ is the sum of delays in all previously processed nodes;
**hist()** decides if $\sigma$ is used for the alignment.

1. $next = 0$; $\sigma = 0$;
2. **while** $(next = \mathbf{seq}(next)) > 0$ **do** { // **seq**() returns the index $j$ of
3. $t = \mathbf{MX2}(next, \sigma)$; // next phase $h_j$ to be aligned.
4. $\forall i, \sigma_i = \mathbf{hist}(d_{ij}(t), \sigma_i)$;
5. } // *next* is set to 0 when all nodes are visited

Figure 13: Heuristics algorithms for minimax delay

# 6 Conclusions

Many real-time systems have well-defined task structures and need to guarantee end-to-end deadlines. In this paper, we present a new methodology to schedule distributed real-time systems with end-to-end deadlines. It is shown that using **DSr**, distributed real-time transactions can be scheduled to minimize the end-to-end delay. We have presented a tight schedulability bound for **DSr**, and also studied the average schedulability bound. We then use a polynomial-time algorithm to find local pinwheel schedule for each node. After each local pinwheel schedule is produced, several phase alignment algorithms can be used to minimize the end-to-end delay of the system. Using this approach, the end-to-end delay for transactions can be better managed.

One of the important reasons for our phase alignment algorithms to work is due to the fact that all periods are harmonic numbers. Without the property, we will not be able to find polynomial time algorithms for many difficult scheduling problems. In this paper, the property of harmonic periods is the result of using **DSr** to transform the original task periods. However, many real-time systems have or can be designed to have harmonic periods. Therefore, even without **DSr**, the proposed methodology is still useful.

# References

[1] Riccardo Bettati, *End-to-End Scheduling to Meet Deadlines in Distributed Systems,* PhD thesis, Technical Report UIUCDCS-R-94-1840, University of Illinois at Urbana-Champaign, August 1994.

[2] M. Y. Chan and F. Chin, "General Schedulers for the Pinwheel Problem Based on Double-Integer Reduction," *IEEE Transactions on Computers*, Vol. 41, No. 6, pp. 755-768, June 1992.

[3] M. Y. Chan and F. Chin, "Schedulers for larger Classes of Pinwheel Instances," *Algorithmica*, Vol. 9, pp. 425-462, 1993.

[4] S.K. Dhall and C.L. Liu, "On a real-time scheduling problem", *Operations Research*, Vol. 26, pp. 127-140, 1978.

[5] R. Gerber, S. Hong and M. Saksena, "Guaranteeing end-to-end timing constraints by calibrating intermediate processes," *Proc. IEEE Real-Time Systems Symposium*, pp. 192-203, Dec. 1994.

[6] Rhan Ha and Jane W. S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," *Proceedings of the 14th IEEE Int. Conference on Distributed Computing Systems,* June 1994.

[7] C.-C. Han and K. J. Lin, "Scheduling Distance-Constrained Real-Time Tasks," *Proc. IEEE Real-Time Systems Symposium*, Dec. 1992.

[8] C.-C. Han and K. G. Shin, "A Polynomial Time Optimal Synchronous Bandwidth Allocation for the Time-token MAC Protocol," *Proc. IEEE INFOCOM'95*, April 1995.

[9] C.-C. Han and K. G. Shin, "Real-Time Communication in FieldBus Multiaccess Network," *Proc. IEEE Real-Time Technology and Applications Symposium,* Chicago, Illinois, May 1995.

[10] C. Hsueh, K.J. Lin, and N. Fan, "Distributed Pinwheel Scheduling with End-to-End Timing Constraints," *Proc. IEEE Real-Time Systems Symposium,* Pisa, Italy, Dec. 1995.

[11] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel, "The Pinwheel: A Real-Time Scheduling Problem," *Proc. of the 22nd Hawaii International Conference on System Science*, pp. 693-702, January 1989.

[12] H. Kopetz, "A Solution to an Automatic Control System Benchmark," *Proc. IEEE Real-Time Systems Symposium*, pp. 154-158, Dec. 1994.

[13] H. Kopetz and G. Grunsteidl, "TTP–A Protocol for Fault-Tolerant Real-Time Systems," *IEEE Computer*, pp. 14-23, January 1994.

[14] C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, pp. 46-61, Vol. 10, No. 1, 1973.

[15] K. Ramamritham and J. A. Stankovic, "Dynamic task scheduling in hard real-time distributed systems", *IEEE Software*, July 1984.

[16] L. Sha, J. Lehoczky and R. Rajkumar, "Solutions for some Practical Problems in Prioritized Preemptive Scheduling," *Proc. of 7th Real-Time Sys. Symp.*, pp. 181-191, Dec. 1986.

[17] L. Sha and S.S. Sathaye, "A Systematic Approach to Designing Distributed Real-Time Systems," *IEEE Computer,* Vol. 26, No. 9, pp. 68-78, Sep 1993.

[18] Jun Sun, Riccardo Bettati, and Jane W.-S. Liu. "An end-to-end approach to schedule tasks with shared resources in multiprocessor systems," Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, Seattle, Washington, May 1994.

[19] W. Zhao, K. Ramamritham and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constranints", *IEEE Trans. on Computer*, Vol. C-36, No. 8, pp. 949-960, Aug. 1987.

# A Definitions and Algorithms of SLINE operations

In this appendix, we present the detailed definitions and algorithms of SLINE operations. $h_j$ is the relative phase on nodes $N_j$ and $N_{j+1}$ as defined in Section 5.1. Each SLINE $W$ corresponds to a serrated line function $W(h_j)$ defined in X-domain $[0, w)$. In our algorithms, we repeatedly reduce the domain size of a serrated line but keep the minimum value of drop point. To keep track of the original $h_j$ value, each point in our representation has two domains: the current domain and the original domain.

The current domain is the X-domain of the serrated line function $W$ represents. The original domain is the original X-domain of each point before any CUT operation. All points used in our discussion are in the current domain unless we specify explicitly. Sometimes, we use $t$ as a shorthand of $h_j$.

*Definition:* An SLINE $W = \{s, w, (x, y), \{d_1, d_2, \ldots\}\}$ where

1. $s$ is the *slope*, which is the slope of the line segments of the serrated line it represents;

2. $w$ is the current domain size;

3. $(x, y)$ is the origin, where $x$ is in the original domain;

4. The last field is a set of drop points. Each drop point $d_i$ has three fields, $(p, a, p')$; $p$ is the X-coordinate of the drop point in the current domain, $a$ is the drop amount of the value at the drop point, and $p'$ is the X-coordinate of the drop point in the original domain. The drop points are sorted in increasing order on $p$.

5. $W(t)$ returns the value of $W$ at $t$.

6. $W'(t)$ returns the X-coordinate in the original domain of $W$ at $t$.

□

Given a $W(t)$ and a $W'(t)$, we can define a $W$. Suppose $F_1, \ldots, F_u$ are delay functions of $t$ with period $w$. From the definition of delay function in Equation (3), they are serrated line functions and have slope equal to 1.

*Definition:* Three operations are defined for SLINE: SUM, MAX and CUT.

1. SUM$\{W\} = W$ and MAX$\{W\} = W$.

2. SUM$\{W, F_1, \ldots, F_u\}$ is an SLINE, $S$, where $S(t) = W(t) + F_1(W'(t)) + F_2(W'(t)) + \ldots + F_u(W'(t))$ and $S'(t) = W'(t)$, $\forall t \in [0, w)$.

3. MAX$\{W, F_1, \ldots, F_u\}$ is an SLINE, $S$, where $S(t) = \max\{W(t), F_1(W'(t)), F_2(W'(t)), \ldots, F_u(W'(t))\}$ and $S'(t) = W'(t)$, $\forall t \in [0, w)$.

4. Let $u = w/E_k$, $E_k|w$, and $V_{E_k}(t)$ is a function of $t$ to a set of $u$ values, where $V_{E_k}(t) = \{t' \mid t' = t + (v-1) \cdot E_k, 1 \leq v \leq u\}, \forall t \in [0, E_k)$;

   CUT$\{W, E_k\}$ is an SLINE, $S$, where $S(t) = \min\{W(v) \mid v \in V_{E_k}(t)\}$, and $S'(t) = \min\{W'(v) \mid W(v) = S(t)\}, \forall t \in [0, E_k)$.

$\square$

Suppose $S$ is an SLINE with $v$ drop points and domain size $E_k$, and there are $u$ periodic delay functions, $F_1$,...,$F_u$, with the same period $E_k$. Because the period of $F_i$ is $E_k$ for every $F_i$, there is a $p_i \in [0, E_k)$ s.t. $F_i(p_i) = 0$. W.l.o.g. suppose the $p_i$'s are in increasing order.

From the definition of SLINE, $d_i$ are sorted on $d_i.p$ in increasing order. For convenience, to insert, delete or append drop points on an SLINE $S$ will not update the original indices of the drop points unless we specify "**reindex** $S$". During the operations of SLINE, we may generate drop points with the same $p$ value. Before we return $S$, we can reduce these drop points to one drop point, called **compact**. For example, if there are two drop points $d_i$ and $d_j$, $i < j$ and $d_i.p = d_j.p$. After compacting, only $d_i$ will be kept while $d_i.a$ changes to $d_i.a + d_j.a$ and $d_i.p'$ changes to $\min\{d_i.p', d_j.p'\}$.

### A.1 Algorithm: Finding the minimum of $S(h_j)$

```
d_0.p = 0;
h_j = x;                        // Set h_j to the current phase in original domain
minimum = value = y;            // value = S(h_j)
for i = 1 to v do {
   value = value + (d_i.p - d_{i-1}.p) * s - d_i.a;
   if (value < minimum) {
      h_j = d_i.p';
      minimum = value;
   }
}
return minimum, h_j;
```

Since $S(h_j)$ is a serrated line, we only need to check the drop points to get the minimum function value. Moreover, since the slope is $s$ between any two drop points, we can use the formula on line 5 to get the function value at each drop point. It is obvious the complexity of the algorthm is $O(v)$.

## A.2 Algorithm: SUM$\{S, F_1, \ldots, F_u\}$

1. $s = s + u$; // *slope* of $S$ increases $u$ for $u$ functions added.
2. $X = x$;
3. $j = 1$;
4. **for** $i = 1$ **to** $u$ **do** { // Generate and merge new drop points from $F_i$ into $S$.
5.   **while** $(d_j.p \leq p_i)$ **and** $(j \leq v)$ **do** {
6.     $X = d_j.p' - d_j.p + p_i$; // $X$ keep track of the X-coordinate in the original domain.
7.     $j = j + 1$;
8.   }
9.   $y = y + F_i(0)$; // Add up $y$ of the origin.
10.   **if** $(j \leq v)$, insert $(p_i, E_k, X)$ before $d_j$;
11.   **else** append $(p_i, E_k, X)$ after $d_v$;
12. }
13. **reindex** $S$;
14. **return compact** $S$;

By definition, the domain size and the $x$ value of the origin will not change under the SUM operation. Because $S(h_j), F_1(h_j), \ldots, F_u(h_j)$ are serrated line functions and the slopes of each linesegment in a serrated line are the same, the slope of $S$ after this summation is the sum of $s$ and the slope of linesegments on each $F_i$. Since the slope of all linesegments in every $F_i$ is 1, the slope of resulting $S$ becomes $s + u$. The summation doesn't change the drop amount at any drop points, but every $F_i$ will contribute one and only one new drop point because the period of every $F_i$ is the same as the domain size of $S$, $E_k$. So we just need to insert the new drop points into $S$.

On line 6, we assign $p'$ of every new drop point to be the same as the X-coordinate in the original domain of the point with same X-coordinate. The summation also changes the $y$ value of the origin of $S$. Each $F_i$ adds up $F_i(0)$ to $y$. It is reflected on line 9. Including the **for** loop on line 4 and **while** loop on line 5, there will be $u + v$ iterations. Because $p_i$'s are in increasing order, the loop will merge the sorted $u$ new drop points into $S$ and keep the drop points of the resulting $S$ in order. So, the complexity is $O(v + u)$. In general, since the complexity of sorting the new drop points while $p_i$'s are not in increasing order must be included, the complexity of SUM$\{S, F_1, \ldots, F_u\}$ is $O(v + u \lg u)$.

## A.3 Algorithm: MAX$\{S, F_1, \ldots, F_u\}$

```
1.  X = x;
2.  j = 1;
3.  for i = 1 to u do {                         // Generate new drop points from F_i.
4.     while ((d_j.p ≤ p_i) and (j ≤ v)) do {
6.        X = d_j.p' − d_j.p + p_i;             // X keep track of the X-coordinate
6.        j = j + 1;                            // in the original domain.
7.     }
8.     D_i = (p_i, F_i(0), X);                  // D_i is a new drop point
9.  }
10. sort D_i's into D'_j's, s.t. D'_j.a's are in deccreasing order.
11. d_0.a = y;
12. y = max{y, D'_1.a};                         // Update the origin.
13. for j = 1 to v do d_j.a = d_{j−1}.a − d_j.a;       // Together with line 14, store the height of
14. for j = v + 1 downto 1 do d_j.a = d_{j−1}.a;       // each linesegment into drop amount.
15. lowest = max{d_{v+1}.a, D'_1.a − E_k};      // lowest is the height of the lowest linesegment
16. j = 1;                                      // which will be picked.
17. for i = 1 to u do {                         // Merge D'_i's into S.
18.    while ((d_j.a > D'_i.a) and (j ≤ v)) do j = j + 1;
19.    if (j ≤ v), insert D'_i before d_j;
20.    else append D'_i after d_v;
21. }
22. reindex S;
23. j = 1;
24. for i = 1 to u + v do                       // Reconstruct drop points.
25.    if ((d_{i+1}.a > lowest) and (d_{i+1}.p > d_j.p)) do {
26.       d_j.a = d_j.a − d_{i+1}.a;            // Reset drop amount.
27.       j = i + 1;
28.    }
29.    else delete d_i;                         // Delete unnecessary drop points.
30. d_j.a = lowest − d_j.a;                     // Reset drop amount of last drop point.
31. reindex S;
32. return compact S;
```

---

Because $S(h_j), F_1(h_j), \ldots, F_u(h_j)$ are serrated line functions, the function produced by the MAX operation is still a serrated line. The slope, domain size, and $x$ value of the origin will not change. This algorithm is similar to the SUM algorithm, merging the new drop points into $S$. But we also need to remove some drop points which no longer exist in the resulting function.

Since the result is a serrated line composed of linesegments, we only need to pick the linesegments which contains the maximum function values. To know which linesegment contains the maximum function values, we need to compare the linesegments. Since the linesegments are parallel, with the same slope, we define the *height* of a linesegment in a serrated line to be the $y$ value of the intersection of the entension of linesegment and the line $X = 0$. So, consider some linesegments within a common X-domain, the linesegment with the maximum function values has the largest height (the highest).

Once we pick up the highest linesegment, the drop points within the X-domain under this linesegment can be removed.

Therefore, we first sort the linesegments based on their heights, and for each section of X-domain between drop points, pick up the highest linesegments to get the linesegments with the maximum function values. Moreover, the new drop amount of the drop points is the difference of the heights between two neighboring linesegments.

On each serrated line function $F_i$, there are only two linesegments, higher one ($height = F_i(0)$) and lower one ($height = F_i(0) - E_k$). Because the drop amounts of the drop points from $F_i$'s are all the same, $E_k$, the higher ones have the same order with the lower ones. Thus, the height of the highest lower linesegment among $F_i$'s is the height of the highest higher linesegment minus $E_k$.

On line 1 to 9, we define the new drop points, keep track the drop points in original domain in $X$, and, for convenience, store the heights of higher linesegments of $F_i$'s in the field of drop amount temporarily. Line 10 sorts the new drop points on its height in decreasing order. The *lowest* on line 13 is the height of the lowest linesegment which will be picked because it is the higher one of the highest lower linesegment from $F_i$'s and the lowest linesegment from $S(h_j)$. Line 12 updates the $y$ value of origin.

On $S$, there are $v + 1$ linesegments. Lines 13 and 14 store the heights of linesegments in the drop amount of $S$ in decreasing order. At this moment, the drop amount stores the height of the linesegment right before the drop point. (Note that $d_{v+1}.a$ stores the height of lowest linesegment. This value is to be used for removing the last drop point if necessary.) On line 15 *lowest* is the height of lowest linesegment which will be picked.

Since the heights are in decreasing order, the loop from line 17 to 21 is to merge the sorted $u$ new drop points into $S$. Line 24 to 29 delete the unnecessary drop points and reset the drop amounts. Line 30 updates the last drop amount.

The complexity of MAX$\{S, F_1, \ldots, F_u\}$ is $O(v + u \lg u)$.

### A.4 Algorithm: CUT$\{S, E_w\}$

```
w = E_w;
origin_x = 0.0;                 // origin_x is the X-coordinate of origin of
height = y;                     // the sub-SLINE in the current domain of S.
for  i = 1 to  v do {           // Transform d_i to the sub-domain.
    d_i.p = d_i.p − origin_x;
```

```
    if (d_i.p ≥ E_w) {                              // The linesegment before d_i crosses the sub-domain.
      t = d_i.p mod E_w;
      origin_x = origin_x + d_i.p − t;              // Move to the sub-domain with d_i.
      height = height + s · (d_i.p − t) − d_i.a;
      if (t = 0.0) {                                // d_i is on the margin of a sub-domain.
        if (y > height) {
          x = d_i.p′; y = height;                   // Keep track of the origin of the sub-SLINE.
        }
        delete d_i;
      }
      else {
        if (y > height + d_i.a) {
          x = d_i.p′ − t; y = height + d_i.a;       // Keep track of the origin of the sub-SLINE.
        }
        d_i.p = t; d_i.a = height;                  // Store the height of linesegment after d_i into d_i.a
      }                                             // and adjust d_i.p to the sub-domain.
    }
    else {
      height = height − d_i.a;
      d_i.a = height;                               // Store the height of linesegment after d_i into d_i.a.
    }
  }
  if ((d_v.p + origin_x) < (E_k − E_w)) {           // Last linesegment crosses a sub-domain.
    height = height + E_w · s;
    if (y > height) {
      x = d_v.p′ − d_v.p + E_w; y = height; }       // Keep track of the origin of the sub-SLINE.
  }
  reindex S; mergesort the drop points on d_i.a;
  reindex S;
  j = number of current drop points in S;
  for  i = j downto 1 do                            // Reconstruct drop points.
    if ((d_{i−1}.a < y) and (d_{i−1}.p < d_j.p)) {
      d_j.a = d_{i−1}.a − d_j.a;                    // Reset drop amount.
      j = i − 1; }
    else delete d_i;                                // Delete unnecessary drop points.
  d_j.a = y − d_j.a;
  reindex S; return compact S;
```

The operation of CUT$\{S, E_w\}$ divides $S$ into $E_k/E_w$ pieces of smaller SLINE's (sub-SLINE's) with the same domain size $E_w$. Each sub-SLINE has a sub-domain, $[0, E_w)$. For each point $t$ in the sub-domain, the operation picks the minimum value among all the sub-SLINE's at $t$. Conceptually, the serrated line is cut and stacked, and only the minimum line segments are kept. We define *height* as in Section A.3. The operation is exactly as picking the lowest linesegments between drop points in a portion of a sub-domain among all the sub-SLINE's. So we need to transform the drop points into the sub-domain, assign the heights, sort the drop points on height, pick the lowest linesegments, and, at last, reconstruct the drop points to form a new serrated line.

Different from MAX, we are interested in the lower linesegments here. So, if a linesegment crosses the sub-domain, we just consider the first half of the linesegment because the portion across the margin of the sub-domain must be the highest linesegment on next sub-SLINE. Also, if a linesegment crosses a sub-domain, it is the highest (also the only) one in the sub-domain. So we don't need to take the portion of linesegment in the sub-domain into account. However, we need to consider the lowest highest height among the different sub-domains. Actually, it is the new $y$ value of the origin. In our algorithm, we cut $S$ by transforming $d_i$ into a sub-domain and ignoring the portion of linesegment across any sub-domain. In addition to a special-case check, which is the last linesegment of $S$ crosses a sub-domain, we can determine the $y$ value in the transformation process. We can thus ignore the portion of linesegment across a sub-domain and all the linesegments needed to be considered can be determined by origin and drop points; we temporarily stores the height of a linesegment in the drop amount of the drop point at the beginning of the linesegment.

Moreover, if there is any drop point which falls right on the margin of the sub-domain, we can remove it. Because the linesegment after the margin is the highest in its sub-domain and the linesegment before the margin is considered by previous drop point. Therefore, the number of drop points (also the heights) needed to be sorted is less than $v$.

In the algorithm, line 1 changes the domain size to $E_w$. Line 4 to 27 transform the drop points in $S$ into a sub-domain. Line 11 to 13, 17 to 19, and 30 to 31 keep the lowest origin of the sub-domain, with the $x$ of the origin in the original domain. On lin 28, it deals with the special case when the last linesegment crosses a sub-domain.

After the transformation, the heights in each sub-domain are already sorted, so we use mergesort to speed the sorting. Since there are $E_k/E_w$ sub-domains and the number of drop points to be sorted is less than $v$, the complexity for the mergesort on line 33 is $O(v \lg(E_k/E_w))$. Line 36 to 40 delete the unnecessary drop points and reset the drop amounts.

The complexity of CUT$\{S, E_w\}$ is $O(v \lg(E_k/E_w))$.