In *The Computer Journal* (British Computer Society), Vol. 38, No. 4, 1995.

Distributed Pipeline Scheduling: A Framework for Distributed, Heterogeneous Real-Time System Design

Saurav Chatterjee and Jay Strosnider
Department of Electrical & Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA
saurav@ece.cmu.edu

Abstract

This paper describes the Distributed Pipeline Scheduling Framework that provides a systematic approach to designing distributed, heterogeneous real-time systems. This paper formalizes Distributed Pipelining Scheduling by providing a set of abstractions and transformations to map real-time applications to system resources, to create highly efficient and predictable systems, and to de-compose the very complex multi-resource system timing analysis problem into a set of simpler application stream and single resource schedulability problems to ascertain that all real-time application timing requirements are met. Distributed Pipeline Scheduling includes support for distributed, heterogeneous system resources and diverse local scheduling policies, global scheduling policies for efficient resource utilization, flow-control mechanisms for predictable system behavior, and a range of system reconfiguration options to meet application timing requirements. An audio/video example is used in this paper to demonstrate the power and utility of Distributed Pipeline Scheduling.

1. Introduction

It is not uncommon for serious performance problems to arise in the development of distributed, heterogeneous real-time systems. Part of the problem has been due to the lack of a systematic approach for designing these systems. Discrete-event simulation and timeline construction approaches tend to scale poorly, with the complexity of the model approaching the complexity of the system under development. These models become unwieldy, become insufficiently maintained and are then discarded. As a result, the performance properties of the final system diverges greatly from initial goals.

This research was supported in part by a grant from the Office of Naval Research, in part by a grant from the Naval Research and Development Laboratory, and in part by a grant from Siemens Corporate Research.

This paper presents a framework, denoted the *Distributed Pipeline Scheduling Framework*, for designing distributed, heterogeneous real-time systems that execute in a pipelined, efficient and predictable manner. Furthermore, analysis techniques presented in the Framework de-compose the very complex multi-resource system timing analysis problem into a set of simpler single resource, single application stream problems that can accurately predict the performance properties of the resulting system design and can be used to determine that all real-time application timing requirements are met. If application timing requirements are not met, the Framework also delineates how manipulating system configuration parameters can attempt to meet the timing requirements.

Figure 1 provides an overview of the Distributed Pipeline Scheduling Framework. Each Logical Application Stream Model (LASM) and Optional Parameters list (OP) capture a real-time application's timing and processing requirements. The LASM is composed of a set of precedence-constraint processing steps. Each *processing step* is an unit of execution on any resource, *e.g.*, a thread is a processing step executing on a processor-type resource. The Target Platform Model (TPM) represents a connected set of system resources, each of which can be found in the Library of Single Resource Scheduling Models.

The Framework first maps each LASM onto the TPM. The mapping algorithm allocates resources to processing steps and adds new I/O processing steps to delineate communication paths. The result is a set of target-specific application streams (TSAS).

The Framework next transforms each TSAS into a distributed pipeline by partitioning groups of processing steps into distributed pipeline stages and assigning initial pipeline stage attributes. Distributed pipelines are represented using a set of Pipelined Application Streams (PAS). Distributed pipelines utilize system resources efficiently and execute in very predictable patterns.

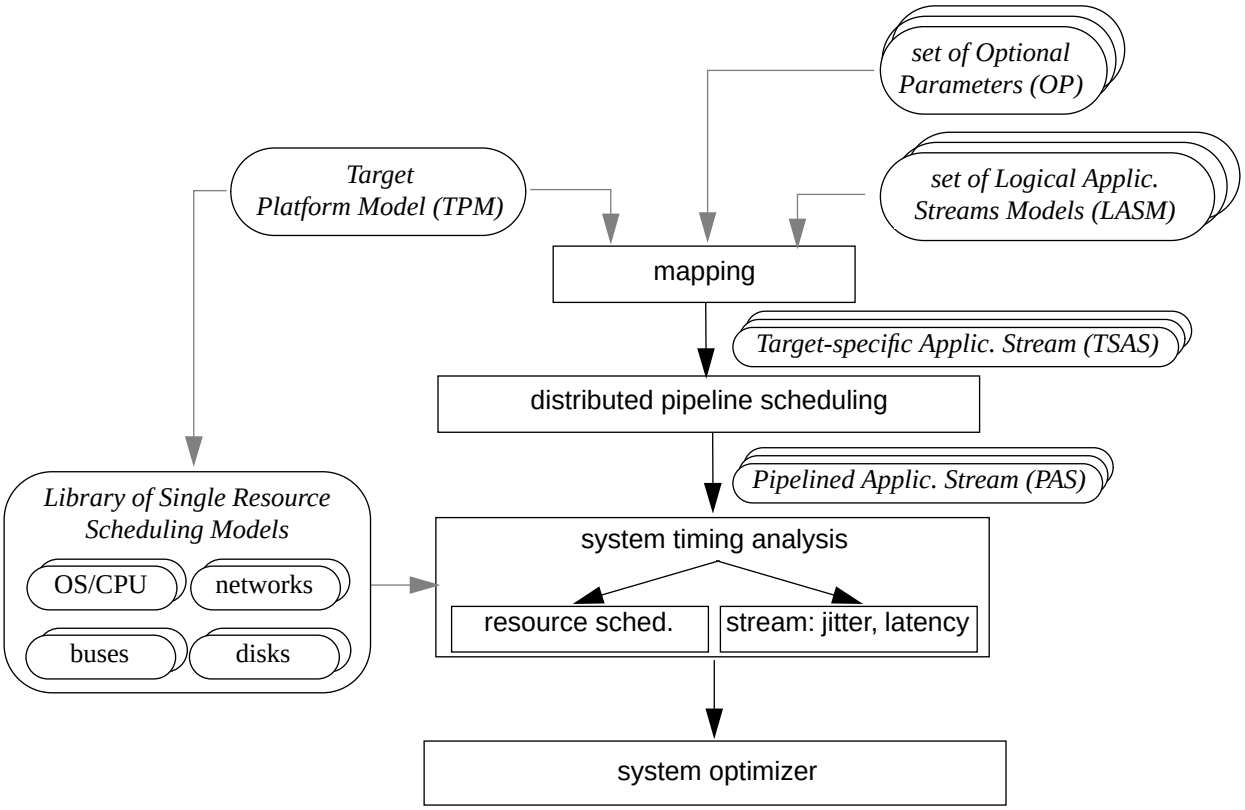


Figure 1: **Distributed Pipeline Scheduling Framework for Design of Highly Efficient and Predictable Distributed, Heterogeneous Real-time Systems**

Next, based on pipeline stage attributes assignments, system timing analysis determines whether all application timing requirements have been met. Distributed pipeline scheduling effectively transforms a very complex multi-resource timing check into sets of simpler (*i*) distributed pipeline end-to-end latency and jitter checks and (*ii*) single resource schedulability checks. This hierarchical strategy enables the Framework to *uniquely* analyze an arbitrary set of system resources, *e.g.*, real-time operating systems/CPUs, backplane buses, disks, and networks, and an arbitrary set of local (resource-specific) scheduling policies, *e.g.*, rate-monotonic, earliest-deadline first.

The system optimizer delineates how manipulating various system configuration parameters affect application timing requirements. Therefore, based on initial system timing analysis and an understanding of how system configuration parameters impact application timing requirements, distributed pipeline stage attributes and resource parameters can be re-configured to meet all application requirements.

The remainder of the paper describes each part of the framework in more detail. Section 2 describes the input to the framework and introduces an example that will be used for illustration purposes. Section 3 provides a set of abstractions and transformation rules that systematically map LASMs to the target platform. Section 4 details the key ideas in distributed pipeline scheduling that result in highly efficient and predictable systems. Section 5 demonstrates how Distributed Pipeline Scheduling de-composes a very complex timing analysis problem into a set of simpler timing analysis problems. Section 6 illustrates how to re-configure distributed pipeline stage attributes based on information learned during timing analysis. Section 7 discusses related work. Section 8 summarizes this paper and explores our current research thrusts.

The following notation is used in this paper. Each attribute variable is the acronym of the full attribute name, *e.g.*, *SL* is stream latency. Furthermore, all application stream, resource and processing step-specific attributes start with *S*, *R* and *P*, respectively. Attribute superscripts denote either a particular resource or a particular application stream, e.g., SL^k is the k^{th} stream latency. Subscripts are only used for processing step attributes to denote the processing step within an application stream, *e.g.*, PRN_i^k denotes an attribute for the i^{th} processing step within stream *k*.

2. Input Specifications

This section provides formal representations of a logical application stream model and a physical target platform model and illustrates how to model an example multi-media system using these representations.

2.1 Modeling Heterogeneous Resources

The *target platform model* (TPM) captures the timing and topological properties of a connected set of physical resources. The target platform is represented using a directed graph, $TPM = \{\Re, \Xi\}$. The set of vertices $\Re$ correspond to system resources and the set of graph edges Ξ specify connectivity between resources. Given a total of R resources in the system, each resource $r \quad (1 \le r \le R)$ is characterized by a set of attributes shown in Table 1. Note that all resource attributes start with 'R.'

RN	resource name
RT	resource type
RL	set of processing steps allocated to resource
RP	set of resource-specific parameters
RS	resource-specific scheduling model stub

Table 1: Resource Attributes

The resource type, *e.g.*, x486 CPU or FDDI network, is used for mapping purposes. Initially, a resource has no load, *i.e.*, $RL^r = \varnothing$. However, as applications are mapped to the system, RL^r contains a list of processing steps allocated to resource *r*. Table 2 shows a list of resource-specific parameters required for mapping and for processing step attributes assignments.

Resource Type	Parameter	Notation
CPU	Cycles per Instruc. Clock Rate Buffer Size	RCI^r RCR^r RBS^r
network	Transfer Rate Packet Size Propagation Delay Buffer Size	RTR^r RPS^r RPD^r RBS^r
disk	Transfer Rate Sector Size Buffer Size	RTR^r RSS^r RBS^r

Table 2: Resource Parameter Attributes

The scheduling model stub, *RS*, provides a reference to the Library of Single Resource Scheduling Models. The Library provides a set of resource-specific scheduling models describing the timing and concurrency properties of each individual resource available in the Target Platform. The scheduling models will be used in Section 5 for schedulability analysis. Scheduling models for OS/CPUs [13], networks [17], buses [18] and disks [6] are available. These models represent a variety of local resource scheduling policies such as rate-monotonic (RMS), earliest-deadline first (EDF), first-in, first-out (FIFO) and round-robin (RR).

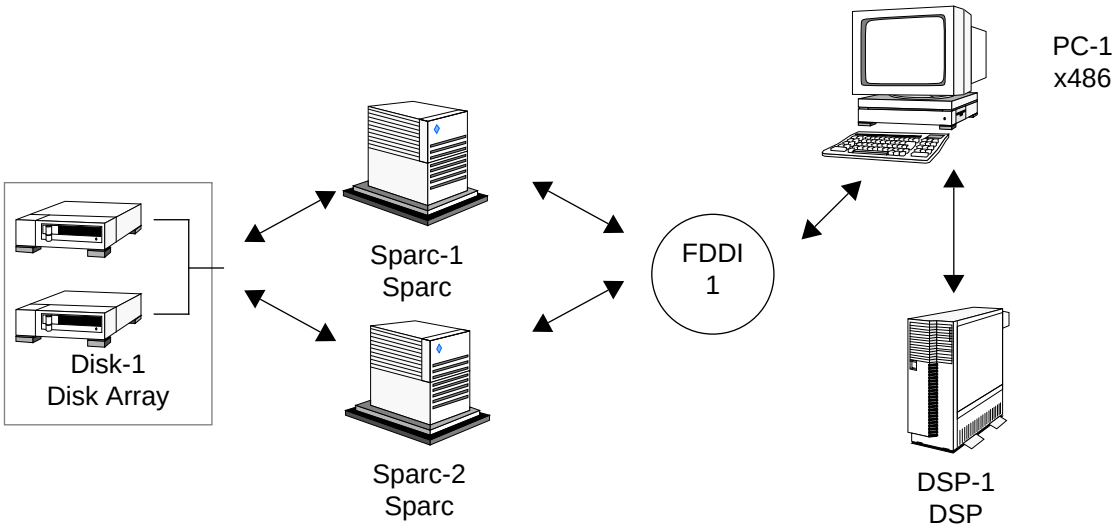
Figure 2 illustrates a simple target platform model for an example audio/video application. Buses were excluded from the target platform for simplicity. The scheduling models in Figure 2b capture the timing characteristics and the desired local scheduling policy of each system resource. The overhead attribute shown in the Figure indicates the context-switch times for OS/CPUs, seek overhead for the disk, and packet overhead for the network. Figure 2c shows an example fixed-priority, OS/CPU scheduling model that may be found in the Library of Single Resource Scheduling Models.

2.2 Modeling Real-Time Applications

A *Logical Application Stream Model* (LASM) captures the *logical, i.e.,* target-platform and I/O resource independent, timing and processing requirements of a periodic real-time application. Given a total of *S* application streams, each application stream $k \quad (1 \le k \le S)$ is characterized using the attributes shown in Table 3. All stream attributes, except for processing steps, begin with the prefix 'S.' Stream latency is the maximum end-to-end latency requirement from when the first processing step becomes eligible for execution until when the last

SL^k	max. end-to-end Stream Latency requirement
$LPS^k = \{LPS_1^k, \ldots, LPS_{g(k)}^k\}$	set of $g(k)$ precedence-constrained Logical Processing Steps
$SIR^k = \{SIR_{min}^k, SIR_{max}^k\}$	min. and max. Stream Input Rates
$SOR^k = \{SOR_{min}^k, SOR_{max}^k\}$	min. and max. Stream Output Rates
$\{J_{in}^k, J_{out}^k\}$	Input and Output Jitter Bounds

Table 3: LASM Attributes



(a) Resource Topology

Name:	PC-1	DSP-1	Disk-1	Sparc-1	Sparc-2	FDDI 1
Type:	x486	DSP	Disk Array	Sparc	Sparc	FDDI
Sched. Model:	RTMach/ RMS	VCOS/ RMS	Disk/ RMS	RTMach/ RMS	RTMach/ RMS	FDDI/ FIFO
Clock Rate:	40 MHz	40 MHz		40 MHz	40 MHz	
CPI	1	1		1	1	
Sector Size			1024 bytes			
Packet Size						512 bytes
Transfer Rate			270 Mb/ sec			200 Mb/ sec
Overhead	30 us	30 us	40 us	30 us	30 us	10 bytes
Buffer Size	4 MB	4 MB		4 MB	4 MB	

(b) Resource Properties

$$S_i^r = min_{0<t\le D_i} \sum_{j=1}^{i} \frac{C_j + Overhead_j}{t} \left\lceil \frac{t}{T_j} \right\rceil + \frac{Overhead_{system}}{t} + \frac{Blocking_i}{t}$$

where

$Overhead_j$	Kernel execution time performing a service on behalf of a user thread
$Blocking_i$	Execution time during which a higher priority thread cannot execute.
$Overhead_{system}$	System overhead not attributed to any one thread.

(c) Generic Non-ideal Scheduling Model

Figure 2: **Example Target Platform Model**

processing step completes execution once. The stream input and output rates specify the rate at which the first and last processing steps execute, respectively. The input and output jitter bounds specify the maximum jitter that the first and last processing steps, respectively, can tolerate. Jitter is defined as the time interval between completion of two successive instances of a processing step.

A LASM is comprised of a sequential set of *g(k) logical* processing steps. Each *logical processing step* is an unit of execution on any resource, *e.g.*, a thread is a processing step executing on a processor-type resource. The *sequential* order specifies *precedence-constraints* between processing steps. Note that each processing step is denoted with the prefix *'L'* to indicate that it is a *logical* processing step. Table 4 lists the attributes for each logical processing step LPS_i^k, $1 \le i \le g(k)$, in the k^{th} LASM.

PC_i^k	Category
$PIDS_i^k$	Input Data Structure
$PIIDS_i^k$	Initial Input Data Structure
$PODS_i^k$	Output Data Structure
$PIPB_i^k$	Generic RISC CPU Instructions per $PIDS_i^k$
PRT_i^k	Resource Type
PCT_i^k	Computation Type

Table 4: Logical Processing Step Attributes

Note that all processing step attributes begin with *P.* There are four processing step categories: source, sink, intermediate and synchronization. The *source* is the left-most processing step, *i.e.*, LPS_1^k. The *sink* is the right-most step, *i.e.*, $LPS_{g(k)}^k$. The remaining steps are denoted *intermediary* or *synchronization* steps. The source step produces the initial data, the intermediary steps filter and process the data, synchronization steps synchronize multiple data streams and the sink consumes the data. This paper does not take into account synchronization steps. Refer to [3] on designing real-time systems using synchronization processing steps.

Processing step LPS_i^k, $1 < i \le g(k)$, executes for the first time after it receives $PIIDS_i^k + PIDS_i^k$ bytes of data; it can execute again after it receives $PIDS_i^k$ bytes of *new* data. The *output data structure* specifies how many bytes of data are produced by the step each time it executes. Either one of the quotients, $PIDS_i^k / PODS_{i-1}^k$ or $PODS_{i-1}^k / PIDS_i^k$, $1 < i \le g(k)$, must be an integer. Processing step LPS_{i-1}^k's output data structure type must match processing step LPS_i^k's input data structure type. The resource type specifies the type of resource on which the processing step can execute.

For processing steps executing on processor-type resources, *i.e.*, threads, the *instructions per byte*, *PIPB*, specify the number of generic (RISC) instructions required to process each input byte in the *PIDS*; this will be used later for converting bytes into CPU execution times. Furthermore, to allow complex thread actions, the *processing step computation type* can be specified as either (*i*) default, (*ii*) grab semaphore lock, or (*iii*) release semaphore lock. This list of thread actions can be expanded as required.

The following processing step attributes are pre-defined as null: $PIDS_1^k$, $PODS_{g(k)}^k$, and $PIIDS_1^k$. $PIPB_i^k$ and PCT_i^k are also pre-defined as null if PRT_i^k is not a processor-type resource.

The example audio/video application suite used in this paper is comprised of two applications streams: audio and video. Figures 3 and 4 illustrate the audio and video LASMs. The audio and video applications each contain four processing steps, a maximum end-to-end latency requirement of 85 msec, no input rate or jitter requirements, and output rates of 8 kHz and 30 Hz, respectively.

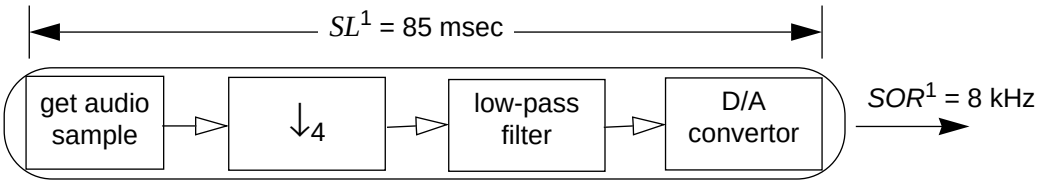
2.3 Optional Parameters List

Given a TPM and a set of LASMs, the Framework will automatically map each LASM onto the TPM and assign several new resource-specific attributes. The Optional Parameters List (OP) provides a method for the designer to override the Framework on specific functions if desired.

The Optional Parameters list specifies target platform-specific attributes for each LASM. The optional parameters are represented using $OP = \{PRN, PET, PER\}$, where $RN^k = \{prn_1, ..., prn_{g(k)}\}$ specifies the exact resource name for each processing step in stream *k*, $PET^k = \{pet_1, ..., pet_{g(k)}\}$ specifies the processing step execution time and $PER^k = \{per_1, ..., per_{g(k)}\}$ specifies the processing step execution rate.

Tables 5 and 6 illustrate one use of the optional parameters list for the audio and video application streams- to bind specific resources to the source and sink processing steps of each logical application stream. This example assumes that Disk-1 contains the desired audio/video data that a client wishes to hear/view and that the client is sitting in front of PC-1/DSP-1.

The OP list can also be used in maintaining a large system. During the initial design phase, the Framework



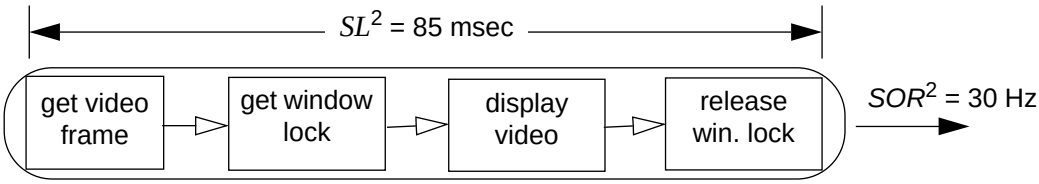
(a) Stream Attributes

Processing Steps:	get audio sample	↓4	low-pass filter	D/A convertor
Resource Type:	Disk	Sparc	DSP	DSP
Input Data Struct:		4 sample	1 sample	1 sample
Output Data Struct:	1 sector	1 samples	1 sample	
Instr./IDS:		20	40	500
Computation Type:		default	default	default

(1 sample = 2 bytes)

(b) Processing Step Attributes

Figure 3: **Audio Logical Application Stream Model**



(a) Stream Attributes

Processing Steps:	get video frame	get window lock	display video	release win. lock
Resource Type:	Disk	x486	x486	x486
Input Data Struct:		1 frame	1 frame	1 frame
Output Data Struct:	1 sector	1 frame	1 frame	
Instr./IDS:		0.01	2	0.01
Computation Type:		grab lock	default	release lock

(1 frame = 360 KBytes)

(b) Processing Step Attributes

Figure 4: **Video Logical Application Stream Model**

Processing Steps:	get audio sample	$\downarrow_4$	low-pass filter	D/A convertor
Resource Name:	Disk-1	∅	∅	DSP-1

Table 5: User-specified Resource Names for Audio Processing Steps

Processing Steps:	get video frame	lock display	display video	release lock
Resource Name:	Disk-1	PC-1	PC-1	PC-1

Table 6: User-specified Resource Names for Video Processing Steps

can calculate estimates of the CPU execution times for all processing steps executing on CPUs. During more advanced design phases, once system resources have been allocated permanently to processing steps, the OP list can be used to replace the estimate execution times with measured values for more exact analysis.

The resource name can be set to $rn_i^k = \varnothing$ if the designer does not wish to *a priori* bind a resource to processing step LPS_i^k of logical application *k*. Instead, the mapping algorithm described next will automatically attempt to allocate system resources to these processing steps.

3. Mapping

Given a target platform model and a set of logical application stream models, the Framework can be used to systematically design a real-time system which meets all application timing requirements and uses all target platform resources efficiently. The first step in this endeavor is mapping. Figure 5 provides an overview of mapping. Mapping allocates system resources to processing steps and adds new I/O resources to delineate communication paths. The resulting processing steps are called *target-specific processing steps*. Next, mapping assigns attributes to these newly created processing steps. Therefore, mapping transforms each LASM into a target-specific application stream (TSAS) comprised of a sequential set of target-specific processing steps.

Mapping attempts to maximize the number of real-time applications whose timing requirements can be met by the system. Unfortunately, the equations used to determine whether application requirements are met by the system cannot be calculated until after several post-mapping steps. Therefore, the mapping goal is to minimize the number of resulting target-specific processing steps per TSAS. Minimizing the number (*i*) maximizes the chance that application latency and jitter requirements will be met and (*ii*) reserves the least number of system resources, thereby maximizing the chance that other application streams can meet their requirements.

Allocation binds processing steps to system resources. A processing step can be bound to any resource of the correct type, *i.e.*, the resource type in the processing step and resource attributes must match. *Routing* finds I/O paths for communication between processing steps. The routing process will make explicit the I/O paths between resources for a given application by introducing additional processing steps for each application stream.

The allocation and routing algorithms attempt to minimize the number of target-specific processing steps per application stream. Application timing

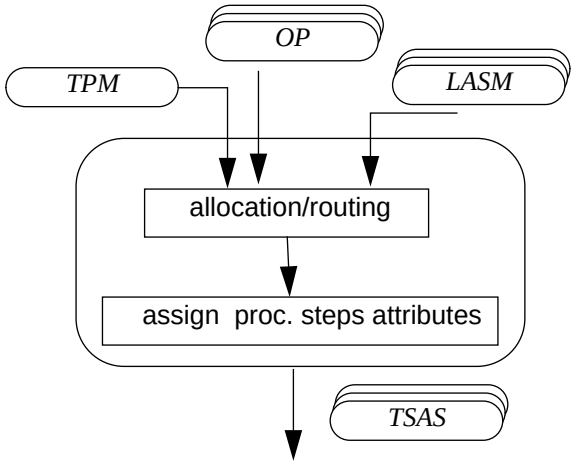


Figure 5: **Overview of Mapping Phase**

SL^k	max. end-to-end Stream Latency requirement
$TSPS^k = \{TSPS_1^k, \ldots, TSPS_{h(k)}^k\}$	set of $h(k)$ precedence-constrained Target-Specific Processing Steps
$SIR^k = \{SIR_{min}^k, SIR_{max}^k\}$	min. and max. Stream Input Rates
$SOR^k = \{SOR_{min}^k, SOR_{max}^k\}$	min. and max. Stream Output Rates
$\{J_{in}^k, J_{out}^k\}$	Input and Output Jitter Bounds

Table 7: Target-Specific Stream Attributes

requirements are not considered. If system timing analysis later determines that the chosen set of system resources cannot meet the application timing requirements, this set will be invalidated and the next minimum-number set will be chosen. The Framework is flexible and does not specify any one algorithm for allocation and routing. Instead, it provides system models, abstractions and cost functions such that CAD algorithms such as bin-packing, simulated annealing or genetic algorithms can be used.

The mapping algorithm expands the original *g(k)* logical processing steps into *h(k)* target-specific processing steps. The *h(k) - g(k)* newly created target-specific processing steps indicate the I/O paths for communication between processing steps. Allocation and routing results for the example audio and video application streams are shown in Figures 6a and 7a. The remainder of this section specifies how to calculate target-specific processing step attributed.

Table 7 lists the parameters used to describe each target-specific application stream (TSAS). Most of these attributes are identical to those for logical application streams. Logical processing steps, however, have been replaced by target-specific processing steps, denoted by the prefix '*TS.*' Table 8 lists the target-specific processing step attributes. The first three attributes are identical to that for logical processing steps. The fourth attribute, computation type, only applies to processor-type resources. The three original computation types have been augmented to include a fourth type: I/O. Target-specific processing steps that were created via the allocation algorithm retain their original logical processing step values for these four attributes. For target-specific processing step $TSPS_i^k$ created via the routing algorithm, the following equations derive values for these four attributes:

$PIDS_i^k$	Input Data Structure
$PIIDS_i^k$	Initial Input Data Structure
$PODS_i^k$	Output Data Structure
PCT_i^k	Computation Type
PRN_i^k	Resource Name
PET_i^k	Execution Time
PER_i^k	Execution Rate

Table 8: Target-Specific Proc. Step Attributes

$$PIDS_i^k = \begin{cases} RPS^r & network \\ RSS^r & disc \\ PODS_{i-1}^k & CPU \end{cases},$$

$$PIIDS_i^k = 0,$$

$$PODS_i^k = PIDS_i^k, \text{ and}$$

$$PCT_i^k = \begin{cases} \text{“I/O“} & PRT_i^k = \text{processor} \\ \varnothing & else \end{cases}$$

where $r = PRN_i^k$. The input data structure equation basically states that a network packet is not transmitted or a disk sector is not written until it is filled. Note that CPUs, when used as I/O, have no such requirements. Also, if desired, $PIDS_i^k$ may be set to $PIDS_{i+1}^k$, for CPUs.

Table 8 also lists three new attributes for all target-specific processing steps. The resource name specifies the resource on which the processing step executes. The execution time specifies how long the processing step executes on the resource. The rate specifies how often the processing step must execute to achieve steady-state behavior with adjacent processing steps. For each target-specific $TSPS_i^k$, the following equations can be used:

$$PET_i^k = \begin{cases} (PIDS_i^k)\,(PIPB_i^k)\,(RCI^r)/RCR^r & \text{processor-type} \\ (PIDS_i^k/RTR^r) & \text{disc/network/bus-types} \end{cases}$$

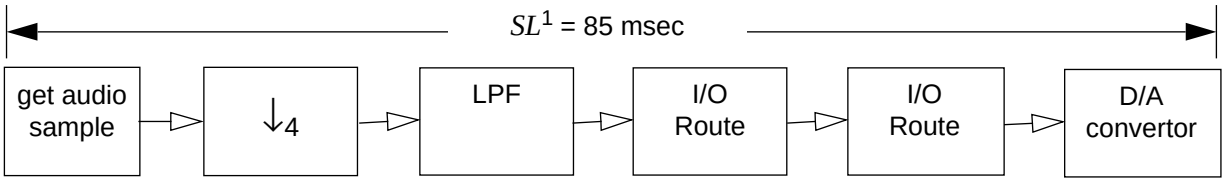
If SIR_{min}^k is defined,

$$PER_i^k = \begin{cases} (PODS_{i-1}^k/PIDS_i^k)\,PER_{i-1}^k & 1 < i \le h(k) \\ SIR_{min}^k & i = 1 \end{cases}$$

If SOR_{min}^k is defined,

$$PER_i^k = \begin{cases} (PIDS_{i+1}^k/PODS_i^k)\,PER_{i+1}^k & 1 \le i < h(k) \\ SOR_{min}^k & i = h(k) \end{cases}$$

where $r = PRN_i^k$. Either equation can be used if both SIR_{min}^k and SOR_{min}^k are defined. These values can be overwritten with values specified by the designer using

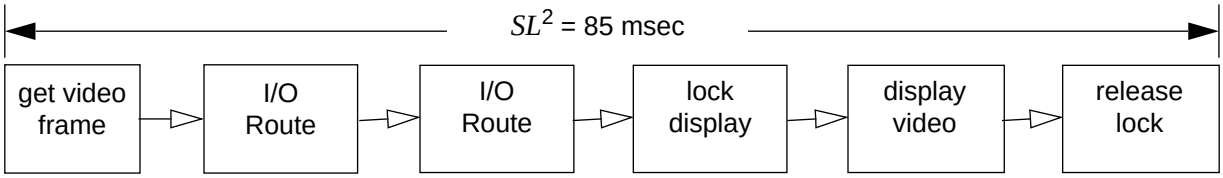


(a) Stream Representation

Processing Steps:	get audio sample	↓4	low-pass filter	I/O Route	I/O Route	D/A convertor
Input Data Str:		8 bytes	2 bytes	512 bytes	2 bytes	2 bytes
Output Data Str:	1 KB	2 bytes	2 bytes	512 bytes	2 bytes	
Resource Name:	Disk-1	Sparc -1	Sparc -1	FDDI 1	PC-1	DSP-1
Exec.Time	30.5 usec	4 usec	2 usec	20.6 usec	20.6 usec	25 usec
Rate:	62.5 Hz	8 kHz	8 kHz	31.25 Hz	8 kHz	8 kHz
Comp. Type:		default	default		I/O	default

(b) Processing Step Attributes

Figure 6: **Audio Target-specific Application Stream**



(a) Stream Representation

Processing Steps:	get video frame	I/O Route	I/O Route	lock display	display video	release lock
Input Data Str:		1 KB	512 bytes	360 KB	360 KB	360 KB
Output Data Str:	1 KB	1 KB	512 bytes	360 KB	360 KB	
Resource Name	Disk-1	Sparc-1	FDDI 1	PC-1	PC-1	PC-1
Exec. Time:	30.5 usec	51.2 usec	20.6 usec	92 usec	18.4 msec	92 usec
Rate:	10.8 kHz	10.8 kHz	21.6 kHz	30 Hz	30 Hz	30 Hz
Comp. Type:		I/O		grab lock	default	rel. lock

(b) Processing Step Attributes

Figure 7: **Video Target-Specific Application Stream**

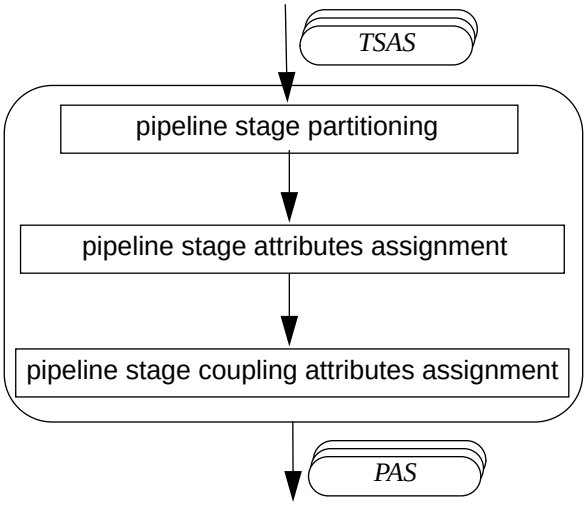


Figure 8: **Distributed Pipeline Scheduling**

the OP list.

Figures 6b and 7b list the initial target-specific processing step attributes for the audio and video application streams. Samples, frames, packets, and sectors in Figures 3 and 4 have been replaced with bytes in Figures 6b and 7b for clarity. Distributed pipeline scheduling next specifies a set of rules such that the processing steps utilize system resources efficiently and execute predictably.

4. Distributed Pipeline Scheduling

Distributed pipeline scheduling transforms a TSAS into a *distributed pipeline* by partitioning groups of target-specific processing steps into distributed pipeline stages and assigning intra- and inter-pipeline stage attributes (Figure 8). Distributed pipelining is similar to processor pipelining. On the other hand, we denote our pipeline as a *distributed pipeline* because distributed pipeline stages are generally asynchronously clocked. As a result, unlike a processor pipeline, a single global clock may not be present to clock every one of the distributed pipeline stages. Also note that unlike processor pipelining, each application stream has its own unique distributed pipeline, albeit executing over shared resources. Furthermore, each distributed pipeline is re-configurable and the number of distributed pipeline stages for each application stream is a function of the logical application stream, the target platform and the current mapping. Finally, all processor pipeline stages execute in lock-step whereas distributed pipeline stages may execute at different times. These differences are summarized in Table 9.

Section 4.1 identifies a set of rules used to partition target-specific processing steps into distributed pipeline stages. Section 4.2 describes distributed pipeline stage attributes. Section 4.3 describes the *coupling* relationships between adjacent pipeline stages within a distributed pipeline.

	Processor Pipeline	**Distributed Pipeline**
Clock	single	multiple
No. of Pipelines	single	one per applic. stream
No. of Stages	fixed	re-configurable
Execution	lock-step	phase-delayed

Table 9: Comparison between Processor and Distributed Pipelining

4.1 Pipeline Stage Partitioning

This section identifies a set of rules to partition a group of target-specific processing steps into distributed pipeline stages such that system resources are utilized efficiently:

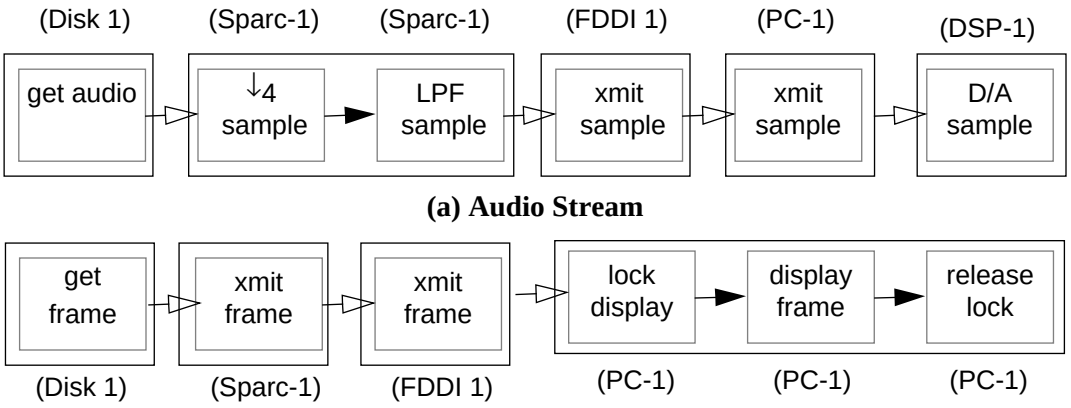
- group successive target-specific processing steps with common rates that have been mapped to a common resource to a single PAS pipeline stage,
- insert a pipeline stage boundary between target-specific processing steps $TSPS_{i-1}^{k}$ and $TSPS_{i}^{k}$ when there is a change in rate, *i.e.*, $PIDS_{i}^{k}/PODS_{i-1}^{k} \neq 1$ or $PIIDS_{i}^{k} \neq 0$,
- insert a pipeline stage boundary between adjacent processing steps that execute on different resources.

The first rule eliminates unnecessary buffering and minimizes latency [11]. The second rule facilitates the efficient utilization of system resources by enabling applications to execute in a pipelined manner. The third rule is an essential feature for the hierarchical system timing analysis, described in the next section, to work. This allows the multi-resource scheduling analysis problem to be decomposed into a set of single stream and single resource schedulability analysis problems. These rules transform the *h(k)* target-specific processing steps into *q(k)* distributed pipeline stages.

Consider the audio and video target-specific application streams introduced in Figures 6a and 7a. Figure 9 shows the corresponding distributed pipeline representation for both application streams. The next two sections assign attributes for the newly created distributed pipeline stages.

4.2 Pipeline Stage Attributes Assignment

Each distributed pipeline is composed of a set of precedence-constrained pipeline stages. Distributed pipeline attributes are similar to those for TSAS and



(solid line: pipeline stage boundary; dashed line: target-specific proc. step boundary)
(hollow arrow: flow-controlled message flow; solid arrow: data-flow messages)

Figure 9: **Distributed Pipeline Representation of the Audio/Video TSAS**

are therefore not shown. Table 10 lists the attributes of each distributed pipeline stage.

The initial assignment of pipeline stage attributes for each application stream is done without explicitly considering the impact of the individual pipeline stage attributes on the application timing requirements. Rather, initial pipeline stage attributes are assigned based upon the following two *assignment rules* which attempt to find the distributed pipeline's natural latency, jitter and buffer characteristics as functions of system resource configuration parameters and application timing specifications:

- initialize all non-I/O pipeline stage periods in relation to the input data structure requirements of the first processing step within the stage,
- initialize I/O pipeline stage periods consistent with the normalized packet periods that would result from moving the data in maximum sized packets across the I/O media.

$PIDS_i^k$	Input Data Structure
$PIIDS_i^k$	Initial Input Data Structure
$PODS_i^k$	Output Data Structure
PRN_i^k	Resource Name
$C_i^k = \{c_{i1}^k, ..., c_{im}^k\}$	set of Execution times corresponding to the m processing steps within pipeline stage i
T_i^k	Period
D_i^k	Local Deadline
P_i^k	Propagation Delay

Table 10: Dist. Pipeline Stage Attributes

The maximum packet size is a configuration parameter for each I/O resource. The normalized packet period rule minimizes required buffering and eliminates unnecessary latency. This approach is necessary for the transfer of large data structures across multi-hop networks. For example, it is not advisable to buffer a full video frame at intermediate stages of a multi-hop network. Note that these system parameters are configurable in the optimization step of the Distributed Pipelining Scheduling Framework.

Because multiple target-specific processing steps may comprise a single distributed pipeline stage, each pipeline stage contains a set of execution times, corresponding to the execution time of each of these processing steps. Since all processing steps within a pipeline stage execute at the same rate, a single period suffices. Furthermore, a single pipeline stage deadline, D_i^k, specifies by what time must all processing steps within a pipeline stage complete their executions. In reality, due to the processing step precedence-constraints, this is identical to stating by what time must the last processing step within the pipeline stage complete its execution. The pipeline stage inherits all other attributes from the TSAS.

If target-specific processing steps b through $b+m$ were partitioned into a single pipeline stage i, the pipeline stage has the following attributes:

$$c_{ij}^k = PET_{j+b-1}^k \quad (1 \le j \le m)$$

$$T_i^k = 1/PER_b^k$$

$$D_i^k = T_i^k$$

$$P_i^k = \begin{cases} RPD^r & PRT_i^k = network \\ 0 & else \end{cases},$$

where $r = PRN_i^k$.

The execution, period, deadline and worst-case propagation delay attributes of each distributed pipeline stage for the audio and video application streams are shown in Table 12.

The set of execution times, periods and deadlines of pipeline stages from different distributed pipelines executing on a shared system resource fully characterizes the resource's "load characteristics." Pipeline stage periods, deadlines, propagation delays, as well as several key coupling attributes introduced in the next section, fully characterize the "stream characteristics" of each distributed pipeline. Section 5 ascertains whether all application timing requirements are met based on these load and stream characteristics.

4.3 Pipeline Stage Coupling Attributes

Section 4.2 assigned individual pipeline stage attributes. This section assigns coupling attributes between every pair of adjacent pipeline stages. Table 11 lists the two coupling attributes - whether the two pipeline stages execute in lock-step or are phase-delayed and the type of flow-control.

θ_i^k	Phase Delay between two the i^{th} and i+1^{th} pipeline stages
F_i^k	form of Flow control between the i^{th} and i+1^{th} pipeline stages

Table 11: Dist. Pipeline Stage Coupling Attributes

In an ideal environment with negligible propagation delays and perfect clock synchronization, all distributed pipeline stages can execute in lock-step. In non-ideal environments, however, manipulating the *phase delay* between the completion of one pipeline stage's execution and the commencement of the next pipeline stage's execution is one technique to meet application timing requirements [4]. This paper assumes the following phase delay between two pipeline stages i and i+1, $1 \le i < q(k)$:

$$\theta_i^k = P_i^k + max(0, \Phi_i^k)$$

where $\Phi_i^k = Clock^p - Clock^r$ specifies the signed *worst-case* clock skew between resources $r = PRN_i^k$ and $p = PRN_{i+1}^k$. This setting guarantees that the data from pipeline stage i will reach pipeline stage i+1 before it is made eligible for execution.

Based on Equation , the audio and video distributed pipelines specify a θ_i^k equal to zero between all pipeline stages except between stages connected via a network. For those stages, θ_i^k is defined as 9 msec, where 4 msec is due to propagation delay and 5 msec is due to the clock skew. Therefore, $\theta_3^1 = \theta_3^2 = 9ms$.

The second pipeline stage coupling attribute specifies the flow control mechanism used to guarantee that the system executes in a predictable manner. Large systems executing in a data-flow, or *work-conserving*, execution pattern may cause congestion. In a data-flow system, the source pipeline stage of each application stream always executes in a periodic manner while all other pipeline stages execute in a pseudo-periodic manner, *i.e.*, pipeline stage $i, 1 < i \le q(k)$, executes as soon the preceding processing step $i-1$ outputs $PIDS_i^k$ bytes of new data. Because the exact time that processing step $i-1$ completes execution is a function of the current load on the resource on which it executes, the exact time when i begins can vary greatly, bringing unpredictability to the system. Unpredictability may cause congestion, may cause processing steps' local deadlines to be missed, and may result in large output jitter. *Jitter* is defined as the time interval between the completion of two successive instances of a pipeline stage. Furthermore, congestion renders system timing analysis, to be performed in Section 5, intractable [9].

To rectify this problem, flow-control regulators may be inserted between every pair of pipeline stages. Flow-controlled service is *non-work conserving, i.e.*, a pipeline stage i can execute for at most $\sum_{j \in C^k} c_{ij}^k$ time units in any time interval T_i^k [21]. Various approaches can be used to implement flow-control, including *jitter earliest-due-date* [20], *stop-and-go queueing* [9], *rate-controlled static priority* [22], and *hiearchical round robin* [12]. Comparisons among these various approaches can be found in [23]. Flow controllers, by guaranteeing that every pipeline stage executes in a strictly periodic manner, eliminate congestion, increase system predictability and reduce jitter [21].

Previous work on flow control has been mostly used in the network domain where all packets are assumed to be of the same size. In heterogeneous systems described in this paper, where adjacent pipeline stages execute at different rates, multi-rate flow control must be used. *Multi-rate flow* control is an extension of basic flow control to incorporate different execution rates between pipeline stages. Refer to [4] on how to extend the basic flow control mechanisms described in the preceding paragraph into multi-rate flow controllers.

This paper assumes that every pair of pipeline stages in the example suite uses Multi-rate Stop & Go Flow Control [4]. Processing steps within a pipeline execute in a data-flow manner. This is indicated in Figure 9 with the solid and hollow arrows. Figure 10 shows the resulting pipelined execution pattern for the audio application. The first pipeline stage becomes

Audio Pipeline Stages:	get audio sample	↓4	low-pass filter	I/O Route	I/O Route	D/A convertor
Name:	Disk-1	Sparc -1		FDDI 1	PC-1	DSP-1
Exec.Time	30.5 usec	512 usec	256 usec	20.6 usec	20.6 usec	25 usec
Period:	16 msec	16 msec		32 msec	125 usec	125 usec
Deadline:	16 msec	16 msec		32 msec	125 usec	125 usec
Prop. Dly:	~0	~0		4 msec	~0	~0

(a) Audio

Video Pipeline Stages:	get video frame	I/O Route	I/O Route	lock display	display video	release lock
Resource Name	Disk-1	Sparc-1	FDDI 1	PC-1		
Exec. Time:	30.5 usec	51.2 usec	20.6 usec	92 usec	18.4 msec	92 usec
Period:	92.6 usec	92.6 usec	46.3 usec	33 msec		
Deadline:	92.6 usec	92.6 usec	46.3 usec	33 msec		
Prop. Dly:	~0	~0	4 msec	~0		

(b) Video

Table 12: Pipeline Stage Attributes (assuming 512 byte FDDI packets)

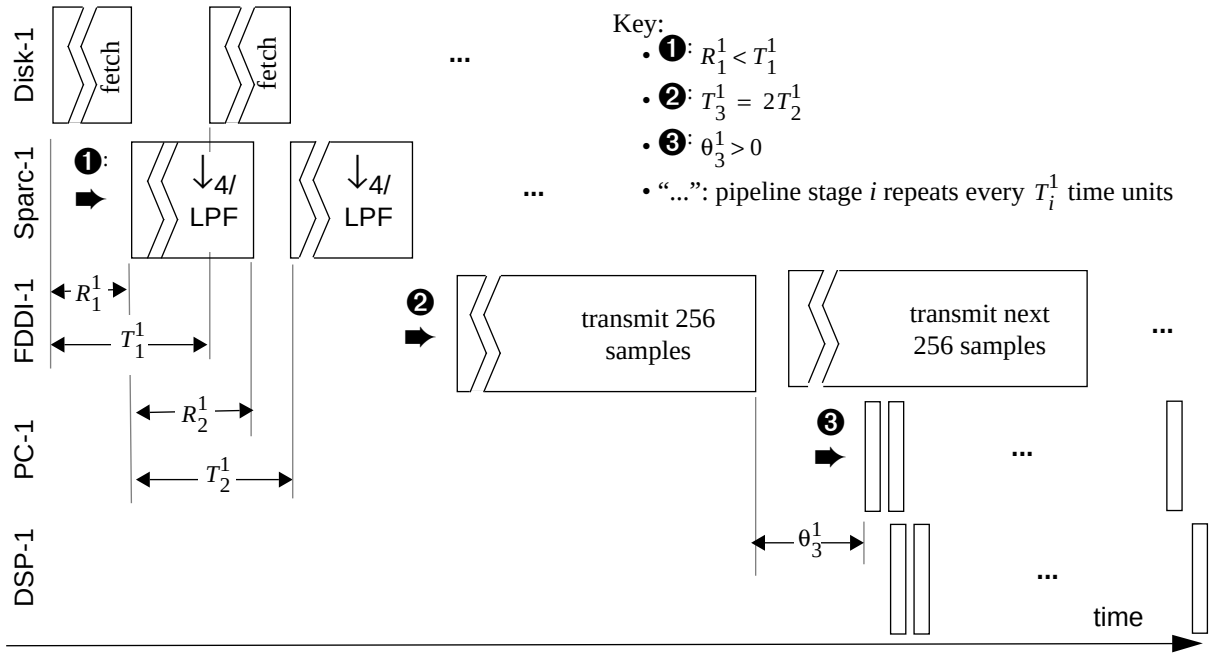


Figure 10: **Distributed Pipeline Multi-Rate Execution Pattern**

eligible for execution at time $t = 0$. During execution it may get preempted by higher priority pipeline stages executing on the same resource. However, assume that it completes by time $t = R_1^1$. Due to flow control, the next instance of pipeline stage 1 cannot begin until time $t = T_1^1$. On the other hand, the second pipeline stage becomes eligible as soon as the first pipeline stage completes execution, at time $t = R_1^1$ and completes by time $t = R_1^1 + R_2^1$.

Due to the change in execution rate between pipeline stages 2 and 3, stage 3 cannot begin until stage 2 executes twice. This is due to the disk sector and network packet size disparity and to the decimation-by-4 algorithm. With this in mind, note that pipeline stage 3 becomes eligible for execution at time $t = R_1^1 + T_2^1 + R_2^1$. The T_2^1 is added due to the flow control mechanism that guarantees that pipeline stage 2 does not begin execution until time t = $R_1^1 + T_2^1$. While executing, the second instance of pipeline stage 2 may get preempted but completes by time $t = R_1^1 + (T_2^1 + R_2^1) + R_3^1$.

Due to the propagation delay and clock skew between the two resources executing pipeline stages 3 and 4, pipeline stage 4 cannot begin until time $t = R_1^1 + (T_2^1 + R_2^1) + R_3^1 + \theta_3^1$ and completes by time $R_1^1 + (T_2^1 + R_2^1) + R_3^1 + \theta_3^1 + R_4^1$. A similar execution pattern is observed for the last pipeline stage. Note that this pipelined execution pattern results in efficient resource usage and that flow control results in predictable resource usage. System timing analysis next checks that the system meets all application timing requirements.

5. System Timing Analysis

Distributed Pipeline Scheduling creates a initial real-time system. System timing analysis next determines whether the system meets all timing requirements. If not, optimization techniques may be used to re-configure the system.

Distributed pipeline scheduling decomposes the very complex multi-resource timing analysis problem into sets of (*i*) end-to-end stream and (*ii*) single resource timing analysis problems:

- *stream timing analysis*: are all end-to-end application timing requirements met, *assuming* that all pipeline stages completed execution by their allotted local deadlines?
- *resource schedulability analysis*: are all local pipeline stage deadlines met?

The second step is necessary because, unlike processor pipelines, distributed pipeline stages from multiple applications may share a resource. Therefore, the Framework must check to see if the resource is able to meet all allocated processing steps' local deadlines. On the other hand, the first step is similar to calculating latency in a processor pipeline, with the following complications:

- distributed pipeline stages within a single pipeline may execute at different rates,
- distributed pipeline stages may have deadlines less than period,
- distributed pipeline stages may be asynchronously clocked, and
- network resources have non-negligible propagation delays.

This hierarchical timing analysis technique provides the Framework the unique ability to analyze complex real-time systems composed of non-ideal, heterogeneous resources, each with its own local scheduling policy. The remainder of this section calculates the worst-case end-to-end latency, I/O rate and jitter bounds and resource schedulability for systems designed using Distributed Pipeline Scheduling techniques. Proofs for the equations can be found in [5]. Latency, I/O rate and jitter bounds can each be calculated in $O(nS)$, where n is the number of pipeline stages per distributed pipeline and S is the number of the distributed pipelines per system. Resource schedulability analysis can be calculated in $O(n^2 m \gamma R)$, where n is the number of pipeline stages per resource, m is the number of target-specific processing steps per pipeline stage, γ is the ratio of the periods of the largest pipeline stage to the smallest pipeline stage per resource, and R is the number of resources per system [7].

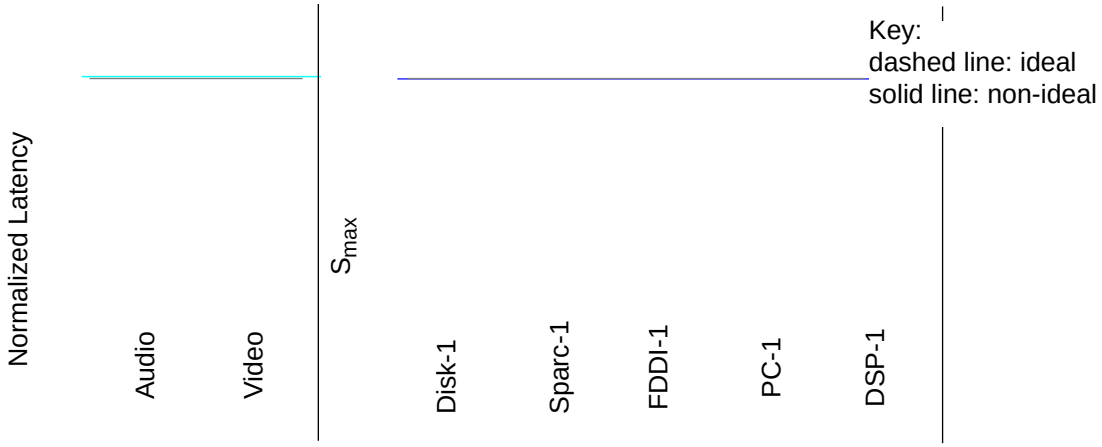
For each pipeline stage i of PAS k, the initial I/O data ratio between pipeline stages i and i+1 is as follows: $\beta_i^k = (PIDS_{i+1}^k + PIIDS_{i+1}^k) / PODS_i^k$.

The following three lemmas determine the worst-case end-to-end latency, input and output rates and input and output jitter bounds. These lemmas assume that $\theta_i^k = P_i^k + max(0, \Phi_i^k)$ and stop-and-go flow control between every pair of pipeline stages. Refer to [4] for similar bounds for other values of θ_i^k.

End-to-end latency for a pipelined application stream k is defined as the worst-case time interval starting when the source pipeline stage becomes eligible for execution until the sink pipeline stage completes execution *once*.

Lemma 1: *Assuming* $D_i^k \le T_i^k$ *and* $\theta_i^k = P_i^k + max(0, \Phi_i^k)$, *the worst-case end-to-end latency for application streams k,* $1 \le k \le S$, *is*

$$Latency^k = \left(\sum_{j=1}^{q(k)-1} \left\lceil \beta_j^k - 1 \right\rceil T_j^k + D_j^k + P_j^k + 2\left|\Phi_j^k\right| \right) + D_{q(k)}^k$$



(a) Latency **(b) Resource Schedulability**

Figure 11: **Initial System Timing Analysis Results**

Figure 10 can be used to verify Lemma 1 for the audio application stream. From Figure 10, the audio end-to-end latency is $R_1^1 + (T_2^1 + R_2^1) + R_3^1 + \theta_3^1 + R_4^1 + R_5^1$. Note that the worst-case scenario occurs if $R_i^1 = D_i^1$ for $i = 1$, 2, ..., 5. Substituting D_i^1 for R_i^1 results in Lemma 1.

Input and output rates determine how many times the source and sink pipeline stages execute in any time interval.

Lemma 2: *Assuming* $\theta_i^k = P_i^k + max(0, \Phi_i^k)$ *and* $D_i^k \le T_i^k$, *the input and output rates, respectively, for all applications k,* $1 \le k \le S$, *are*

$$Rate_{IN}^k = 1/T_1^k \text{ and}$$

$$Rate_{OUT}^k = 1/T_{q(k)}^k$$

Input and output jitter determine the maximum time interval between the completion of two instances of the source and sink pipeline stages, respectively.

Lemma 3: *Assuming* $\theta_i^k = P_i^k + max(0, \Phi_i^k)$ *and* $D_i^k \le T_i^k$, *the worst-case input and output jitter bounds, for all applications k,* $1 \le k \le S$, *are*

$$Jitter_{IN}^k = T_1^k + D_1^k \text{ and}$$

$$Jitter_{OUT}^k = (T_{q(k)}^k + D_{q(k)}^k)$$

End-to-end latency, throughput and jitter lemmas *assume* that all processing steps completed by their local deadlines. Because multiple distributed pipeline stages may execute on a shared resource, resource schedulability analysis can determine if all pipeline deadlines are indeed met. This paper uses the Degree of Schedulable Saturation [17], S_{max}^r, as the resource schedulability metric. However, the Framework is able to accommodate any other non-binary resource schedulability metric.

Lemma 4: *Resource schedulability for each resource r,* $1 \le r \le R$, *is as follows:*

$$S_{max}^r = max_{1 \le i \le m(r)} S_i^r$$

where S_i^r can be calculated for any resource *r* using the scheduling models available in the Library of Single Resource Scheduling Models.

The following theorem can be used to determine if all application timing requirements are met.

Theorem 1: *Given a set of distributed pipelines, all application timing requirements are met if,* $\forall r, 1 \le r \le R, \forall k, 1 \le k \le S$,

- end-to-end latency requirements are met:
  $Latency^k \le SL^k$
- I/O rate requirements are met:
  $SIR_{max}^k \ge Rate_{IN}^k \ge SIR_{min}^k$ and
  $SOR_{max}^k \ge Rate_{OUT}^k \ge SOR_{max}^k$
- jitter requirements are met:
  $Jitter_{IN}^k \le J_{IN}^k$ and $Jitter_{OUT}^k \le J_{OUT}^k$
- all pipeline stage deadlines are met: $S_{max}^r \le 1$

Proofs for Theorem 1 and Lemmas 1 to 4 can be found in [5].

Partially applying Theorem 1 to the example audio/video application suite results in the graphs shown in Figure 11. Graphs for jitter and I/O rates were not shown for space considerations. Figure 11a shows the normalized latency results for the audio/video application suite for both ideal and non-ideal cases. *Normalized latency* is defined as the actual latency divided by the required latency; latency requirements are met if the normalized latency is less than one. The ideal case corresponds to zero propagation delay for network resources and zero clock skew between all

resources. Since the ideal normalized latency is below one for both applications, under ideal circumstances the latency requirements are met. For the non-ideal case, however, assuming a 4 msec propagation delay on the FDDI resource and a 5 msec clock skew, the latency result is not met for the audio application.

The ideal and non-ideal resource schedulability graphs are shown in Figure 11b. The difference between the non-ideal and ideal graphs is due to resource overhead and blocking. Since these are all below one, all local pipeline stage deadlines were met.

6. Optimizing Pipeline Stage Attributes

Initial system design and analysis identifies whether all application timing requirements are met. If certain requirements are not met, distributed pipelines can be re-configured to meet these requirements. Several optimization techniques are available:

- re-map,
- change pipeline stage deadlines,
- change flow control parameters, or
- change system configuration parameters.

Re-mapping creates new target-specific application streams that may possibly result in distributed pipelines with different numbers of pipeline stages executing over different system resources. Based on the load characteristics of these new system resources and new pipeline stage attributes, system timing requirements may be met.

Changing pipeline stage deadlines affect end-to-end latency, input and output jitter and resource schedulability. Generally, decreasing deadlines decreases latency and jitter but increases the resource schedulability metric, S^r_{max}. Increasing deadlines has the opposite effect. A trade-off between these various metrics can be done to meet application requirements.

Changing flow control parameters affect latency, input and output rates and jitter only when networks have non-negligible propagation delays and resource clocks are not synchronized. Under these conditions, the phase delay parameter, θ^k_i, can be used to reduce jitter at the expense of latency and vice-versa [4].

Changing system configuration parameters can affect end-to-end latency, input and output rates and jitter and resource schedulability. Certain configuration parameters, such as an operating system's timer tick interval, only affect resource schedulability [13]. Other parameters, such as network packet size, affect the period of a pipeline stage and hence affect latency, I/O rates and jitter and resource schedulability.

For example, Table 13 lists the new pipeline stage attributes for the audio and video example streams when the FDDI network packet size is changed to 64 bytes. Changing the packet size reduces the period of the network pipeline stages. The resulting latency and resource schedulability graphs are shown in Figure 12. Because this reduces the normalized audio latency to below one, all application timing requirements are met. Note that now propagation delay and clock skew are a larger portion of the audio latency, as shown by the greater difference between the ideal and non-ideal latency graphs. Similarly, changing the packet size doesn't affect the ideal FDDI resource schedulability. On the other hand, since overhead is now a larger part, the non-ideal resource schedulability result increases.

7. Comparison to Other Approaches

There has been a lot of recent activity in end-to-end design and analysis of real-time systems. A majority of this research has focused on end-to-end analysis over a set of precedence-constrained tasks/ sub-tasks executing on a single resource. Harbour, Klein and Lehoczky [7] provided a framework for analyzing an uni-processor resource executing precedence-constrained tasks with varying execution priorities and synchronization requirements. Jeffay [11] introduced a real-time producer/consumer paradigm for expressing task precedence-constraints and for reasoning about timing behavior of programs. Gerber, Hong and Saksena [8] provided a comprehensive uni-processor design methodology for guaranteeing application end-to-end timing requirements.

There have been several papers on multi-resource scheduling. Malcom and Zhao [15], Verma [19], and Sathaye [17] explored end-to-end analysis of multi-hop network resources. Huang and Du [10] extended the uni-processor real-time consumer/producer paradigm and provided a multi-dimensional bin-packing approach to resource allocation and task scheduling. Limitations included a target platform consisting of a single CPU and a disk, each executing the rate-monotonic scheduling paradigm. No end-to-end timing analysis was provided. Natale and Stankovic [16] provided a dual off-line/on-line framework for guaranteeing deadlines for processes communicating via synchronous primitives. Bettati and Liu [1] provided a set of heuristics for resource allocation and end-to-end stream analysis for homogeneous multi-processor systems. Applications were scheduled in a flow-shop/data-flow manner. Flow-shop scheduling has been applied to a homogeneous set of resources, *i.e.*, all CPUs with identical scheduling paradigms. Flow control mechanisms were not provided and resource propagation delays and clock synchronization issues were not considered.

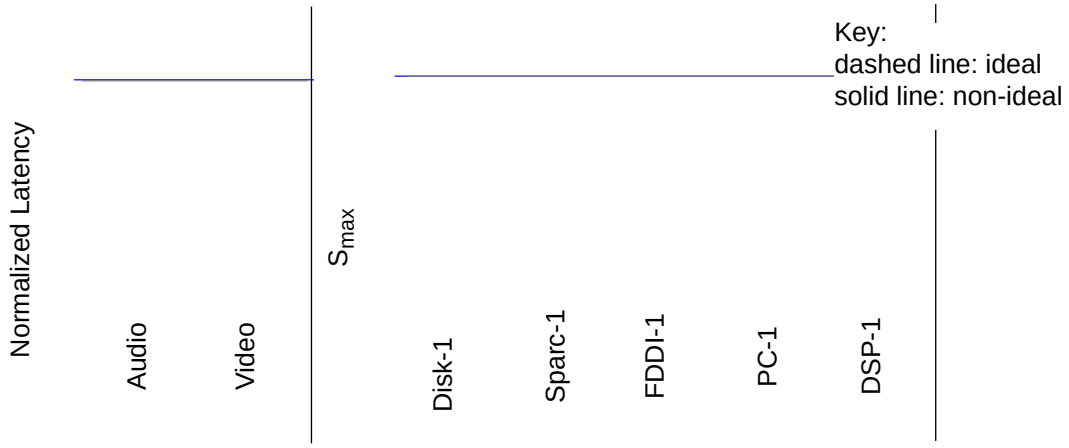
Pipeline Stages:	get audio sample	↓4	low-pass filter	I/O Route	I/O Route	D/A convertor
Name:	Disk-1	Sparc -1		FDDI 1	PC-1	DSP-1
Exec.Time	30.5 usec	512 usec	256 usec	3 usec	20.6 usec	25 usec
Period:	16 msec	16 msec		4 msec	125 usec	125 usec
Deadline:	16 msec	16 msec		4 msec	125 usec	125 usec
Prop. Dly:	~ 0	~ 0		4 msec	~ 0	~ 0

(b) Audio Application Stream

Pipeline Stages:	get video frame	I/O Route	I/O Route	lock display	display video	release lock
Resource Name	Disk-1	Sparc-1	FDDI 1	PC-1		
Exec. Time:	30.5 usec	51.2 usec	2.8 usec	92 usec	18.4 msec	92 usec
Period:	92.6 usec	92.6 usec	5.78 usec	33 msec		
Deadline:	92.6 usec	92.6 usec	5.78 usec	33 msec		
Prop. Dly:	~ 0	~ 0	4 msec	~ 0		

(b) Video

Table 13: Optimized Pipeline Stage Attributes (assuming 64 byte FDDI packets)



(a) Latency **(b) Resource Schedulability**

Figure 12: **Re-configured System Timing Analysis Results**

8. Conclusion

This paper described the Distributed Pipeline Scheduling Framework that provides a systematic approach to designing distributed, heterogeneous real-time systems that utilize resources in an efficient, pipelined and predictable manner. Key elements of Distributed Pipeline Scheduling include:

- defining a clear abstraction for representing periodic real-time applications that is both meaningful to an application designer and contains sufficient information such that the applications can be mapped efficiently to a distributed target platform
- developing a clear abstraction for capturing the fundamental properties of distributed pipelining
- providing a natural flow control mechanism that eliminates congestion
- decomposing the very complex multi-resource scheduling problem into a set of single resource scheduling problems with well defined interactions
- providing a analysis methodology to support arbitrary, heterogeneous scheduling policies among system resources, and
- delineation of how manipulating system configuration parameters affect various application timing metrics.

This work is currently being extended in several fronts. Mapping and optimization techniques are being quantitatively examined. Furthermore, application stream specifications, mapping algorithms and system timing analysis are each being augmented to incorporate stochastic execution times, to allow dynamic application arrivals and departures [3], and to provide probabilistic timing guarantees, respectively.

References

[1] R. Bettati and J. Liu, "Algorithms for End-to-End Scheduling to Meet Deadlines" *Proceedings of the 2nd IEEE Conf. on Parallel and Distributed Systems*, Dec. 1990.

[2] S. Chatterjee and J. Strosnider, "Distributed Pipeline Scheduling: End-to-End Analysis of Heterogeneous, Multi-Resource Real-Time Systems," In the *15th International Conference on Distributed Computing Systems*, Vancouver, Canada, May 1995.

[3] S. Chatterjee and J. Strosnider, "A Generalized Admissions Control Strategy for Heterogeneous, Distributed Multi-media Systems," *CMUCSC-95-3 Technical Report*, Carnegie Mellon University, April 1995.

[4] S. Chatterjee and J. Strosnider, "Designing Distributed Multimedia Systems Configured to Meet Application Latency and Output Jitter Requirements," *CMUCSC-95-4 Technical Report*, Carnegie Mellon University, June 1995.

[5] S. Chatterjee and J. Strosnider, "Distributed Pipeline Scheduling: A Framework for Distributed, Heterogeneous Real-Time System Design," *Technical Report*, Carnegie Mellon University, June 1995. Please email the author at *saurav@ece.cmu.edu* for a copy.

[6] S. Daigle and J. Strosnider, "Disk Scheduling of Continuous Media Data Streams," *SPIE Conference on High-Speed Networking and Multimedia Computing*, 1994.

[7] M. Harbour, M. Klein, and J. Lehoczky, "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority," *Proceedings of the 1991 Real-Time Systems Symposium*, December 1991.

[8] R. Gerber, S. Hong and M. Saksena, "Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes," *IEEE Real Time Systems Symposium*, 1994.

[9] S. Golestani, "Congestion-Free Transmission of Real-Time Traffic in Packet Networks," *Proceedings of INFOCOM*, San Francisco, June 1990.

[10] J. Huang and D. Du, "Resource Management for Continuous Multimedia Database Applications," *IEEE Real Time Systems Symposium*, 1994.

[11] K. Jeffay, "The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems," *ACM/SIGAPP Symposium on Applied Computing*, Feb. 1993.

[12] C. Kalmanek, H. Kanakia, and S. Keshav, "Rate Controlled Servers for very High-speed Networks," *IEEE Global Telecommunications Conference*, pp 300.3.1 - 300.3.9, December 1990.

[13] D. Katcher, H. Arakawa, and J. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers," *IEEE Transactions on Software Engineering*, 19(9), September 1993.

[14] D. Katcher, "Engineering and Analysis of Real-Time Operating Systems," *Ph.D. Thesis*, Carnegie Mellon University, 1994.

[15] N. Malcom and W. Zhao, "Guaranteeing Synchronous Messages with Arbitrary Deadline Constraints in an FDDI Network," *IEEE Conf. on Local Computer Networks*, 1993.

[16] M. Natale and J. Stankovic, "Dynamic End-to-End Guarantees in Distributed Real Time Systems," *IEEE Real Time Systems Symposium*, 1994.

[17] S. Sathaye, "Scheduling Real-Time Traffic in Packet Switched Networks," *Ph.D. Thesis*, Carnegie Mellon University, 1993.

[18] E. Snow, "Engineering FutureBus+ IEEE 896.1 for Real-Time Applications," *M.S. Thesis*, CMU, December 1992.

[19] D. Verma, "Guaranteed Performance Communication in High Speed Networks," *Ph.D. Thesis, University of California at Berkeley*, November 1991.

[20] D. Verma, H. Zhang, and D. Ferrari, "Guaranteeing Delay Jitter Bounds in Packet-switched Networks," *Proceedings of Tricomm 91*, pp35-46, April 1991.

[21] H. Zhang and D. Ferrari, "Rate-Controlled Service Disciplines," *Journal of High Speed Networks* 3(4), 1994.

[22] H. Zhang and D. Ferrari, "Rate-controlled Static Priority Queueing," *Proceedings of IEEE INFOCOM 93*, pp 227-236, April 1993.

[23] H. Zhang and S. Keshav, "Comparison of Rate-based Service Disciplines," *Proceedings of ACM SIGCOMM 91*, pp 113-122, September 1991.