Report Number: WUCS-91-14

1991-10-01

# A Recurrence Model for Asynchronous Pipeline Analysis

Fengmin Gong, Zubin Dittia, and Gurudatta M. Parulkar

This paper presents an analytical model for performance analysis of asynchronous pipelines based on recurrence relations. The model accurately describes the behavior of an asynchronous pipeline, yet yields to very efficient computations. It has been implemented and verified using discrete event simulations. Results of the verification experiments are also described. We also outline a number of ways that the proposed recurrence model can be used to gain insight into many issues concerning engineering of asynchronous pipelines.

# A Recurrence Model for Asynchronous Pipeline Analysis

Fengmin Gong, Zubin Dittia and Gurudatta M. Parulkar

WUCS-91-14

October 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

*This work is supported in part by the National Science Foundation, and an industrial consortium of Bellcore, BNR, DEC, Italtel SIT, NEC, NTT and SynOptics.*

# A Recurrence Model for Asynchronous Pipeline Analysis[†]

Fengmin Gong*
*lfg@wucs.wustl.edu*
(314) 935-4163

Zubin Dittia*
*zubin@dworkin.wustl.edu*
(314) 935-4163

Gurudatta M. Parulkar*
*guru@flora.wustl.edu*
(314) 935-4621

January 23, 1992
WUCS-91-14

*Original draft on October 7, 1991*
*Revised on January 23, 1992*

## Abstract

This paper presents an analytical model for performance analysis of asynchronous pipelines based on recurrence relations. The model accurately describes the behavior of an asynchronous pipeline, yet yields to very efficient computation. It has been implemented and verified using discrete event simulations. Results of the verification experiments are also described. We also outline a number of ways that the proposed recurrence model can be used to gain insight into many issues concerning engineering of asynchronous pipelines.

[†]This work is supported in part by the National Science Foundation, and an industrial consortium of Bellcore, BNR, DEC, Italtel SIT, NEC, NTT, and SynOptics.

*Computer & Communications Research Center, Bryan 405 - Box 1115, Washington University, One Brookings Drive, St. Louis MO 63130-4899.

# A Recurrence Model for Asynchronous Pipeline Analysis

Fengmin Gong
*lfg@wucs.wustl.edu*
(314) 935-4163

Zubin Dittia
*zubin@dworkin.wustl.edu*
(314) 935-4163

Gurudatta M. Parulkar
*guru@flora.wustl.edu*
(314) 935-4621

## 1. INTRODUCTION

There are two well-recognized techniques for achieving concurrency by simultaneous operation on multiple computing elements: *parallelism* and *pipelining*. In the parallelism approach, a hardware structure is replicated many times and each replica executes in parallel different parts of the large computation; pipelining technique splits the hardware structure into a sequence of substructures (called stages) corresponding to phases of the computation task and allows all stages to operate simultaneously on different parts of the computation [1, Chapter 1].

In principle, a pipeline can either operate in lock-step fashion (controlled by the "ticks" of a global clock) or operate by having each stage processor synchronize only with its neighbors using a handshaking protocol. In the first case, the pipeline is called a *synchronous* pipeline; the second case corresponds to an *asynchronous* pipeline. Synchronous pipelines are generally simpler to design. Pipelining has also been recognized as an important computation model at the programming level [3, 4]. Instruction pipelines and arithmetic pipelines have been used with great success in modern computer architectures.

Synchronous pipelines have received considerable attention over the years. For instance, many methodologies for the design and operation of hardware pipelines have been well-documented [1, 5, 6, 12]. Hwang and Xu have presented equations for the speedup of their pipeline net [2]; Dubey and Flynn [7] have also reported closed-form solutions for the speedup of a hardware pipeline, which account for the effect of various hardware overheads as well as pipeline setup/flush overhead and delay variations. All these results only apply to synchronous pipeline operation. We have not seen analytical models that deal with asynchronous pipelines in the literature.

### 1.1. Motivations

We believe two factors are mostly responsible for the limited use of asynchronous pipelines so far. The complexity of asynchronous control logic is high; and characterization of asynchronous behavior has been difficult, thus limiting our understanding of asynchronous pipelines. Distributed applications that mandate the use of asynchronous pipelines are just beginning to become practical.

Asynchronous pipelines are absolutely necessary when pipeline stages are physically distributed or have varying processing delays. For example, in a related research effort, we have proposed pipelining across high-speed networks as an efficient model for televisualization, called PTV (Pipelined TeleVisualization) [8]. Televisualization pipelines are significantly different from most hardware pipelines in several ways:

- The complex visualization computation makes it almost impossible to obtain perfect partitioning.
- Stages of a PTV pipeline are distributed across (inter)networks with communication errors and variable delays.
- Each stage of the pipeline is an independent computer with multiprogrammed load, and fluctuation of this load causes the processing delay at the stage to also vary with time.
- User interaction with different stages of the PTV pipeline is required, which also introduces unpredictable delays.

These characteristics clearly suggest that a global clock will be very difficult to provide. Also, asynchronous operation allows each processor to work at its potential speed while synchronous operation would force every stage to keep in pace with the slowest among them. Therefore, asynchronous pipeline operation is necessary to achieve high performance for such distributed applications.

### 1.2. Outline

To help better understand the behavior of asynchronous pipelines, we present a model for their performance analysis in this paper. We derive a set of recurrence relations that describe the exact behavior of certain abstract asynchronous pipelines with DAG (directed acyclic graph) topology; we also devise two simple transformations that allow effects of communication overhead and extra buffers to be modeled using the recurrence relations. We develop an efficient algorithm for evaluating the recurrence relations and verify our model with discrete event simulations. The result is an accurate and efficient tool for performance analysis of distributed asynchronous pipelines. Detailed application of the model to the analysis of particular pipelines is a task of future endeavors.

The rest of the paper is organized as follows: Section 2 derives a set of recurrence relations for an abstract asynchronous pipeline having a linear topology. Section 3 outlines transformations for modeling communication overhead and buffers in the recurrence model. Section 4 derives recurrence relations for abstract pipelines with a DAG topology. Section 5 introduces a station abstraction to aid in the description of application pipelines and presents results from experiments that help verify the recurrence model approach. In Section 6, we discuss how the recurrence model can be used to gain insight into engineering of asynchronous pipelines along with some related work. Finally, Section 7 provides concluding remarks.

## 2. LINEAR PIPELINES

In this section, we define an abstraction to real-life linear asynchronous pipelines and derive a set of recurrence relations to describe the behavior of the abstraction. The abstraction is necessary to make the pipeline behavior amenable to simple recurrence description. Methods for modeling more realistic aspects of the pipeline will be presented in Section 3.

### 2.1. Abstract Linear Pipeline

Suppose we have a linear pipeline with $m$ stages and $n$ data items to be processed using the pipeline. Each stage is simply a processing element that gets an input data item, processes it, sends the result to the next stage, and then repeats this process on the next data item. Figure 1 illustrates such a pipeline.

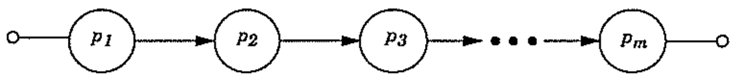

Figure 1: An Abstract Pipeline

We refer to a list of stage processors $p_1, p_2, ..., p_m$, with processor $i+1$ depending on the output of processor i (for all $i$, $1 \leq i \leq m-1$), as a dependency chain. We also assume that the following is true for this pipeline:

- Asynchronous behavior: There is no single global clock for the synchronization of pipeline stages. Therefore, coordination between stages is realized by a handshaking protocol.
- Ideal Handshaking: A stage always has complete and up-to-date knowledge about the state of the next stage.
- Eager computation: Each stage starts processing a data item as soon as it arrives at the stage and sends it to the next stage as soon as the next stage is ready to accept it.
- Negligible data communication overhead: Time overhead associated with exchanges of data is negligible; methods to explicitly model this overhead will be discussed in Section 3.
- No additional buffer: There is assumed to be no additional buffer besides the single working space within each processor. Again, modeling of buffers will be discussed in Section 3.

### 2.2. Recurrence Relations

*Notations.* Let $D$ be an $m \times n$ matrix of processing delay:

$$D = [d_{ij}] \quad 1 \leq i \leq m, 1 \leq j \leq n$$

where $d_{ij}$ denotes the processing delay for data item $j$ at stage $i$.

We now introduce two recurrence variables: $f_{ij}$ is the time at which data item $j$ finishes being processed at stage $i$; $l_{ij}$ is the time at which data item $j$ leaves stage $i$ and arrives at stage $i+1$. Since a data item has to be processed at a stage before it can leave the stage, $l_{ij} \geq f_{ij}$ holds for any data at any stage. We can derive recurrence relations involving $f_{ij}$ and $l_{ij}$ as follows.

*Initial Conditions.* Since the pipeline is empty at the beginning, the first data item can flow down the pipeline as fast as the processing at each stage allows. Hence the following initial condition is true:

$$l_{i1} = f_{i1} = \sum_{k=1}^{i} d_{k1} \qquad 1 \leq i \leq m$$

*Boundary Conditions.* It should also be observed that stage 1 does not depend on data from any previous stages, and it is always ready to process the next data item as soon as stage 2 is ready to accept the current processed data item. On the other hand, the stage $m$ does not need to wait for any subsequent stage to accept its output, and thus, a data item is consumed (therefore considered to have left the pipeline) as soon as it has finished being processed by stage $m$. These two observations are formalized as two boundary conditions:

$$f_{1j} = l_{1j-1} + d_{1j} \qquad 2 \leq j \leq n$$

$$l_{mj} = f_{mj} \qquad 1 \leq j \leq n$$

*Recurrence Body.* For an internal stage of the pipeline, we emphasize the following two important behavioral properties: (1) a stage can start processing a data item if and only if the data has arrived from the previous stage, and (2) a data item can leave for the next stage if and only if it has finished being processed at the current stage and the next stage has sent out its previous data item (and is therefore ready to accept another). Figure 2 depicts two typical timing scenarios at an internal stage of the pipeline. For data item $j$ at stage $i-1$, by the time the processing finishes, stage $i$ is already free to receive data item $j$, and so $l_{i-1j} = f_{i-1j}$. The corresponding waiting time is represented by the hatched area. At stage $i$, data item $j$ finishes earlier than stage $i+1$ is ready to receive it (i.e., $l_{i+1j-1} > f_{ij}$), thus causing data item $j$ to wait at stage $i$ until time $l_{ij} = l_{i+1j-1}$. This waiting time is marked by the gray area in the figure.

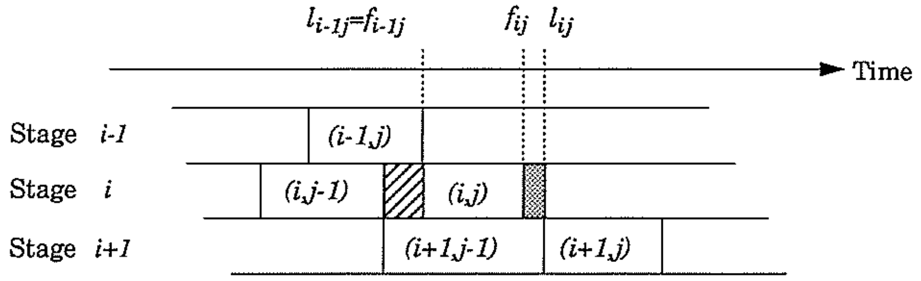

Figure 2: Dependency Timing

The general behavior of an internal stage is captured in the following recurrence body:

$$f_{ij} = l_{i-1j} + d_{ij} \qquad 2 \leq i \leq m, 2 \leq j \leq n$$

$$l_{ij} = \max(f_{ij}, l_{i+1j-1}) \quad 1 \le i \le m-1, 2 \le j \le n$$

Given the 2-dimensional array $D$, the recurrence relations characterize exactly the behavior of the pipeline. By evaluating the recurrence relations for the set of input data items, we will obtain a complete trace of the finish time and the leave (or departure) time for each data item at every stage ($f_{ij}$ and $l_{ij}$ for $1 \le i \le m, 1 \le j \le n$). With this data, a set of commonly used performance measures for pipelines can be defined.

First, the ***average cycle time*** ($AC$) is a measure of time it takes, on average, for the pipeline to completely process one data item:

$$AC = \frac{f_{mn}}{n}$$

The ***throughput*** ($TH$) of a pipeline is the number of data items completely processed by the pipeline per unit time. It is simply the reciprocal of the average cycle time:

$$TH = \frac{1}{AC}$$

Definition of ***speedup*** ($SU$) requires another implementation of the computation as a reference. Let $T$ be the time it takes to process the set of $n$ data items by the reference implementation. The speedup is defined as the ratio between $T$ and the time taken by the $m$-stage pipeline to process the same set of data. It is given as:

$$SU = \frac{T}{f_{mn}}$$

Similarly if we are interested in the pipeline performance after the pipeline is filled up, we can define a new average cycle time $AC^+$ as:

$$AC^+ = \frac{f_{mn} - f_{m1}}{n-1} \quad n > 1$$

$AC^+$ will be used as the performance metric in this paper.

It is important to note that the recurrence relations derived above do not explicitly model the feedback control of a pipeline, and this will also be true for the recurrence relations to be derived in Section 4.2. This is not an issue because feedback paths normally carry control information requiring very small bandwidth. Moreover, important effects of feedback (e.g. flush of pipeline) can be indirectly modeled using the recurrence model by selecting suitable values of processing delays.

## 3. MODELING COMMUNICATION OVERHEAD AND DATA BUFFERS

Assumptions of zero data communication delay and zero buffering are not realistic. In this section we introduce two simple transformations which will allow us to model the effects of delay and buffering in the recurrence model.

### 3.1. Modeling Communication Overhead

When large volumes of data have to be exchanged between pipeline stages which are separated by networks, the overhead associated with this data communication is no longer negligible due to network delay, data loss, and data corruption. In general, the communication overhead consists of three components: effective end-to-end transport time for each data item $t_c$, the overhead for passing a data item from the communication system to the application space $t_r$, and the overhead for passing data in the other direction $t_s$. Time $t_c$ will include the effect of end-to-end propagation delay, transmission time (data item size/data rate), and error control overhead. We denote the pure intrastage computation delay by $t_p$. Figure 3 shows how the overheads affect pipeline operation by using a space-time diagram. Being consistent with the next generation host communication architecture, we assume there is a communication processor at each pipeline stage which operates in parallel with the main stage processor. Therefore, communication processing can overlap in time with intrastage computation. However, the overheads $t_r$ and $t_s$ are limiting the degree of overlap possible. Without loss of generality, the diagram is drawn for a 4-stage pipeline with the assumption that $t_r = t_s$, $t_c = t_p + t_r$, and all overhead values are constant throughout the pipeline. Four types of legends are used to mark areas corresponding to different delays in the diagram. Their specific associations are given at the bottom of Figure 3. All unmarked spaces in the diagram represent idle periods. It is clear that the space-time diagram can be interpreted as representing a new 7-stage pipeline, with end-to-end transport making up the 3 new stages. If we number the stages from 1 to 7 top-down, the equivalent processing delays become: $(t_p + t_s)$ for stage 1; $t_c$ for stages 2, 4, and 6; $(t_p + t_r + t_s)$ for stages 3 and 5; and $(t_p + t_r)$ for stage 7.

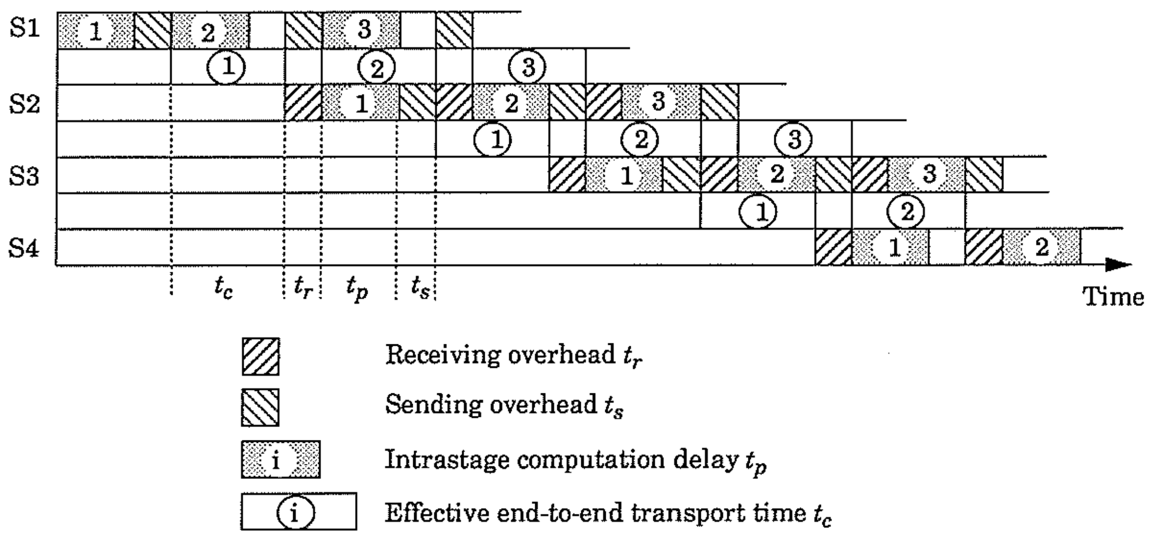

Figure 3: Communication Overhead Effect

Thus, for a given $m$-stage pipeline with interstage communication overhead, we can create an abstract pipeline with $(2m-1)$ stages, where for every pair of original computation stages $cp_i$ and $cp_{i+1}$, a new communication stage $cm_i$ is added between them. The equivalent processing delay values for $cp_i$, $cm_i$, and $cp_{i+1}$ will be determined as demonstrated above. The transformed pipeline now conforms to the abstraction of Section 2.

### 3.2. Modeling Data Buffers

In reality, asynchronous pipelines should always have buffers in the stages to deal with transient speed mismatch between stages in order to achieve efficient operation. Therefore, our recurrence model must be able to model buffers in the pipeline. In general, these buffers are finite FIFO queues with some delay overheads. Our aim is to find an equivalent representation of a real pipeline with buffers that will conform to the abstraction of Section 2. This will allow us to analyze any real-life linear pipeline using the recurrence relations derived in Section 2.2.

An elegant transformation results from the realization that in a real pipeline, an internal stage $i$ (with no extra buffer) actually serves as a single buffer between stage $i-1$ and stage $i+1$, in addition to providing the processing function. In another words, due to the presence of stage $i$, when data item $j$ finishes processing at stage $i-1$, whether it will be able to leave and free up stage $i-1$ does not affect stage $i+1$ at that moment. Stage $i+1$ will be affected only after data item $j$ finishes processing at stage $i$. This buffering effect is modeled in the derived recurrence relations. It is reflected in the property that at stage $i$, the leave time of data item $j$ ($l_{ij}$) depends only on the leave time of date item $j-1$ at stage $i+1$ ($l_{i+1j-1}$). Therefore, physical buffers in an application pipeline can be modeled as a segment of abstract pipeline. Let $B_i$ be the size of the extra buffer in the stage $cp_i$ of application pipeline, and $Q_i$ be a constant overhead associated with the buffer. Then we can create an abstract pipeline segment which consists of $B_i$ new stages, $b_1, b_2, \ldots, b_{B_i}$, in addition to $cp_i$. This transformation is depicted for stage $cp_i$ in Figure 4.

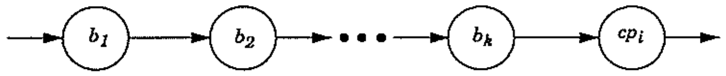

Figure 4: Buffer Transformation ($k = B_i$)

The overhead $Q_i$ can be represented in two ways. It can be distributed among the $B_i$ new stages, or it can be added to the processing delay of $cp_i$ with processing delays for the new stages set to zero. The two approaches are equivalent as far as pipeline performance is concerned.

## 4. DAG PIPELINES

The model developed so far is for linear asynchronous pipelines. In practice we can expect to have distributed applications in which the computation is more effectively organized as a pipeline with parallel processors at stages. It would thus be very useful if we could extend the recurrence model to such cases. We address precisely this issue in this section.

## 4.1. Abstract DAG Pipeline

Figure 5 shows a pipeline with a DAG topology. The first stage processor $p_1$ prepares the input data stream and splits it among three processors ($p_2$, $p_3$, and $p_7$). Processors $p_2$ and $p_3$ perform some computations in parallel, and their results are gathered by processor $p_4$. The result from $p_4$ is again distributed to $p_5$ and $p_6$ for further processing. Finally, processor $p_8$ performs processing on the combined results of $p_5$, $p_6$ and $p_7$ to produce the pipeline output.

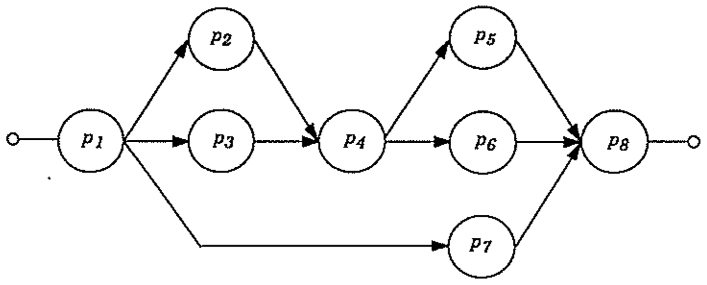

Figure 5: An Abstract Parallel Pipeline

Now we define a set of terms that will be useful in the derivation of recurrence relations. A *predecessor* of a processor $p$ is a processor whose output is directly input by $p$. A *successor* of a processor $p$ is a processor that requires direct output from $p$. A processor is a *source processor* if it has no predecessors. A processor is a *sink processor* if it has no successors. For example, in Figure 5, $p_2$ and $p_3$ are two successors of $p_1$; $p_2$ is a predecessor of $p_4$; $p_1$ is the source processor; and $p_8$ is the sink processor.

As a consequence of multiple processors operating in parallel, data flow may split and merge along the pipeline. The semantics of these actions directly affect the performance model. Therefore, a set of assumptions about the parallel behavior within the pipeline is in order:

- All data items are tagged with a unique sequence number at the source processor(s). This tag is maintained throughout the pipeline.
- Processor $p$ requires data items with the same tag from all its predecessors to be present before it can start processing. Once the processing finishes, all data items that served as input will be consumed and an output data item from the current processor is produced, carrying the same tag as the corresponding input data items.
- When a processor has more than one successor, the processor will produce one output data item for each of its successors, with all data items carrying the same tag. The current processor can begin processing the next input data item(s) only after output data items have been forwarded to its successors.

### 4.2. Recurrence Relations

*Notation.* We view the set of dependencies among processors of the DAG as a set of dependency chains. We use a two dimensional indexing scheme for the identification of processors. The processors are partitioned into stages as follows. Let $m$ be the total number of stages, which is defined as the number of processors on the longest dependency chain. Each source processor is labeled as a stage 1 processor and a sink processor as a stage $m$ processor. For any internal processor (i.e., not a source or sink processor), if $i$ is the maximum distance from the processor to any sink processor, this processor is labeled as a stage $(m-i)$ processor. If $w_i$ is the number of processors at stage $i$, we label these processors by tuples $(i,1),(i,2),\cdots,(i,w_i)$. Also, we use *pred*$(i,j)$ to denote the set of predecessors for processor $(i,j)$ and *succ*$(i,j)$ to denote the set of successors to processor $(i,j)$. The abstract parallel pipeline in Figure 5 can be labeled using the above scheme as shown in Figure 6. For example, processor $(3,1)$ has two predecessors $(2,1)$ and $(2,2)$, and two successors $(4,1)$ and $(4,2)$.

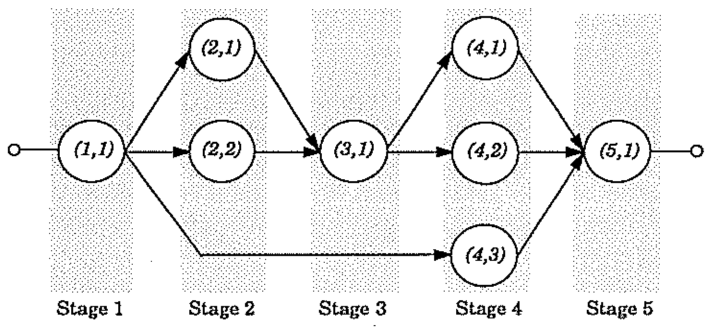

Figure 6: A Parallel Pipeline – Labeled

Now we introduce recurrence variables. Let $s_{ijk}$ be the time at which processing of data item $k$ begins at processor $(i,j)$. Let $d_{ijk}$ denote the processing delay for data item $k$ at processor $(i,j)$, and let $f_{ijk}$ denote the finish time of data item $k$ at processor $(i,j)$. Finally, $l_{ijk}$ represents the time at which data item $k$ leaves processor $(i,j)$ and arrives at all processors in *succ*$(i,j)$. We assume that a processor begins execution when all input data items have arrived, and it can forward its output data item to its successors only when they are ready to receive the data item. A set of recurrence relations in these variables can be derived and organized as initial conditions, boundary conditions, and recurrence body.

*Initial Conditions.* Since the pipeline is empty initially, the first data item ($k=1$) can be processed at a given processor as soon as all the corresponding input data items are received from predecessors; and the resulting output data item can move to successors immediately after its processing is finished. The first data item is available to the source processors at time 0. Therefore, the following initial conditions hold for the pipeline:

$$s_{1j1}=0$$

$$f_{1j1}=d_{1j1}$$

$$l_{1j1} = f_{1j1}$$

$$s_{ij1} = \max_{(i',j')\in pred(i,j)} (l_{i'j'1})$$

$$f_{ij1} = s_{ij1} + d_{ij1}$$

$$l_{ij1} = f_{ij1}$$

where:

$$2 \le i \le m$$

$$1 \le j \le w_i$$

*Boundary Conditions.* Operation of source and sink processors is also special. First, source processors ($i = 1$) never have to wait for input data items because they are assumed to have all data items to begin with. Sink processors ($i = m$) never have to wait to forward an output data item because they consume their own output. These correspond to the boundary conditions of the pipeline:

$$s_{1jk} = l_{1jk-1}$$

$$f_{1jk} = s_{1jk} + d_{1jk}$$

$$l_{1jk} = \max_{(i',j')\in succ(1,j)} (l_{i'j'k-1})$$

$$s_{mjk} = \max_{(i',j')\in pred(m,j)} (l_{i'j'k})$$

$$f_{mjk} = s_{mjk} + d_{mjk}$$

$$l_{mjk} = f_{mjk}$$

where:

$$1 \le j \le w_i$$

$$2 \le k \le n$$

*Recurrence Body.* Processors of internal stages ($2 \le i \le m-1$) have to wait for input data items from all predecessors to arrive before the processing can start; once the processing is complete, the output data item has to be held until all successors are ready to receive it. This logic is formulated into the recurrence body:

$$s_{ijk} = \max_{(i',j')\in pred(i,j)} (l_{i'j'k})$$

$$f_{ijk} = s_{ijk} + d_{ijk}$$

$$l_{ijk} = \max[\max_{(i',j')\in succ(i,j)} (l_{i'j'k-1}), f_{ijk}]$$

where:

$$2 \le i \le m-1$$

$$1 \le j \le w_i$$

$$2 \le k \le n$$

Comparing the above with the recurrence relations for the linear pipeline in Section 2.2, we can see that the main difference between the two is the introduction of additional *max* functions for the parallel pipeline in order to model synchronization behavior for splitting and merging of the data flow. The transformations developed in Section 3 for modeling communication overheads and buffers also apply to DAG pipelines.

## 5. EXPERIMENTS

The recurrence models presented in the previous sections have been implemented as a single C program. Two simple experiments have been conducted to verify that the recurrence relations correctly characterize the behavior of asynchronous pipelines. In this section, we will first introduce an abstraction that allows applications to specify the pipeline to be modeled as a set of parameterized stations. We then describe the implementation and present results of the experiments.

### 5.1. Station Abstraction

In Section 2 we have shown how data communication overhead and extra buffers can be modeled by using simple transformations. However, for complex application pipelines, the corresponding abstract pipelines would be very large and complex due to expansions resulting from the transformation. To help control this complexity, we introduce a *station* abstraction in this section.

We define two types of stations which can be used by applications to describe their pipelines. A type 1 station is shown in Figure 7. The type 1 station is for representing intrastage computation of an application pipeline, and it consists of input buffers, receiving overhead $t_r$, actual computation $cp$, and sending overhead $t_s$. The receiving overhead $t_r$ is the time for the computation process to gain access to an input data item from the communication process. The sending overhead $t_s$ is the time associated with passing an output data item to the communication process. An input buffer is characterized by its size and delay overhead. A station will have one set of input buffer for each of its predecessors. The computation process $cp$ is characterized by a processing delay distribution.

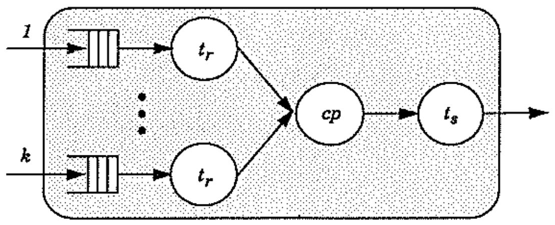

Figure 7: Computation Station

A type 2 station is shown in Figure 8. The type 2 station is an abstraction for the interstage communication in an application pipeline. It consists of a buffer followed by the main communication process $cm$. The buffer is described by its size and a delay overhead, and the communication process has a processing delay with some distribution.

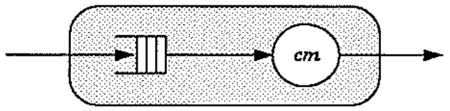

Figure 8: Communication Station

## 5.2. Implementation

The current implementation provides the following main functions:

- It allows any DAG to be specified as an asynchronous pipeline.
- Each abstract station is specified using parameters as defined in Section 5.1.
- It allows processing delays to be specified as either a normal distribution or as a sequence of values stored in a file.
- The average cycle time and a complete trace of finish time and leave time are produced.

Evaluation of the recurrence relations is the major computational cost of our analysis method. Compared to traditional discrete event simulations, abstraction using recurrence relations already avoided those overheads associated with the management of queues and events. However, a brute force implementation may be very inefficient. For example, one may implement the evaluation as a recursive function, but due to the double-indexed nature of the recurrence relations, the implementation will be difficult. Furthermore, it is well known that a recursive implementation of a computation is less efficient in time than its non-recursive implementation. We have developed a very efficient iterative algorithm for the evaluation of the recurrence relations. The key to the algorithm is the efficient evaluation order determined from the dependencies among the recurrence variables.

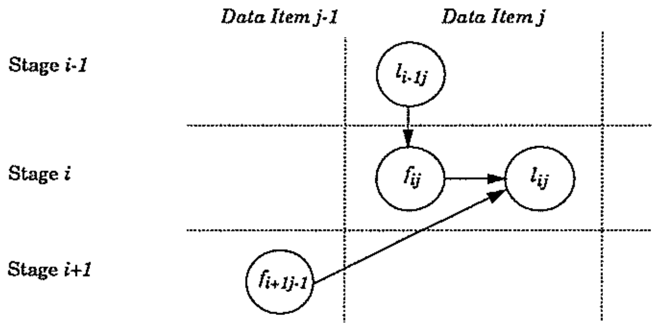

Figure 9: Evaluation Dependency

Figure 9 illustrates the general evaluation dependencies for $f_{ij}$ and $l_{ij}$, where an arrow ($\rightarrow$) means that the evaluation for the variable at the tail of the arrow has to precede that of the variable at the head. It is easy to see that the following pseudo code defines an iterative structure that is consistent with such dependencies for all the variables:

```
var $m$: integer; {total number of stages}
    $n$: integer; {total number of data items}
    $f_{ij}$, $l_{ij}$: real; {$1 \le i \le m$, $1 \le j \le n$}
for $j = 1$ to $n$ do
    for $i = 1$ to $m$ do
        begin
            compute $f_{ij}$;
            compute $l_{ij}$;
        end.
```

Computation of $f_{ij}$ and $l_{ij}$ is defined by the recurrence relations given in Section 2.2. A similar sequential evaluation order is used for the recurrence relations of DAG topologies.

All user interactions with the current implementation is character-oriented. The implementation can be improved by adding a number of features: (1) a graphical interface that allows pipeline topologies to be specified by simply dragging and connecting blocks, (2) support for parameters of each station to be easily edited from the graphical interface, and (3) a post-processor that allows performance results to be plotted graphically in different metrics and styles.

## 5.3. Experiment Results

For experiments, both the recurrence model and discrete event simulation were applied to a 3-stage linear pipeline and a 3-stage pipeline with two processors at the second stage, referred to as a parallel pipeline. The findings are summarized next with discussions. We use $cp_i$ to denote a computation station (type 1 as defined in Section 5.1) and $cm_i$ to denote a communication station (type 2 as defined in Section 5.1). The exact topologies of the two example pipelines are shown in Figure 10 represented as abstract stations.

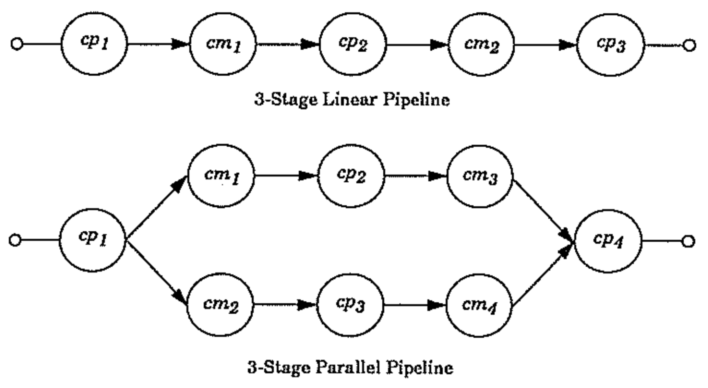

Figure 10: Example 3-Stage Pipelines

The particular simulator used was BONeS[‡]. In this study, all stages (computation/communication)

[‡]The Block Oriented Network Simulator is a software package for modeling and simulation of communication networks from COMDISCO Systems Inc.

are assumed to have a normally distributed processing delay with mean equal to 50[§]. The standard deviation is varied from 0 to 50 with step size 10, and four buffer sizes of 0, 1, 10, 20 are examined. Buffer sizes are always the same across stages. In all cases, both simulation and the recurrence model are run with more than 9000 data items to ensure statistical significance for the results. Figures 11 and 12 present respectively the results for the linear and the parallel pipelines from the recurrence model. The same set of results were also obtained with the BONeS simulation model, and they match almost exactly with results of the recurrence model. Hence, the simulation results are omitted from the plot. However, it is important to note that the recurrence model is at least an order of magnitude faster than the BONeS simulator under very similar conditions. The metric used in these plots is the average cycle time calculated after the pipelines have been filled up (i.e. $AC^+$).

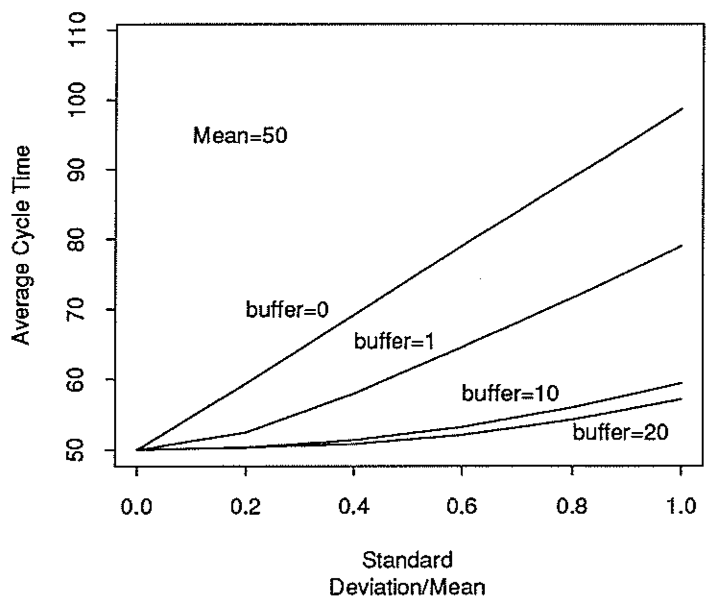

Figure 11: Linear Pipeline

First we examine the linear pipeline results shown in Figure 11. When processing delay for all stations is constant at 50 (i.e., standard deviation = 0), the average cycle time is 50 regardless of buffer size. As the standard deviation is increased, and there is no buffering, the average cycle time shows a linear increase; with buffer size 20, the increase in average cycle time is very little. Thus, the family of curves (for buffer sizes 0,1,10,20) shows that the increase of average cycle time with the increase in standard deviation is slower with larger buffers. This result fully conforms to our expectation of the effects of buffering. First of all, due to the data dependency in pipeline computation, a slow-down at any one stage will affect all the downstream stages by delaying the

[§]According to the normal distribution, there are a small number of samples of negative value. These samples are clipped to 0 to maintain their validity as delay samples in both the recurrence model and the BONeS simulation.

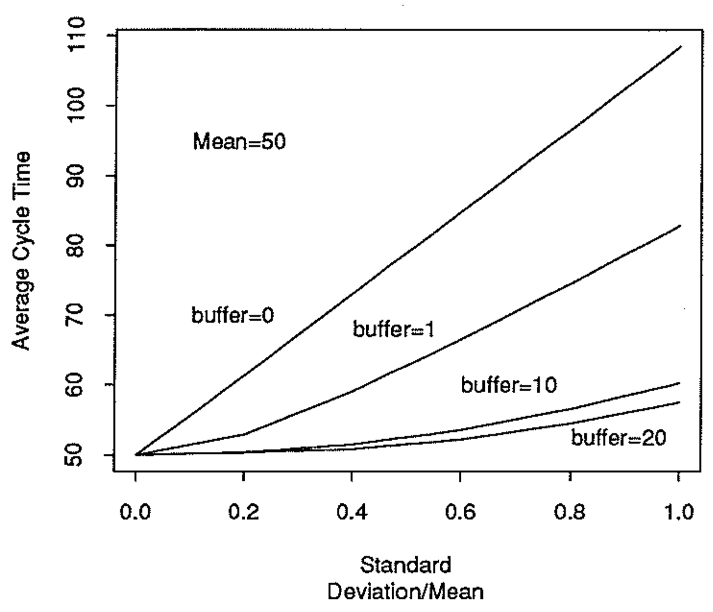

Figure 12: Parallel Pipeline

arrival of subsequent data items at those stages; furthermore, since there is no buffering, a faster stage is always forced to wait for its slower neighbors (to synchronize on per data item basis) and a slowed-down stage has no chance of catching up because it cannot buffer data items. Therefore, increased variation in processing delay directly translates into longer average cycle time of the pipeline as represented by the line labeled buffer=0 in Figure 11. On the other hand, when sufficient buffer space is introduced, short term variations of processing delay (as dictated by the distribution) do not have much impact on the average cycle time, as indicated by the curve labeled buffer=20 in Figure 11. This is because a stage is now rarely forced to wait in order to synchronize with its neighbors. This trend suggests that as the buffer size goes to infinity, the average cycle time for the pipeline would approach the mean of the normal distribution. However, the decrease in average cycle time with respect to increased buffer size does not seem to be linear within the ranges of buffer size and standard deviation observed.

In the case of a parallel pipeline, Figure 12 shows a similar trend, that is, the average cycle time increases linearly with increase in standard deviation when buffer size is 0. As larger buffers are used, average cycle increases slowly with increasing standard deviation. Comparison of Figures 11 and 12 show that for the same processing delay and buffer size, a parallel pipeline has a larger average cycle time than that of a linear pipeline except when standard deviation is zero. This is correct, because in the case of parallel pipelines, the waiting resulting from splitting and merging of stages contributes to a longer average cycle time. This kind of waiting is due to synchronization among parallel data streams, and it cannot always be eliminated by using extra buffers.

## 6. DISCUSSIONS

We can think of many ways in which the recurrence model can be applied to the study of issues related to asynchronous pipelines. Several ideas are discussed in this section.

As we have demonstrated, the recurrence model developed is an accurate performance evaluation tool. For a given asynchronous pipeline system, the topology and the number of buffers at each stage can easily be obtained; then the processing delay matrix can be derived either from a known distribution or from trace data collected from other experiments. Given this information the recurrence model can readily be applied and any desired performance measures can be obtained.

Another potential area of application for the recurrence model is optimal design of asynchronous pipeline systems. In such applications, a global optimization strategy has to be defined, which will maintain constant some of the parameters of the system (e.g., topology, number of stages and distribution of processing delay) while systematically varying other parameters (e.g., number of buffers at each stage). The recurrence model can be used to evaluate performance for each of these settings to determine an optimal setting according to a specified set of criteria.

One of our goals is to use the recurrence model to study the interprocess communication issues for PTV (Pipelined Televisualization). In particular we need to understand how the flow and error control mechanisms within a transport protocol affect PTV application performance. We can use type 2 stations to model interstage communication provided by the transport protocol. The processing delays for these stations can be derived from the underlying network characteristics and the flow and error control overhead. We will be able to observe the impact of flow and error control on the performance of a PTV pipeline by varying certain control parameters while keeping the rest of the parameters constant, and to explore the trade-offs involved with using different control strategies.

### 6.1. Related Work

Greenberg, Lubachevsky and Mitrani [9] very recently reported their work on parallel simulations that also utilized recurrence relations. The authors took a new approach to parallelizing simulations of queuing systems. First, they formulated simulations of queuing systems as solutions to a set of recurrence relations. Then they presented methods for solving the recurrence relations using multiple processors. The queuing systems studied include the G/G/1 queue, queueing networks with global FCFS structure, and a variety of general networks of G/G/1 queues (e.g., with non-FCFS discipline and feedback). It was shown that a much higher level of parallelism in simulation can be achieved with recurrence relations than previously possible with standard event-list approach.

Our work has been motivated by the need for an efficient and accurate model for analysis of distributed asynchronous pipelines. The systems under consideration always have finite buffers as opposed to infinite buffers of G/G/1 queues. We also need to define exactly how to model the intrastage computation and the interstage communication in pipelines. These high level differences have led to the following specific differences between the recurrence relations derived:

- Given our emphasis on the asynchronous behavior of the pipeline, we did not model the arrival process explicitly as in [9]. But it can easily be modeled by an additional input stage with its processing delay distribution equal to the interarrival time distribution.
- Finite buffers are modeled differently. In [9], the authors derived a separate set of recurrence relations for a series of queues with bounded buffers. These relations were of higher order. We introduce a transformation that models buffers by additional abstract stages, thus avoiding the need for separate higher order recurrence relations for different buffer sizes.
- When dealing with queuing networks with parallel queues, the authors in [9] had concentrated on only G/G/1 queues (i.e., with infinite buffers). Therefore, no backward dependency had to be modeled by their recurrence relations. In the case of asynchronous pipelines, if the buffer of the successor of a stage is full, the stage will have to wait. This dependency is explicitly modeled in our recurrence relations.

The recurrence relation approach had also been used earlier in somewhat limited ways for studying G/G/1 systems (e.g., see [10]). Not only were the buffers assumed to be infinite, the objective was limited to deriving waiting time for jobs rather than a complete trace of the system (also pointed out in [9]).

## 7. CONCLUSION

In this paper we have presented a recurrence model as an efficient tool for performance analysis of asynchronous pipelines. We have also reported experimental results that verified the set of recurrence relations and transformations. We emphasize the model's strength of being able to provide very accurate modeling while requiring very little computation power. Particularly, the DAG model is powerful enough to apply to any parallel pipeline with a DAG topology, which is an essential component of many distributed computing systems. We also outlined a number of ways that the recurrence model may be used to study various issues in the engineering of asynchronous pipelines.

In the light of recent work by Greenberg et al, we can consider to extend the current model to the cases where dynamic routing of data items is allowed and to explore parallel computation algorithms for the recurrence relations.

## Acknowledgements

The authors would like to thank many individuals at the Computer and Communications Research Center for their help in using the BONeS simulator. Particularly Milind Buddhikot's expertise helped in removing some really subtle bugs in the simulation model. We appreciate the useful comments from Dr. Mark Franklin that have helped improve the presentation of this paper.

## References

[1] Kogge, Peter M., *The Architecture of Pipelined Computers*, Hemisphere Publishing Corporation 1981.

[2] Hwang, K., and Z. Xu, "Multipipeline Networking for Compound Vector Processing", *IEEE Trans. Computers*, Vol. 37, No. 1, January 1988, pp. 33–47.

[3] Gajaski, Daniel, et al., "CEDAR", in Tutorial, *Supercomputers: Design and applications*, edited by Kai Hwang, Computer Society Press 1884, pp. 251–275.

[4] Carriero, Nicholas, and David Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed", *ACM Compt. Surv.*, Vol. 21, No. 3, Sept. 1989, pp.323–358.

[5] Hwang, Kai and Faye' A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill 1984.

[6] Dasgupta, Subrata, *Computer Architecture A modern synthesis, Volume 2: Advanced Topics*, John Wiley & Sons 1989.

[7] Dubey, P. K. and M. J. Flynn, "Optimal Pipelining," *J. Parallel and Distributed Computing* 8, 10–19 (1990).

[8] Gong, Fengmin, *Segment Streaming for Efficient Pipelined Televisualization* (dissertation proposal), Washington University Computer Science Department, technical report WUCS-90-39, St. Louis, November 1990.

[9] Greenberg, Albert G., et al., "Algorithms for Unboundedly Parallel Simulations", *ACM Trans. Compt. Systems*, Vol. 9, No. 3, August 1991, pp. 201–221.

[10] Baccelli, F., et al., "Acyclic Fork-Join Queueing Networks", *J. ACM*, Vol. 36, No. 3, July 1989, pp. 615–642.

[11] Kung, H.T., "The CMU Warp Processor", in *Supercomputers Algorithms, Architectures, and Scientific Computation*, (eds.) F.A. Matsen and T. Tajima, University of Texas Press, Austin 1986, pp. 236–247.

[12] Meng, Teresa H., *Synchronization Design for Digital Systems*, Kluwer Academic Publishers 1991.