Pipelining—The generalized concept and sequencing strategies

by C. V. RAMAMOORTHY and K. H. KIM
University of California
Berkeley, California

INTRODUCTION

In this paper, we generalize the concept of pipelining to increase throughput, processing speed, resource utilization and reliability. Its wide application spectrum is demonstrated and major design problems such as sequencing, reconfiguration, etc., are indicated. A scheme called the dynamic sequencing and segmentation model (DSSM) is proposed as a solution for providing efficient sequencing with very low overhead. The model is analyzed under various realistic environments and its performance is evaluated.

Pipelining can be defined as a technique of imbedding concurrency in a computer system by implementing it in the form of a pipeline, a configuration of independent autonomous units each of which is dedicated to perform a specific subfunction in an overlapped mode with others. An autonomous unit or a segment of a pipeline is also called a pipeline-segment or a facility-segment. Pipelining has emerged as an important aspect of computer architecture especially of scientifically oriented computers. A large number of computers have been built with one or more pipelined functional units.\(^1,2,3,11\)

Pipelining, however, still remains as an ad hoc procedure and a specialized technique of exploiting computational parallelism. Total parallel processability in computations has not been fully exploited in conventional pipeline systems. In this discussion, we classify the approaches to parallelism exploitation into two categories: (1) a passive approach, and (2) an active approach. In the passive approach the parallelism is exploited without any modification of execution sequences within a program, while in the active approach the exploitation is achieved by the automatic detection of the parallelism inherent in the program at various levels and the judicious sequencing of these detected parallel processable tasks. Conventional pipelining has mostly been confined to the passive approach to exploiting the local parallelism, i.e., the parallelism between adjacent tasks.

Pipelining in a functional unit is achieved in two steps: (1) segmenting the functional unit into a number of facilities (segments) each requiring generally the same amount of execution time and (2) streaming independent jobs through these segments.

Levels of pipelining

The principle of overlapped concurrent operation can be employed effectively at various levels in the computer architecture.

Pipelining at the gate level is exemplified in the design of the instruction processing unit (IPU). The IPU can be decomposed into various functional segments, namely, instruction fetch, instruction decode, address generation, etc. (Figure 1). It takes one minor cycle for a task (instruction) to pass through each segment. Thus after a stream of tasks enters this pipeline, the pipeline starts outputting one task per minor cycle. Microprogram prefetch, that is, overlap of decoding the current microinstruction with fetching the next microinstruction is another example at this level.

The next level for the application of pipelining is the subsystem level, with the pipelined arithmetic units being the typical examples. Pipelined ADD, MULTIPLY, DIVIDE and SQUARE-ROOT functions have been in existence in a number of contemporary computer structures. Figure 2 is the conceptual representation of the operation of the DIVIDE unit of IBM 360/91, where, as \(D_i\) iteratively approaches 1, \(N_i\) approaches \(N/D\), the quotient.

Generalization

So far, the design practices have confined the principles of pipelining (overlapped concurrent operations) to facility-segments with almost equal processing times and whose control and data flows are linear or unidirectional. Since computational sequences cannot always be divided into equal processing time-segments and since recursion and iteration are common computational modes, it would be desirable to remove the restrictions and extend the precept of pipelining. We shall next discuss the concomitants of this generalization. We shall characterize the generalized pipelining into the following aspects.

The first criteria would be segment-lengths (processing times) of each facility-segment in the pipeline. These processing times can be the same or variable.
is an example of a pipeline with two variable-length segments (CPU and I/O).

The second factor for characterizing pipelines is the direction of control and data flow. It could be unidirectional (e.g., the IPU) or bi-flow or bidirectional (e.g., IBM 360/91 DIVIDE unit).

The third criteria would be degree of replication of the pipeline. There may be several pipelines which perform identical operations on different operands (e.g., TI-ASC's arithmetic pipes).

The fourth characterization of the pipeline system is its reconfigurability. The system could be non-reconfigurable (e.g., IBM 360/91) or dynamically reconfigurable. Figure 3 shows an example of a replicated reconfigurable pipeline as implemented in the TI-ASC computer. There are four identical pipelined arithmetic units (AU's), each consisting of eight segments. All AU's would perform the same operation. Four segments among eight ones in each AU can be configured into a pipeline configuration for FIXED POINT MULTIPLY at one time while at another time six of them could be configured into a FLOATING POINT ADD pipeline. Although examples show linear pipelines, configurations in general need not be linear but could be planar (two-dimensional) or multi-dimensional.

The facility-segment may be not only hardware but also hardware-software complex of any complexity. The latter type can be often seen in the pipeline of the system level (Figure 4). A number of special-purpose computer networks can be also looked upon as this class of pipelining. The development of various pipeline systems corresponding to combinations of these characteristics is rather an evolutionary step in the design of pipeline systems. The fully generalized concept of pipelining encompasses both conventional parallel processing and pipelining as subsets. Further improvement of resource utilization is achievable. The fully generalized pipelining is capable of utilizing resources which would have been idle in the conventional approach. In the case of a pipeline which is reconfigurable in multi-dimensions, those resources could often be configured into a certain pipeline and used for computations resulting in the increased utilization. Improvement of reliability is another gain of generalization. Reconfigurability and replication in the generalized pipeline provide the sound basis for the incorporation of graceful degradation as well as fault diagnosis and recovery techniques. An example of the generalized pipeline, with dynamic reconfiguration and replication is shown in Figure 5.

**Design problems**

Now that generalization of the concept has been discussed, we shall consider problems involved in designing such a pipeline system. Some of the common problems in the design of pipeline systems are the following: (1) design configuration of pipelines, (2) sequencing strategy, (3) resolution of storage conflicts, (4) determination of program suitability, and (5) efficient execution of sequences (efficient utilization of resources).

The design configuration of a pipeline system involves the determination of proper segment-lengths, their processing speeds, the degrees of their replication and their reconfigurability to achieve the necessary performance criteria (throughput rate) on a spectrum of jobs.

The basic problem in sequencing is to decide the order of executions of independent tasks such that their completion time is minimized. The performance of a pipeline system is highly sensitive to sequencing strategies. Figure 6 illustrates the effect of sequencing in the case of a simple pipeline consisting of two segments. The sequencing problem is in general twofold: (1) the development of the algorithms and (2) the overhead generated in using it. In other words, one has to develop an efficient algorithm such that the overall execution time including the overhead is minimized. There is an obvious trade-off here since the closeness to optimality of the algorithm increases its run-time overhead.

In the next section, an effective approach is introduced.

Another design problem is concerned with storage conflicts. This problem generally occurs when more than one task being executed concurrently needs to access the same memory module. The memory conflicts can be classified into two types. The first type is the local conflict occurring between consecutive instructions in a program when they are executed concurrently and need to access the same memory module. The other type is the global conflict oc-
SEQUENCING STRATEGIES

The next design problem is the one of program suitability. Some programs may reconfigure the pipeline excessively without utilizing each configuration effectively. It would be desirable to develop the techniques for restructuring such ill-formed programs so that the restructured ones may become more suitable (reduce total execution time) for the specific type of pipelining (Figure 7).

Efficient execution of sequenced tasks is another design problem. Efficiency consideration arises in the process of activation and synchronization of tasks. These bookkeeping operations must be minimized in both extent and frequency.

In the rest of the paper, one of these common problems, namely the sequencing problem is discussed in detail with a description of a new approach of resolution.

SEQUENCING STRATEGIES

Concept of the dynamic sequencing and segmentation model (DSSM)

As mentioned before, an efficient sequencing strategy is essential to the pipeline system, but its run-time overhead becomes expensive. A run-time overhead is accumulated in concurrent processing since a number of decisions regarding resource allocation, etc. must be made dynamically. Actually this has been the main obstacle in parallelism exploitation.

To overcome this problem a scheme called the dynamic sequencing and segmentation model (DSSM) is developed which overlaps the unproductive overhead (bookkeeping and administrative) computations with the execution of computational tasks so that the effect of overhead is greatly diminished and an efficient exploitation of parallelism is realized.

In this system a dedicated unit for sequencing, called the sequencer, operates on one segment of the computational job while the processor, called the executor, executes the previous (already sequenced) segment (Figure 8).

The DDSM itself is an example of pipelining at the high (system) level. For the purpose of sequencing and segmentation, computational jobs are modelled by the parallel task graph (PTG). A task is a set of instructions which, once initiated, can be carried out to its completion by a pipeline without the need for additional inputs. Thus one can consider a single instruction a task. A task can be often partitioned into a set of subtasks, each of which can be executed by one pipeline-segment. In the rest of the paper, no distinction is made in use between a task and a subtask. A program or a job in the pipeline system consists of a set of tasks. A PTG is a loop-free directed graph in which (1) each node represents either a task or a set of tasks in a loop, and (2) a transition exists from the node i to the node j if and only if the task j depends for its initiation on a result generated by the task i.

In the PTG, each loop or a strongly connected subgraph is abstracted into a single node, or task. A sequencing procedure is developed to execute the loop-free PTG. If any loop or strongly connected subgraph is large (many nodes and/or large execution times), its execution sequence during

![Figure 5](image_url)  
Figure 5—An example of the generalized pipeline

![Figure 6](image_url)  
Figure 6—An example of the effect of the sequencing

![Figure 7](image_url)  
Figure 7—An example of program restructuring

![Figure 8](image_url)  
Figure 8—The concept of the DSSM
Design principles of the DSSM

Dynamic sequencing

The sequencer determines the execution sequence of a segment of tasks while a previous segment of tasks is being executed. This sequencing technique possesses high adaptability to dynamically varying environments, which is superior to the static (pre-) sequencing. Presequencing or static sequencing may result in inefficient executions because it cannot take into account the variation of job characteristics over time, or the run-time changes in resource utilization patterns due to external interrupts, etc.

Basically, the sequencing problem can be formulated as follows. Given the configuration and characteristics of the pipeline, and a set of related tasks to be executed, it is required to develop an execution sequence (schedule) that reduces the total execution of the set of tasks. It is assumed that the task dependencies and the task execution times (or their estimates) are known. In this paper, we shall focus ourselves to the problem of scheduling and sequencing a set of related tasks in a single large program as represented by its PTG. The problem of sequencing unrelated and independent jobs is a simple case and we shall not consider this. It is well-known that the overhead for computing the optimum sequence increases almost exponentially with the size and the structure of the task graph. Moreover, the resulting sequence could at best be suboptimal because of variances in their predicted values of execution times during run-time, etc. A heuristic algorithm which produces a reasonably efficient sequence with much less overhead would be highly desirable. The heuristic should accomplish the following. It should update the set of ready tasks according to the precedence relationships and the availability of facilities and it should select a task from the ready set by applying various selection rules. One example of a selection rule is to calculate the length of the longest path in time from each of the ready nodes (tasks) to the exit node and then to select the node with the longest path among them.¹⁴ Since the suitability of any heuristic is highly dependent upon the system configuration and the characteristics of input jobs, comparative evaluation of algorithms under the given environment will be the reasonable approach for selecting a suitable algorithm. An example of the sequenced PTG is presented in Figure 10. Figure 10(a) shows an example of the PTG where each node (task) is associated with the necessary information for sequencing, the estimated time and the kind of facility required for the execution of the node (task). A double line in Figure 10(b) denotes the determined sequence of the parallel processable tasks competing for the same kind of facility. For instance, nodes 3 and 6 represent two parallel processable tasks requiring the same kind of facility, F₁ and they are sequenced into the execution order (3,6). The execution of the program according to this sequence is expected to take 105 time-units.

Dynamic segmentation

Segmentation here is also a dynamic segmentation analogous to the dynamic sequencing. It behaves according to the distribution of the work-load in the DSSM which is a dynamically varying characteristic.

Segmentation consists mainly of two parts: (1) To decide on the size of the next computation-segment, i.e., the number of nodes in the next segment to be sent to the sequencer, and (2) To select the nodes in the given task graph for that segment. The efficient segmentor has to pick up the appropriate size of segment for sequencing such that the sequencing time of the new segment is almost the same as the execution time of the current segment. Once a segment is selected, its execution time can be estimated and this is used to determine the size of the next segment. Therefore, it is desirable to know the nature of variation of overhead (time to generate an efficient execution sequence) with the size (number of nodes) of the segment analyzed (i.e., overhead characteristic).

![A PTG](image1)

![A Sequenced PTG](image2)

Figure 10—An example of the sequenced PTG
Pipelining—The Generalized Concept and Sequencing Strategies

The other factors that affect the overhead have been ignored for simplicity. Once the algorithm is fixed, this relation can be obtained by experiments and utilized in the DSSM.

Figure 11 shows an example of the overhead characteristic for a specific sequencing discipline. The curve represents a portion of the data obtained by the experiment with the randomly generated PTG's (refer to later section). The number of nodes contained in each PTG ranged between 4 and 120. We used the heuristic algorithm developed in Reference 10 for sequencing. The selection rule in this algorithm is basically to calculate the priority for each task \( i \) in the set of ready tasks by using the following priority function \( p(i) \) and then select the task of the highest priority.

\[
p(i) = \text{SIGN}(T(i) - T_s(i)) \cdot \text{MIN}(T(i), T_s(i))
\]

where \( T_s(i) = \text{MAX}_{j \in S(i)}[T(j)] \), \( S(i) \) is a set of successor tasks of the task \( i \), and \( T(i) \) = execution time of the task \( i \). The simulated DSSM was run on a CDC 6400 computer. One time-unit of overhead in Figure 11 is roughly equal to 1 millisecond taken by the simulated sequencer for analysis. The smooth curve in Figure 11 is an interpolated approximation of the data represented by the discontinuous line shown in it. If a certain segment is going to take 75 time-units for execution, the suitable size of the next segment will be of 21 nodes. Of importance here are the shapes of the curves rather than their numerical significance, since the latter depends upon the method of implementation. The curve approximates a polynomial. The reason for this is that the analysis time of a segment grows rapidly with the number of tasks (nodes) in it.

Next, the picking up the given sized segment from the task graph, is done by maintaining precedence relations between the tasks. A sophisticated method requiring a large overhead would not be favorable unless it makes the significant increase in the system performance. This appeared to be valid by experiments with two algorithms of different degrees of sophistication. One of them called the precedence

\[
\text{partition and outdegree (PPO) algorithm is shown below for the purpose of illustration. The principle is to select nodes in the order of precedence partitions and in the order of outdegrees within the same partition (Figure 12).}
\]

The other algorithm used in the experiments was the precedence partition and random assignment (PPR) algorithm which is different from the PPO algorithm only in that nodes within the same precedence partition are selected randomly. For the example PTG in Figure 12, this algorithm will randomly pick up either nodes (1) and (2) or nodes (1) and (3) as the next segment.

\[\text{Balancing the workload}\]

As was discussed in the preceding section, the method of selecting the segment to be sequenced should be carefully designed since the effectiveness of the DSSM is highly sensitive to it. When the execution time of the segment just analyzed is accurately known, the sequencer would select a new segment whose analysis time (overhead) is equal to the execution of the segment (just analyzed). If this state of affairs is sustained, that is, the execution time of the current segment is almost equal to the sequencing time of the next segment, the next segment is chosen by consulting the overhead curve (Figure 11).

On the other hand, when the execution times are not accurately known, then either the sequencer or the executer may occasionally have to wait for the other to complete its function. When there is a fixed amount of buffer storage between the sequencer and the executer, the segment already analyzed by the sequencer may be queued until the executer becomes available. If the execution of the current segment finishes before the sequencer completes its analysis of the next segment, the executer would be idle until the next segment is ready. In this case, the DSSM is said to be in the sequencer-overloaded state. On the other hand, when the executer is busy processing a segment, either the next segment just analyzed by the sequencer has to be queued or the sequencer would have to wait if there is not space left in buffer storage. If there is at least one segment queued in the buffer storage when the sequencer just completes the analysis of the next segment, the DSSM is said to be in the executer-overloaded state. In any of these cases, the performance of the DSSM may become somewhat degraded.

Many factors are ignored in the evaluation of the overhead.
may be unbalanced sometimes. In such a case the workload is not accurately known but roughly estimated, it is presumed that the sequencer-overloaded state may persist for a while. The situation is detected by the sequencer examining the status of the buffer storage each time it completes the analysis of a segment. Then the segmentor will select a larger segment of the size corresponding to the overhead equal to the sum of execution times of all segments queued up (including the segment just analyzed) so that the sequencer may complete the analysis of the next segment about the time when the executer completes executions of the segments queued. In the case of the sequencer-overloaded state, the following two cases can be considered. If the execution time of the segment just analyzed is accurately known, the segmentor takes a segment of the size corresponding to the overhead equal to the execution time of the analyzed segment, called a segment of the proper size. On the other hand, when the execution time is not accurately known but roughly estimated, it is presumed that the sequencer-overloaded state may persist if the segmentor takes a segment of the proper size. Thus the segmentor takes a segment of the size smaller than the regular size. This is then repeated until the workload distribution reaches the balanced state. In this way, the DSSM can resume the balanced state from any unbalanced state.

Behavioral analysis

By analyzing the overhead curve, two interesting properties of the DSSM can be discovered.

First, the stabilized segment-size is predictable. That is, the size of the computation-segment (number of nodes or tasks) becomes almost fixed after some transient time. We shall call this the stabilized segment-size.

The straight lines 1, 2, 3 of Figure 13 called the normal execution time curves represent the expected execution time of the segment depending upon the segment-size, when $T_s$ is 1.5, 2.5 and 4.0 respectively. Here $T_s$ denotes the average effective execution time per node in the given PTG and is obtained as follows:

$$T_s = \frac{T_e}{ADOP}$$

where $T_e$ denotes the average execution time per node and ADOP denotes the average degree of overlapped processing which is in turn defined as the average number of nodes being executed concurrently during the execution of the given PTG. More specifically, $T_e$ is the average amount of time required for executing one task (node) by the required facility in the executer. It can be interpreted as the quantitative measure of the level of a task which is directly proportional to the level of a pipeline. Provided that the executer is always busy, $1/T_s$ is the average number of nodes which the DSSM executes in a unit of time. $T_s$ is dependent upon the level of a task and the ADOP which are dependent upon the job-characteristic and the system configuration.

We shall call the crosspoint of the normal execution time curve with the overhead curve (excluding the origin) the stabilized point. Provided that the overhead curve has a monotonically increasing property, there will exist a unique stabilized point. For instance, the stabilized segment-size is 36 when $T_s=4.0$ in the diagram. The reason is as follows. For the sake of explanation, let $n_i$ denote the number of nodes in the $i$th segment picked up by the segmentor and $t_i$ denote the normal execution time corresponding to the size of the $i$th segment. Suppose $n_i$ is 30 which is smaller than the stabilized size 36. After $t_i$ is passed through the origin. The reason is as follows. For the sake of explanation, let $n_i$ denote the number of nodes in the $i$th segment picked up by the segmentor and $t_i$ denote the normal execution time corresponding to the size of the $i$th segment. Suppose $n_i$ is 30 which is smaller than the stabilized size 36. Now $n_i$ is determined from the point in the overhead curve corresponding to $t_i$ (=120) time-units of overhead. $n_0=32$ and $t_0=n_0 T_e = 128$. Similarly, $n_0=32$, $t_0=132$ which is greater than 36. By the same reasoning, it can be intuitively seen that $n_i$ iteratively converges to the stabilized size 36 in the case where $n_i$ is greater than 36.

Next we can determine the lower bound of the average execution time per node ($T_s$) in the DSSM. The line 1 of Figure 13 is the derivative of the overhead curve which passes through the origin. It corresponds to $T_s=1.5$. This $T_s$ multiplied by the ADOP which is constant, is the lower bound of $T_s$. Below this bound the execution of a job in the DSSM could result in the completion time longer than the one by the sequential execution without any segmentation and sequencing. Since $T_s$ is dependent upon the level of a task, there is a practical limit to the level at which the DSSM can be effectively applied. One approach to overcome this limit would be to reduce the overhead by the specialized implementation of the critical portion of the sequencer using firmware or hardware.

Performance evaluation and implementation

In this section, we present some recent results of performance evaluation studies carried out to validate the feasibility of the DSSM for implementation. This evaluation was done by simulation.

A PTG is the main input to the simulated DSSM and the connectivity matrix was adopted for the internal representation of the PTG.

In order to test a wide range of inputs, the random PTG generator (RPTGG) has been developed. It produces the
Pipelining—The Generalized Concept and Sequencing Strategies

A directed graph of random connection without loops, and the user can control the arc density and the pattern of the graph in order to tune it up to the desirable pattern based on intuition or experience. It is a useful tool for various studies in parallel processing. The basic algorithm is shown in Figure 14.

The overhead characteristics used was the one determined by the experiment in an earlier section of this paper. The execution time of each task was randomly generated using the average value \( T_a \). Simulations were performed for several values of \( T_a \) (1, 2, 5 and 10 time-units). One time-unit represents 1 millisecond taken by the simulated executor for execution. The whole simulator was prepared in FORTRAN and run on the CDC 6400 computer.

The first observation was the negative effect of the sophisticated segmentation algorithm. The benchmark algorithms used were the PPO and the PPR algorithms introduced earlier. By simulation, the PPO algorithm didn’t show any significant improvement of the gain compared to the PPR algorithm while its overhead was many times larger than that for the PPR algorithm. The PPO algorithm was discarded and only the PPR algorithm was used throughout the remaining simulation.

As a measure of performance, we used the following quantity, the gain by overlap.

\[
\text{Gain} = \left(1 - \frac{T_s}{T_f}\right) \times 100
\]

where \( T_s \) = Total completion time for the program in the pipeline system and \( T_f \) = Total completion time for the program in the sequential (non-overlapped) system.

The following diagrams show the typical results obtained (Figures 15(a), 15(b)).

Curves 1 and 2 represent the case where a job is sequenced as a whole without being segmented. That is, the system doesn’t have the segmentor. A whole PTG is analyzed at a time by the sequencer and then sent to the executor. Curve 1 is the case where the overhead is completely ignored, while curve 2 contains the full overhead. Therefore, curves 1 and 2 are the upper and lower bounds of the gain obtainable by the system without the segmentor.

Curves 3 and 4 represent the case of the DSSM. Curve 3 is the case where the overhead required for segmenting and sequencing the first segment, is completely ignored, while curve 4 contains the full amount of the overhead.

Curves 1 and 3 can be regarded as those situations in which jobs are continuously entering the system and the sequencing of the new job is fully overlapped with the execution of its preceding jobs.

On the other hand, curves 2 and 4 can be regarded as those in which jobs are entering at discrete intervals and the sequencing of the new job is not at all overlapped with the execution of preceding jobs. Therefore, the gain under the continuous operation will range between those of curves 1 and 2 for the system without the segmentor and between those of curves 3 and 4 for the DSSM.

It is noteworthy that the DSSM maintains a high gain close to the maximum upper bound of performance (curve 1). This implies that it is highly adaptive to a dynamically varying environment. This can be more precisely stated by using the following notations. \( G_i(n) \) denotes the gain represented by the curve \( i \) in the diagram corresponding to \( n \) nodes. \( AG(n) \) denotes the average gain obtained by the DSSM in the case of \( n \) nodes. \( AG(n) \) is obtained from the diagram as \( \frac{1}{N} \sum [G_i(n) + G_j(n)] \).\( N \) denotes a set of all instances of the

---

From the collection of the Computer History Museum (www.computerhistory.org)
discrete variable \( n \) which represents the number of nodes. Then, the efficiency of the dynamic sequencing and segmentation, \( Q \), can be defined as follows:

\[
Q = \frac{\text{AVG}_{n \in \mathbb{N}} \left[ \frac{\text{AG}(n)}{G(n)} \right]}{\#(N)}
\]

where \( \text{AVG}_{n \in \mathbb{N}}[X(n)] \) represents the average value of \( X(n) \)'s corresponding to all \( n \in \mathbb{N} \), and \( \#(N) \) represents the cardinality of \( N \). As indicated by the diagram, the values of \( Q \) obtained from simulations were mostly over 85\% where \( T_s \) was larger than 2 time-units.

As expected from the analysis of the overhead curve, the simulation results showed poor performance when the average execution time per node was less than the lower bound (when \( T_s = 1 \) in Fig. 15(b)).

In short, simulation confirmed the validity of the analysis as well as the steady high performance of the DSSM.

The implementation of the DSSM must be based on the characteristics of the environment. The level of pipelining, the capability of multiprogramming and the multiplicity of facilities are the prime factors which affect the implementation. Also the level of language in which the input tasks are prepared is one of the factors which affect the method of parallelism detection. It is much simpler to detect parallelism from the machine language program than from the high level language program. In the case of the high level language input, the implementation varies depending upon whether the program is compiled and executed or directly interpreted.

The physical capacity of the LAU consisting of the sequencer and the segmentor may be much smaller than the one of the executor because (1) the algorithms employed for the sequencing and the segmentation normally don’t make use of any complex operations and (2) the input data to this unit is an abstract model of the program instead of the program itself. It could be a special purpose processor with a small storage whose critical functions are hardware- or firmware-implemented.

The executor may vary from a set of small processors to a set of large scale processors.

**Extension of the DSSM**

**Extensibility of LAU**

The sequencing and the segmentation are not the only functions which can be provided by the LAU. In other words, the LAU can be easily extended to perform other desirable look-ahead functions. Typical examples are the look-ahead for branching decisions and the fault diagnosis and recovery. It is often possible that the execution of one segment provides enough information to make a branching decision in advance before the decision point is encountered in later segments. Also the LAU can periodically diagnose the part of the executor without interfering with the execution. In addition, it can assist dynamic storage allocation.

Explorations of various useful look-ahead functions and the relevant implementation techniques are under way.

**Pipelined computer network**

Concepts of the pipelining and the DSSM can be effectively applied to a reconfigurable computer network which consists of a number of processors. The processors in the network can be partitioned into two sets according to their functions. One set acts as sequencers while the other set acts as executers. Since there can be several sequencers, more than one sequencer may be employed to the sequencing of several segments of a single job (this mode of sequencing is called the parallel sequencing) or a set of independent jobs.

The ratio of the number of executers to the number of sequencers is variable and thus the system becomes highly adaptive to the varying workload distribution. In other words, each processor may function as a sequencer at one time and as an executer at another time. The function of each processor is determined depending on the workload distribution in such a way that the system keeps the balance (Figure 16).

Amongst the executers, each processor functions as a facility-segment of a reconfigurable pipeline. Consequently, a set of facilities available for configuring pipelines are dynamically changing. New techniques must be developed to solve problems in pipeline configuration, sequencing algorithm, etc.

Because of the consistent decreases in the hardware cost, a computer network becomes feasible and attractive in the future, especially in the form of miniprocessor networks where each miniprocessor takes a specialized functional role, possibly via microprogramming.
CONCLUSION

The generalized pipelining appears to be a promising approach to further increase the throughput rate, processing speed and system reliability. A need exists for efficient implementation techniques. Pipelined computer network is a further generalization and needs to be investigated in detail.

REFERENCES
