A SCHEDULING ALGORITHM FOR THE PIPELINED IMAGE-PROCESSING ENGINE

by

Charles V. Stewart
and
Charles R. Dyer

Computer Sciences Technical Report #664

September 1986

A Scheduling Algorithm for the Pipelined Image-Processing Engine

Charles V. Stewart
Charles R. Dyer

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

In this paper we present a heuristic scheduling algorithm for the National Bureau of Standards' Pipelined Image-Processing Engine (PIPE). PIPE is a special purpose machine for low-level image processing consisting of a linearly connected array of processing stages. A program is specified as a planar directed acyclic graph where each node represents an operation on an image, and each arc represents an image output by one operation and input to another. The algorithm uses the greedy approach to schedule operations on a stage. It uses several heuristics to control the movement of images between stages. The worst case time for the schedule generated by the algorithm is $O(N)$ times the optimal schedule, where N is the maximum width of the graph. Several examples are given where the algorithm generates a nearly optimal schedule.

The support of the National Bureau of Standards under Order No. NANB510565 and the National Science Foundation under Grant No. DCR-8520870 is gratefully acknowledged.

Table of Contents

1. Introduction

Recently, a number of one-dimensional array architectures for low-level image processing have been designed including the Cytocomputer [1], PIPE [2], Warp [3], Pixie-5000 [4] and the Vicom VMV-1000 [5]. All of these machines consist of a linearly connected array of processing stages. Each stage contains some local image buffer memory and specialized hardware functional units for performing image processing operations such as point and neighborhood functions. One of the main drawbacks to effectively using these machines, however, has been the lack of tools for programming such complex systems. In particular, the major problem is how to take a program consisting of a set of image operations and assign these operations to the processing stages. This scheduling problem requires the specification of both *when* and *where* operations should be performed, taking into account both the architecture of individual stages and the communication constraints imposed by the linear connectivity of the stages. This is a generalization of the job-shop scheduling problem [6]. Similar scheduling problems also arise in scheduling arithmetic expressions on linear arrays of processors [7].

In this paper we consider the scheduling problem for one specific architecture, the National Bureau of Standards' Pipelined Image-Processing Engine (PIPE) which has been designed for real-time, low-level image processing applications [2]. PIPE accepts a 256 by 256 image as input every 1/60 second from an attached camera. The images are processed in a series of processing stages. Each stage has data communication pathways to itself and to its two adjacent stages. Each stage can perform a number of arithmetic, neighborhood and logical operations on one or more images within this 1/60 second. These operations are performed at every point in an image. PIPE outputs its processed images either to a host processor or to another processor, called ISMAP, which is designed to extract more global and symbolic features of the images. Thus, PIPE is designed for extracting low-level image features in real-time.

Because the individual operations of PIPE are applied to an entire image, it is useful to represent a PIPE program using a directed, acyclic graph (DAG). The nodes of the graph represent the individual operations to be performed. Each operation is applied to an entire image. The arcs indicate the dependencies between nodes, representing images output by one operation and input to another. Present methods of programming PIPE require mapping such a graph onto the PIPE hardware by hand. This paper presents a heuristic algorithm for automatically generating this mapping. The algorithm could be used as part of a PIPE compiler.

The present version of the algorithm works on planar DAG's. It maintains a frontier list of operations that are ready to be scheduled. Initially, this list contains the children of the input images. During each time unit the algorithm schedules as many operations from the frontier list as possible. In order to schedule an operation the input images must be available on the stage, there must be a hardware functional unit available that can perform the given operation, and there must be a communication pathway off the stage which is available so that the resulting image can be either stored in one of the stage's image buffers or shifted to another stage. After all the possible operations in a time unit have been scheduled, the exact output stage locations for the results of the operations are chosen. Several heuristics are used to choose among the output options. The algorithm continues to schedule time units until the frontier list is empty. At this point a valid mapping of the graph onto PIPE will have been generated.

The remainder of this paper presents the details of PIPE, the scheduling algorithm for mapping a graph onto PIPE, and an analysis of this algorithm. Specifically, Section 2 describes the details of the PIPE architecture. Section 3 presents a more formal description of the program graphs which are assumed to be input. Section 4 gives an outline of the algorithm. Section 5 describes several ideas that are important in understanding the detailed description of the algorithm given in Section 6. Section 7 presents several examples. Section 8 analyzes the algorithm in terms of the execution time of the schedule generated for PIPE and the correctness of the scheduling process. Section 9 describes some potential enhancements to the algorithm.

2. PIPE Architecture

In this section we describe the architecture of PIPE. For more details, see [2,8]. PIPE is designed for parallel processing of two-dimensional image data only. All operations are applied to one or more images, and all data paths communicate images. The organization of PIPE is a linear sequence of processors, called stages, plus two specialized stages for input and output. Each stage has its own local image memory, called image buffers, which is used to store intermediate image results. There is no shared global memory.

Image data communication between stages is accomplished via direct interconnections from each stage to its two adjacent neighbors, and to itself as shown in Figure 1. Because all operations performed in each stage take a fixed amount of time, communication between stages can be performed synchronously at the end of each time unit. Thus, at the end of each time unit each stage, i, can output up to three images independently. One output goes to stage $i+1$ via the forward (fwd) path. A second goes to stage $i-1$ via the backward (bwd) path. A third is output to stage i itself via the recursive (rec) path. In addition to direct communication with each stage's nearest neighbors, PIPE has two "wildcard" busses, called VBUS A and VBUS B, which may be used to output an image to one or more arbitrary stages. These wildcard paths are not shown in Figure 1. Only one image may occupy each of these wildcard paths during any time unit.

The input stage receives images from up to two input devices. It outputs an image either to stage 1 via its forward path, or to other stages via one of the wildcard busses. The output stage, not shown in Figure 1, receives images from the last stage's forward path or from other stages via the wildcard bus. This stage outputs images to the host independent of the processing in the other stages.

A pipeline of operations can be performed on a sequence of images passing through PIPE as follows. Each stage performs a sequence of operations on an image it receives from the previous stage, and then passes its resulting image to the next stage for further processing. The final image output by the last stage exits PIPE through the output stage to the host. In this scheme there is a continuous, one-way flow of images through PIPE using the forward paths to communicate images between stages. However, because of the presence of the additional communication pathways in PIPE (i.e., the backward path, recursive path, and two wildcard busses), there is considerably more flexibility in defining sequences of image operations. That is, PIPE is in effect a linearly connected set of synchronous, MIMD parallel processors, where each processor is specially designed for performing image processing operations.

In order to simplify the design of the scheduling algorithm presented in this paper, we have ignored the possibility of using either of the wildcard busses for communication between stages. The wildcard busses will only be used to initially distribute the input images to the stages, and to move the final result images from the stages to the output stage.

Figure 2 shows an individual stage of PIPE. Within a stage three basic types of image operations can be performed. All operations in a stage take place within 1/60 second. The simplest is a pixel arithmetic operation in which each pixel is transformed into a new pixel value by a specified function. This function depends only on the given pixel's value and not its spatial position in the image. These operations are implemented via look-up tables which are shown in Figure 2 as LUT functional units. The second type of operation includes all arithmetic functions of two operands which compute the value of an output pixel as a function of the corresponding pair of pixels in the two input images. A simple arithmetic logic unit, denoted ALU in Figure 2, computes these results. Finally, 3 by 3 neighborhood operations can be specified which compute the value of an output pixel as an arithmetic of the nine values in a 3 by 3 block of pixels centered at the corresponding point in the input image. (The LUT's, ALU's and NBR's may compute binary operations as well, but this paper considers only arithmetic, not Boolean, neighborhood operations.) Each PIPE stage contains two hardware units for performing these types of operations; in Figure 2 these are shown as NBR functional units.

The hardware functional units in a stage can be grouped into two parts: those functional units that are before the stage's image buffers, and those that follow them. Currently, each PIPE stage can contain up to thirty two 256 by 256 image buffers. Because the number of image buffers available is large, we ignore in this paper the issue of image buffer management, effectively assuming unlimited buffer space in each stage.

The *pre-buffer* functional units are used to combine up to three input images into one using the LUT's and ALU's shown in Figure 2. Input to the image buffers is from either ALU B, one of the wildcard busses, or the DMA bus. Stage buffers can store images for an arbitrary length of time. At each time unit some selected buffers may be written into. All others retain their previous contents. This is important in maintaining results from prior operations for future use.

Following the image buffers there are two sets of operations which can be performed. First, an image can be input to an LUT functional unit (PRE-NBR LUT). The output pf this unit is then piped into *both* NBR's. The NBR's can compute separate functions. The second set of operations following the stage buffers allows images to be combined using an LUT and an ALU. The LUT (TVF LUT) receives one or two images as input. In the algorithm given below, the possibility of two images input to the TVF LUT is ignored. The ALU (ALU C) requires two images as input. The images input to the TVF LUT or the ALU C may come from either of the stage buffers or from any of the NBR's.

In summary, each stage contains a number of image buffers and three types of hardware functional units organized in a fixed manner. All of the operations performed within a stage are completed in 1/60 second, independent of the types of functions specified in each functional unit.

Finally, images can be output from a stage in several ways. The first method of output allows the contents one of the image buffers to be output to the host via the DMA bus (see the middle of Figure 2). The second method sends output to any subset of the fwd, rec or bwd paths, or either of the wildcard busses. The algorithm given here will only use the wildcard busses to move images to the special output stage; all intermediate result images are output using the fwd, rec and bwd paths. The output to each is determined independently and may come from either of the buffers, either of the NBR's, the TVF LUT, or the ALU C. The overall effect of these features yields a considerably more flexible design than a one-way pipeline.

3. DAG Representation of PIPE Programs

The PIPE architecture implies that all PIPE programs are collections of operations applied to entire images. The relations between these operations can be specified in terms of images which are output by one operation and input to another. Consequently, PIPE programs can be represented as a directed acyclic graph (DAG). A node in the DAG corresponds to a single operation on an entire image. Arcs in the graph associate the output image of one operation as the input image of another operation. An example DAG for a simplified version of the Canny edge detection algorithm [9] is shown in Figure 3. Each of the nodes in the DAG will be computed using either an ALU, LUT or NBR functional unit in a stage. Figure 4 shows a graph with the types of operations replacing the names for those operations.

Throughout the remainder of this paper we will only consider DAG's containing nodes labeled with their operation type names. Furthermore, DAG operation types will be designated using small letters (alu, nbr, lut), and hardware functional units in PIPE will be designated using capital letters (ALU, NBR, LUT). Notice that the nbr and lut nodes have only one incoming arc, and the alu nodes have exactly two.

In addition to the PIPE stage simplifications stated in the previous section, we also assume that all PIPE programs are given as *planar* DAG's. Furthermore, the DAG must remain planar after the addition of two hypothetical nodes outside of the graph. The first node has in-degree 0 and has arcs for each input image. The second node has out-degree 0 and has arcs into it from each result node. A result node is a node where the result of the computation specified by that node is to be output from PIPE. Note that result nodes do not necessarily have to be leaf nodes of the original DAG (i.e. the DAG before the addition of the hypothetical nodes), but all leaf nodes are result nodes.

The planarity assumption simplifies the process of determining the outputs from a stage to other stages. No changes in the left to right ordering of images on PIPE will need to be considered. The importance of this will become clear in Section 5. The planarity assumption also makes sense in terms of the interconnection structure of PIPE. Since a stage's output only goes to itself and its two nearest neighbors (ignoring the wildcard busses for the moment), communication across a large number of stages is difficult. The addition of the hypothetical nodes simplifies the computation of the heuristics used to estimate the distance until an alu operation must take place to the left or to the right of a given operation.

4. General Description of the Algorithm

The algorithm works through the DAG in a top down manner. Throughout the scheduling, a frontier list holds all the nodes which are candidates for being scheduled; that is, all unscheduled operations having all their ancestors previously scheduled. Initially, the frontier list contains the children of the input images. For each time unit the algorithm schedules all the operations possible on the stages. It uses heuristics to determine the order of scheduling and to position the outputs from the stages. This output positioning entails choosing from among the fwd, rec and bwd paths from a stage. In the course of scheduling the frontier list gradually moves through the DAG until all operations have been scheduled.

The following is a more complete outline of the algorithm. Most of the details are deferred until Section 6.

1. Initially, one pass is made through the complete DAG to compute several heuristics for each node. These heuristics assist in determining the length of the computation, and in determining output paths for the images output by each operation. The first heuristic determines the minimum number of time units required to complete the operations descending from a node. It is called the *critical path length*, or CPL. There are three heuristics used in determining output paths. One estimates the width of the subgraph rooted at each node (the FAN). The other two estimate the number of time units until descendents of this node have to "merge". That is, the distance to the nearest, leftmost alu descendent (the LTM), and the distance to the nearest, rightmost alu descendent (the RTM). (These alu descendents must involve an arc that is not a descendent of the node.) Note that the terms "alu" and "merge" are used almost interchangeably here. Merges take place in the DAG when two paths meet. The merging of paths takes place only at alu nodes of the DAG.

2. Determine stage positions for the input images. All input images are loaded into the

stage buffers using the wildcard busses. An input image with a large value of the FAN heuristic will have more empty stages between it and its nearest neighboring input images. However, the images are limited in the distance between them by the values of LTM and RTM heuristics. These heuristics ensure that the images are near enough so that their descendents may merge as soon as possible. Similar computations will be made at the end of each time unit to determine whether an output image from a stage will use the bwd, rec or fwd paths.

Once the locations for the input images have been determined, build the initial frontier list containing the operations that are ready to be scheduled and the stage number where each will be computed.

3. Repeat the following steps until the frontier list is empty. Each iteration represents scheduling all the operations that are to take place in one time unit. At the beginning of this loop all the input images to all of the nodes on the frontier list will always be in the buffers of the stages. Thus, the loop first schedules operations that may *follow* the stage buffers. Next, it determines the output paths. Finally, it schedules operations that may occur before the stage buffers in the next time unit.

a. Sort the operations in the frontier list by decreasing order of CPL value. Each candidate operation will have a stage in which it may be scheduled.

b. For each operation in the sorted list: schedule it if it passes two tests. The first test determines if there is an unallocated ALU, NBR or LUT unit on the stage associated with the node. The appropriate functional unit will depend, of course, on the node's type (alu, nbr or lut). The second test checks to see if there is space to output the results of an operation on either the bwd, rec, or fwd path. To determine if there is space, all of the result images are output to the leftmost successor stage possible from each stage. When an operation is successfully scheduled, add its children to the frontier list, and record its location and "path" within the stage.

c. After all the operations that will fit in a stage at the current time have been scheduled, determine the best path for each output image. Use the heuristics described in step 2 above to determine if an output image should go left (bwd path), center (rec path), or right (fwd path). This will ensure that the paths which need to merge move toward each other. Such paths will eventually merge in a pre-buffer ALU of a stage.

d. Schedule the operations for the pre-buffer functional units for the next time unit. Update the frontier list to remove these operations. Note that there may be some operations on the frontier list that were ready in the last time unit, but could not be scheduled because of hardware resource limitations. They will remain in the frontier list to be scheduled at a future time.

5. Important Concepts

Before proceeding to the details of the scheduling algorithm in Section 6, it will be useful to first introduce some concepts that are central to understanding it.

5.1. Planarity

The most important effect of working only with planar graphs is that it eliminates the problem of determining the best ordering of output images from the stages. Initially, due to the placement of the images, the frontier list is in left to right order. Throughout the computation of the schedule, the frontier list will always maintain this ordering. Thus, when an alu operation is ready for scheduling, the two inputs to it are adjacent to each other in the frontier list. Since no two paths cross each other in the DAG, there is no need to reorder the images that are output from the stages. The only remaining problem is determining how many stages should be between images.

5.2. Arcs vs. Nodes

Throughout the above discussion the frontier list has been described as containing a list of nodes. Obviously, the nodes of the DAG represent the candidate operations that need to be scheduled. An image that results from such an operation must be communicated to other stages via the output paths. In general, a single image may need to be output to more than one stage. For example, a node with four children requires the result of its operation to be available for four future operations. Because of this duplication of output images, it is easier to use "arcs" as the unit of reference within the algorithm. Each arc represents an output image from the computation of one node and input to the computation of another. In addition, there is exactly one input arc for lut and nbr nodes in the DAG, and for alu nodes there are two. Thus, the remaining description of the algorithm will involve the scheduling of arcs. Only in choosing the exact location for an operation, will the node be considered. And, of course, the actual operations of

PIPE will involve nodes and the results of nodes.

5.3. Frontier List

The frontier list can now be described precisely. Initially it contains each of the output arcs from the input image(s). Currently, the algorithm assumes that there are at most two input images. When an operation is scheduled, the incoming arc(s) to the operation is (are) removed from the frontier list and the outgoing arcs from the computation replace it. Thus the frontier list is the set of arcs that are ready for scheduling. The frontier list also represents the set of all images currently stored in all of the stages' image buffers. Recall that there are two arcs on the frontier list for each alu node to be scheduled. Because the DAG is planar, they must be adjacent to each other in the frontier list. Both will be removed after the alu node is scheduled on a stage. Finally, observe that the frontier list is a "cut set" of the graph. Thus there are exactly as many possible frontier lists as there are cut sets.

5.4. Pre-Buffer and Post-Buffer Operations

The outline of the algorithm given in Section 4 emphasizes the operations that follow the image buffers on a stage of PIPE. This emphasis is due to the architecture of a stage. Once the contents of the fwd, rec and bwd paths are specified for all stages at a given time unit, the scheduling of operations onto the pre-buffer functional units of all stages at the next time unit are completely determined. This is because the images arriving via these paths must be merged into one (see Figure 2). Following the stage buffer there is much more flexibility in the architecture. The contents of any buffer may enter the NBR computations. The results of the NBR computations and any stage buffer are available for ALU C and TVF LUT. All of these are available for output via the bwd, rec and fwd paths.

6. Algorithm Details

This section gives the details of the algorithm for mapping a DAG onto PIPE. The computations described in the first two subsections, for determining the values of the heuristics and assigning the initial location for the images, are done only once for each DAG. The remaining three subsections describe steps performed iteratively until the frontier list is empty. Each iteration schedules operations on all the stages for a single time unit.

6.1. Heuristics

Each heuristic and a method for computing it are described below. The heuristics are computed for each arc of the graph. Each is an estimate of a property of the graph which is used in the scheduling process.

6.1.1. CPL

The critical path length heuristic estimates the longest path through the graph from each arc to a leaf node. In general, there may be more than one leaf node. The CPL value for an arc is a lower bound on the number of time units required to compute the subgraph rooted at the arc. It is defined to be the maximum of the lengths of all possible paths to a leaf from the arc. The length of a path is determined by assuming that the operations (nodes) on the path will be computed as soon as possible. That is, no delays are incurred due to either a lack of hardware stages or dependencies on other paths in the DAG. For example, a path consisting of lut, alu, alu, lut, nbr and alu nodes, will be counted as a requiring a single time unit (see Figure 2).

All CPL values can be computed in linear time during one bottom-up pass through the DAG. Each leaf node is assumed to fit as late as possible in a stage. The remaining possible operations before the node on the stage are called the STAGE_LIST. For example, after an alu operation the STAGE_LIST will contain (lut alu alu lut nbr). The STAGE_LIST is recorded on the incoming arcs to the leaf nodes. These arcs are given a CPL value of 1.

For an arc, a_0, not leading to a leaf node, the CPL value is determined from the CPL and STAGE_LIST of each of a_0's child arcs. For each child arc compute the path length through it, including the node that a_0 leads to. If a_0's node is contained in the child arc's STAGE_LIST, the CPL value through the child is the same as the CPL value of the child. The STAGE_LIST through this child is the updated STAGE_LIST. If the node does not fit into the child arc's STAGE_LIST, add one to its CPL value and compute a new STAGE_LIST. This new STAGE_LIST is computed in the same manner as for a leaf node. The CPL value for a_0 is the maximum of the CPL's computed through its child arcs. The STAGE_LIST for a_0 is the shortest resulting STAGE_LIST of those having the maximum CPL value.

This computation proceeds until the arcs for the input images are reached. The total time required is $O(A)$, where A is the number of arcs in the DAG.

6.1.2. FAN

The FAN value estimates the width of the widest possible subgraph descending from an arc. To compute the exact value would require examining all possible frontier lists below this arc. Instead, the computation of the FAN value limits its look ahead to the grandchildren of the given arc. The FAN value is computed to be the maximum width of this subgraph. There is a problem with this definition, however. There may be alu nodes in this limited subgraph which have incoming arcs that are not in the subgraph. These alu nodes should not be counted as part of both this subgraph and another subgraph. To account for this, the computation of the maximum width subgraph is done twice. The first computation computes the maximum width of the subgraph with these alu nodes removed. The second computation computes the maximum width of the subgraph with these alu nodes included. The FAN value for this arc is the average of these two values. Assuming that the out-degree of each node is bounded by some constant, the computation of the FAN value requires constant time for each arc. The total computation therefore requires $O(A)$ time.

6.1.3. LTM / RTM

To understand the LTM and RTM values consider two arcs a_l and a_r that are adjacent to each other on a valid frontier list. If the subgraphs of a_l and a_r are independent of each other, then RTM (a_l) and LTM (a_r) will both be -1. The value -1 indicates that they will not merge. However, if the subgraphs are not independent, then a descendent of a_l and a descendent of a_r will "merge" at an alu node. Because of the definition of planarity given in Section 3, the arcs to be merged will be the rightmost path descending from a_l and the leftmost path descending from a_r. The LTM value for a_r is the length of the path from a_r to this alu node. This path is found by taking the leftmost child arc of each arc starting from a_r until the closest alu node is reached. This alu node must have an incoming arc that is a descendent of a_l. The RTM value for a_l is computed similarly, except that the rightmost path is used at each step. Thus, the purpose of the LTM and RTM values for a given node are to measure the number of time units until a merge will take place with arcs to the left and to the right of this arc.

The actual computation of the LTM value is similar to that of the CPL value. There are three main differences. First, the computation only considers the leftmost child arc of each arc. Second, when an arc is the *right* input to an alu operation, its LTM = 0 and its STAGE_LIST is (lut alu). This corresponds to a left merge at the beginning of a stage. The third difference is when the left child arc does not lead to a merge. In this case the LTM value will be -1. As in the CPL computation, $O(A)$ time is required to compute both the LTM and RTM values for all arcs in the graph.

6.2. Mapping the Input Images onto PIPE

Currently, the algorithm assumes that there are at most two distinct input images. This allows the images to be simultaneously moved into the appropriate stage buffers at the beginning of the first time unit. The stages into which they are input are determined by the following computation. There will be as many copies of each image as there are outgoing arcs from that image. Denote these arcs "image arcs". For example, in Figure 3 there is one input image and two image arcs. The LTM, RTM and FAN values of these image arcs guide their initial placement on PIPE.

To compute the placement, initially determine the maximum distance between each pair of adjacent arcs a_i and a_{i+1}. This distance is the minimum of two values: (1) twice the maximum of RTM (a_i) and LTM (a_{i+1}), and (2) the average of FAN (a_i) and FAN (a_{i+1}). The motivation for this step is that arcs should always remain "within reach" of each other while still having enough room to compute all the descendent operations from each arc. If two arcs are within the distance specified above, then the rightmost descendent of a_i and the leftmost descendent of a_{i+1} could be shifted towards each other on PIPE in successive time units, and then "merged" using an ALU without any unnecessary delays. To see this, recall two facts: (1) the RTM and LTM values are lower bounds on the merge distances, and (2) the output of each stage can be shifted either one stage to the left or right, i.e. two paths can move at most two stages closer together at each time unit.

If all the image arcs will fit onto PIPE with the distances between them as determined above, then these distances determine the best placement of the arcs. If the distances are too large, they are all scaled down uniformly. The scale factor is the ratio of the sum of the distances to the total number of PIPE stages. After scaling, some adjustment may still need to take place. In the final adjustment, priority is given to the arcs with the greatest CPL values. The result of this computation specifies the relative placement of image arcs onto the stage buffers of PIPE. This step requires $O(M)$ time, where M is the number of outgoing arcs from the input images.

6.3. Post-Buffer Stage Scheduling

Once the placement of all the input image arcs is determined, the initial frontier list is built and scheduling of post-buffer operations begins. The frontier list is partitioned into a set of sublists, one for each stage. Each element of a stage frontier list contains an arc and where that arc is located on the stage. Initially this location will be one of the stage buffers. Later, when an arc is scheduled, its child arcs replace it on the frontier list. The "location" for these arcs is defined to be the functional unit on the stage where the given arc's associated node is computed (e.g. NBR A or PRE-NBR LUT). To avoid confusion in the subsequent discussion, when we refer to "nodes" we will mean operations in the program graph, and when we refer to "units" we mean the hardware functional units in a stage that perform those operations.

The first step in scheduling an individual time unit is to sort the entire frontier list into decreasing order of CPL value. Arcs with the same CPL value are kept in left to right order. The sorted frontier list is divided into groups of equal CPL value. The groups are scheduled in order using the results of scheduling the higher priority groups. This includes ensuring that there are hardware interconnection paths available out of each stage. At the end of scheduling each priority group, there may be paths with multiple arcs on them (these arcs will have the same parent arc). The scheduler then attempts to split up these arcs so that they use multiple output paths. This will ensure that the highest priority arcs are output to multiple stages so that they do not interfere with each other in subsequent time units. The details of these computations are given in the next two subsections.

6.3.1. Priority Group Scheduling

Within a priority group, arcs are scheduled in left to right order. In scheduling each arc there are two conditions to be verified. The first test is to determine if there is enough room on the stage for the arc's associated operation. The second test determines if there is space to output the results of this operation.

Given an arc to be scheduled, the determination of a legal placement depends on the type of operation the arc leads to. There are many special cases, so only a sketch of the main ideas is given here. If the node is an alu, then the other incoming arc must be available in the same stage. In addition, ALU C in this stage must not have been allocated for another operation.

The scheduling of an lut or nbr node is more complicated. For a nbr node several possibilities must be checked before proceeding. The first check is to make sure that at least one NBR unit on the stage is empty. In addition, the candidate nbr node must be compatible with the contents of the PRE-NBR LUT. If this LUT has not been allocated, there is no problem. In this case the PRE-NBR LUT will be allocated as the identity operation so that the image corresponding to the current arc passes through it unchanged. Otherwise, if the PRE-NBR LUT is already allocated as the identity operation, there is an operation already scheduled for one of the NBR's. In this case, the candidate nbr node must be a sibling of the previously scheduled nbr node. This ensures that the input image of the candidate nbr is the same as that of the nbr node already scheduled, i.e. their incoming arcs are from the same parent node. In this case the candidate nbr node may be scheduled in the other NBR unit. Otherwise, if the PRE-NBR LUT already has a non-identity operation in it, the arc under consideration must be a child arc of the node corresponding to this operation. If any of the above tests fail, then the nbr node cannot be scheduled, and it remains on the frontier list.

For an lut node, the decision is only difficult if both the PRE-NBR LUT and the TVF LUT are unallocated. If only one is unallocated then it is used. If both are already used, the node cannot be scheduled. If neither is used, a choice must be made. The PRE-NBR LUT is chosen if there is a nbr child of the lut. The PRE-NBR LUT is also chosen if there is an alu child of the lut and if the other arc for the alu is on the frontier list of this stage. When the PRE-NBR LUT is allocated and the candidate arc has non-nbr children, the algorithm must be careful to allow these children to pass through a NBR unit on the stage unchanged. To ensure this, one of the NBR's must be allocated as the identity operation.

Before testing to see if there is an output path available, final result images are scheduled to be transferred to the output stage via one of the wildcard busses. If the wildcards are all allocated, then the results must be output via the normal paths and moved to the output stage at a later time.

In this part of the scheduling all outputs are directed to the leftmost possible interconnection path. That is, the bwd path is always used if possible, followed by the rec path, and finally the fwd path. At this point, all sibling output arcs from the same node have only one output path allocated to them. These arcs may be output on the bwd path if (1) they are the leftmost arcs on the stage frontier list, and (2) if there are no other arcs scheduled to be input to the stage immediately to the left. Other arcs could be scheduled to be input to that stage if there is output on the rec path of that stage, or if there is output on the fwd path of the stage to the left of it. (The only time more than one path may have *inputs* to a stage is if all the input arcs on the paths can be merged in ALU A and ALU B of that stage.) The requirement that the output arcs be the leftmost on the stage frontier list ensures that the left to right ordering is maintained. If outgoing arcs cannot use the bwd path, the rec and fwd paths are tried in turn. For each of these paths, the outputs of this stage and the stages to the right may need to be shifted to make room for other outputs. Output arcs may only use the forward path if they are the rightmost in the frontier list.

If an arc on the frontier list can be scheduled according to the preceding tests, the new internal stage computations and the new outputs are recorded. When an arc succeeds in being scheduled, its children arcs replace it on the frontier list. If these arcs can still be scheduled on the same stage as their parent arc, they are added to the list of candidate arcs in the proper position by CPL value.

6.3.2. Spreading Out High Priority Outputs

When results are scheduled for output using the procedure described in the previous section, all adjacent outgoing arcs from the same node are output together on the same path. This can be done since the image resulting from the node computation is really what gets output. It allows as many operations to be scheduled within a CPL group as possible. Once all arcs in a CPL group are scheduled, however, it desirable to take paths having multiple arcs and split the arcs onto a second path. This will allocate more stages to these arcs so they don't interfere with each other in future scheduling. The only paths considered are those having at least one arc whose CPL value is equal to that of the CPL group just scheduled.

After a CPL group is scheduled, the algorithm attempts to split multiple arcs on a path to more than one path. It searches the stages for outputs with multiple arcs where at least one arc has the given CPL value. These outputs are split, if possible, across more than one output path. Arcs that are about to be involved in a merge are given precedence in this process. The algorithm attempts to output each such arc on a separate path. Output arcs will not be split up if extra paths are not available (this includes making sure the inputs to the stages in the next time unit are valid) or if the arcs under consideration are not on the left or right end of a stage's frontier list.

6.3.3. Timing

The time for scheduling a single time unit is as follows. Assume that the maximum width of the frontier list is proportional to the number of PIPE stages, N. The sorting of the frontier list requires $O(N \log N)$ time. Also, for each operation being scheduled, constant time is required to determine if it fits on a stage. $O(N)$ time is required to determine if an output path is available. Since there are $O(N)$ possible arcs on the frontier list, the time for testing all the candidates is $O(N^2)$. Finally, the splitting up of output paths associated with multiple arcs is also achievable in time $O(N^2)$. This requires keeping a list of the output paths with multiple arcs for each CPL value. A path is placed on the list in order by the greatest CPL value of any of the arcs on the path. Thus, the total time for scheduling operations which can be executed in one time unit is $O(N^2)$.

6.4. Output Path Selection

The scheduling of the operations for a single time unit places all outputs as far to the left as possible (described in Section 6.3). The procedure produces a valid schedule for a time unit. It does not try to output images that are contained in buffers. However, in some cases these images buffers may need to be output to shift them onto an empty stage or to shift towards another image on PIPE. It also ignores the positioning of arcs so merges (alu's) may take place and so that there are enough empty stages between arcs to schedule the descendents of these arcs. The placement algorithm given below decides which output paths to use while taking these possibilities into consideration. It uses the FAN, LTM and RTM heuristics to guide its decisions.

6.4.1. Placement Algorithm

The first part of this computation builds an output list for each stage. Included are arcs that are on the left or the right of the frontier list, and any outgoing arcs for operations that follow the stage buffers. The arcs on the left or right ends of the frontier list may be stored in buffers. This output list will contain at most three outputs for each stage. Included with each potential output are (a) the arcs involved, and (b) the output choices (bwd path, rec path, fwd path, or remain in buffer).

Once these output lists are built, repeat the following until all outputs have a single path. (1) Compute the farthest left path and the farthest right path possible for each output. The leftmost and rightmost paths for an output are constrained by the choices made for other outputs. (Recall that the left to right order of the paths is bwd, rec, fwd.) (2) If all outputs have exactly one path then this process is complete. Otherwise, (3) choose the output arc with the greatest CPL value, and (4) pick the output path for this arc (described in detail below).

In choosing the path for an output, the algorithm determines whether it is best to use the bwd, rec or fwd path. If the best path is not available, then the next best path is chosen. For example, the algorithm could determine that it is best to move an output to the right, but the only possibilities for this output may be the bwd path and the rec path. In this case, the next best choice, the rec path, is allocated instead.

The path for an output is determined by first looking for merges that may take place at the beginning of the next time unit. If a merge can take place, then the output path is chosen to allow this to take place. This includes the possibility of three arcs merging in the beginning of a time unit. In this case the arcs must be in adjacent stages and the three way merge will take place in the middle stage. If a merge can not take place then the best path for an output is chosen based on the LTM, RTM and FAN values of the output arc(s) and those of their left and right neighbors on the frontier list. In choosing the direction, the algorithm first ensures that the output arc(s) is (are) near enough to each other on PIPE to be able to merge without delay. If they are, it chooses the path based on the values of FAN for the arcs in question. This process proceeds similarly to the procedure for locating input images described in Section 6.2. If either the left or the right neighbors on the frontier list are too far away, i.e. if merging will be delayed, then the path chosen is in the direction of the delay. If both are too far away, then the direction chosen is toward the merge that will take place earlier. In practice, merges are only delayed in rare circumstances that are discussed below.

6.4.2. Timing

To allocate stage output paths for all output arcs in a single time unit requires $O(N^2)$ time. To see this, first note that computing the farthest path to the left or the farthest path to the right for the outputs requires $O(N)$ time. Also, the search for the highest CPL value among the outputs with undecided paths requires $O(N)$ time. Finally, finding the left and right neighbor of an output in the frontier list requires constant time, and the choice of the location requires constant time. Thus, choosing the path for each output requires $O(N)$ time. Since there are at most N locations to choose, the total time is $O(N^2)$.

6.5. Pre-Buffer Stage Scheduling

As mentioned above, the pre-buffer operations on a given stage are completely determined from the contents of its input paths. In scheduling these operations, inputs corresponding to multiple arcs cannot be modified. Otherwise, any LUT and ALU units are scheduled if possible. Once the output of ALU B is determined, the frontier list for the stage is rebuilt. This list will consist of the arcs resulting from ALU B (ALU B could be the identity operation or it may be empty) and the arcs corresponding to images currently stored in the stage buffers. The arcs in the frontier list are maintained in left to right order as they occur in the program DAG.

This completes the description of the scheduling algorithm. It iterates through the above steps until the frontier list is empty. The algorithm produces a description of where and when operations are to be performed on PIPE, and how output images are moved between stages. Stage buffers are marked as free whenever the corresponding arcs are scheduled or when the arc is output from the stage.

7. Examples

Several examples are given in this section. Two of them show schedules for complete DAG's. Another shows how the heuristics work together to maintain an optimal placement of images in stages. Finally, an example is given where the algorithm performs poorly.

7.1. Example 1

The solution generated by our scheduling algorithm for the program DAG in Figure 4 is given here as an example. Figure 5 shows the DAG with nodes and arcs labeled to help simplify the description. Nodes are labeled beginning with the letter *n*. Arcs are labeled beginning with the letter *a*. Table 1 shows the values of the heuristics for each arc. Values of -1 for LTM or RTM indicate that no merge occurs. Figure 6 shows the results of the scheduling. In this figure, the node labels have been replaced by a triple indicating, in order, the time, stage and functional unit where that operation takes place. Also, the output paths used for shifting *a19* are shown.

The input image is scheduled to be input to stages 3 and 6. In the first time unit only nodes *n2* and *n3* may be scheduled. Note that the outputs from *n2* are split up. Its child arcs go to stages 2, 3 and 4. In the second time unit, nodes *n5, n6, n7* and *n8* are scheduled. At the beginning of the third time unit, arcs *a8* and *a9* are merged in stage 2, and arcs *a12* and *a13* are merged in stage 5. Finally, notice how arc *a19* is shifted so that it arrives at stage 3 for the final operation associated with node *n22*.

The algorithm produces a schedule requiring seven time units. Using the wildcard bus only for initial input and final output (the same way the algorithm uses the wildcard busses), the best we have been able to do by hand is six time units. Note that although the CPL value for the entire DAG is four, PIPE cannot be scheduled to achieve this time. Doing so would require that *n11* and *n17* be done in time unit 3. This could only be done if arcs *a16* and *a10* were input together at the beginning of the same stage. Since these arcs cannot be combined before the stage

Arc	CPL	FAN	LTM	RTM
a0	4	4	-1	2
a1	4	2	2	-1
a2	3	2	-1	1
a3	3	2	1	2
a4	2	1	1	1
a5	3	1	1	1
a6	3	1	1	-1
a7	2	1	-1	1
a8	2	2	1	0
a9	2	2	0	1
a10	2	1	1	1
a11	2	1	0	0
a12	2	1	1	0
a13	2	1	0	-1
a14	1	1	-1	0
a15	1	1	0	0
a16	2	1	1	0
a17	2	1	0	0
a18	2	1	0	2
a19	1	1	0	-1
a20	1	1	0	0
a21	2	1	1	0
a22	2	1	0	2
a23	1	1	-1	0
a24	1	1	0	0
a25	1	1	0	1
a26	1	1	0	1
a27	1	1	-1	1
a28	1	1	-1	0

Table 1. Heuristic Values for Arcs in Figure 5.

buffers, the four time unit schedule is not possible.

7.2. Example 2

Figures 7 and 8 show another example program DAG along with the schedule that the algorithm generated. The schedule requires seven time units. The best we have been able to do by hand is six.

7.3. Other Examples

One of the main ways that delays in scheduling can occur is when arcs are too far apart to merge immediately. Usually, this will not happen because the algorithm uses the LTM and RTM heuristics to keep arcs close enough so that there is no delay when their descendents need to merge. However, even when two arcs are within the distances required by their LTM and RTM values, their descendents may not be able to merge without incurring delays. Consider, for example, Figure 9. For arcs *a1*, *a2* and *a3* all the significant LTM and RTM values are equal to 2. Hence, it is legal for *a1* to be in stage 1, *a2* to be in stage 5 and *a3* to be in stage 8. However, in this case the paths will not be able to merge in two time units. This is because the successors of *a2* will not be able to shift both to the left and the right. But, in this case, arcs *a1*, *a2*, and *a3* would never have gotten to stages 1, 5 and 8, respectively. The FAN value of each of these arcs will be 1, so the algorithm will keep these three arcs in adjacent stages, instead of allowing them to move farther apart. This example shows how the heuristics complement one another to maintain an optimal number of stages between arcs.

Figure 10 shows an example where the algorithm performs poorly. In the schedule generated, the arcs with the highest CPL values are scheduled almost sequentially for a number of time units. Assume for this example that the width of PIPE is six stages. The descendents of *a0* will be allocated to the leftmost five stages by the beginning of the third time unit. The subgraph descending from *a1*, containing the arcs with the highest CPL values, initially is width 1 and therefore requires only one stage. Later, however, it branches out. When this first occurs only one stage will be allocated to the subgraph, so the operations in it must be scheduled sequentially. After some time, the right subgraph will receive the number of stages that it needs, but for the intervening period it is nearly sequentialized.

This problem has two causes. First, the FAN heuristic has a limited amount of look-ahead in estimating the width of the subgraph below an arc. In this example, the value of FAN for *a1* does not accurately reflect the size of its subgraph. This problem with the FAN heuristic is commonly referred to as the "horizon effect". The second cause is due to the "greedy" nature of the scheduling algorithm. Because of this, preference is given to scheduling as many nodes in one time unit as possible. (Some low priority nodes may be blocked when arcs are split across multiple paths as described in Section 6.3.2.) In this example, there are initially many descendents of *a0* on the frontier list, and only one descendent of *a1*. Hence, the algorithm allocates all but one stage to the descendents of *a1*. Once the frontier list changes to include a number of descendents of *a1*, the algorithm can only gradually shift the descendents of *a0* to the left and allocate more stages to the descendents of *a1*.

8. Analysis

As shown in Section 6, assuming the maximum width of the frontier list is $O(N)$, where N is the number of stages, then the worst case time for computing the mapping for a single time unit is $O(N^2)$. The total time is therefore the length of the PIPE program generated times $O(N^2)$. More generally, if the maximum width of the frontier list is M, $M > N$, then the worst case time for computing a single time unit is $O(M^2)$.

Two other areas of analysis, correctness of algorithm and the execution time of the PIPE program generated by the algorithm, are covered in the next two subsections.

8.1. Correctness

In this section we show that a valid PIPE program is generated by the algorithm. It assumes that a legal, planar DAG has been input. The first part of the proof establishes that a legal frontier list will always be maintained. The second part shows that there is a bounded amount of time before some arc is removed from the frontier list. Thus, since there are no loops in the graph and since the graph is finite, the algorithm will eventually terminate. The third part, completing the proof, shows that all nodes are eventually scheduled.

A legal frontier list is in left to right order and is a cut set of the graph. Initially, the frontier list contains the input image arcs in left to right order. This initial list is obviously a cut set. The frontier list is changed only when a node is scheduled. This change represents the replacement of one or two arcs (in the case of an alu node) by an ordered list of its (their) child arcs. If the frontier list was in left to right order before the arc(s) was (were) replaced, the new frontier list will also be in left to right order since the graph is planar. Furthermore, assuming that the previous frontier list was a cut set, the new one will be also. This can be seen by assuming that it is not a cut set. Then there must be a path from an input image to a leaf node with no arcs from this path on in the frontier list. Since the previous frontier list was a cut set, exactly one of the arcs on the path must have been on the previous frontier list. This arc must be one of the ones that was scheduled, otherwise the new frontier list would automatically be a cut set. Since the path must include one of the arcs that was replaced, it must also include one of the child arcs. But these arcs are in the new frontier list. Hence no such path exists. Thus, the new frontier list is a cut set, contradicting our assumption that it was not one.

The second part of the proof shows that there is a finite amount time before some arc is removed from the frontier list. Consider all the arcs in a frontier list that have the highest CPL value. There are two cases to consider. First, if any of these arc leads to a lut or nbr node, then at least one of them will be scheduled in the next time unit. The second case occurs if all arcs having the highest CPL value lead to alu nodes. In this case there must be a pair of arcs that lead to the same alu node. These arcs will be adjacent to each other in the frontier list, but they may be several stages apart. However, the farthest apart they may be is the width of PIPE, N. In this case, the arcs will move toward each other shifting one stage per time unit. In time N/2 they will meet and the alu node will be scheduled. This constant of N/2 represents the bound on the delay between changes to the frontier list.

The final part of the proof shows that all nodes are scheduled. Assume to the contrary that at least one node is never scheduled. Let *n* be a node such that *n* is never scheduled, but all of its ancestors are. There must be at least one such *n*. Since *n*'s ancestors are scheduled, the arc(s) leading into *n* eventually are placed on the frontier list. But, if these arcs are placed on the frontier list, *n* must be scheduled since the algorithm is guaranteed to terminate. This contradicts the assumption that there is such an *n*, establishing that all nodes are eventually scheduled. This completes the proof of the algorithm's correctness.

8.2. Time Bounds for the PIPE Program Generated

Time bounds for the execution time of the PIPE program generated by our scheduling algorithm are given in this section. The bounds are given first in terms of a DAG whose width is $O(N)$, and then they are given in the general case. Clearly, the maximum CPL value for all image arcs in a given program DAG is a lower bound on the number of time units required to execute the DAG on PIPE. In the first example in Section 7 the best possible mapping required two more time units than indicated by the maximum CPL value. For a general DAG, there are an exponential number of possible frontier lists. The problem of finding the schedule which minimizes the execution time on PIPE is a generalization of a problem that was shown to be NP-complete in [7], hence our problem is also NP-complete.

In measuring how far the generated program differs from the optimal program, it is best to consider when and how long an arc having the highest CPL value in the frontier list may be delayed. First recall that there are at most $O(N)$ arcs in the frontier list. There are two ways that an arc with the maximum CPL value can be delayed. First, it may be blocked from scheduling because other arcs with the same CPL value are chosen ahead of it. This delay is $O(N)$ time units. Second, an arc leading to an alu may be delayed because it is too far from the other incoming arc to the alu. (Note that this other incoming arc has the same CPL value.) Fortunately, these two $O(N)$ delays are not multiplicative. Only arcs leading to alu nodes are delayed in waiting for merges. If an arc leading to an alu node is delayed waiting for a merge, it will not be delayed once it is near enough to the other incoming arc to the alu node. This implies that the longest an arc with maximum CPL value may be delayed is $O(N)$.

Finally, there can be at most six arcs with the same CPL value on one path. Thus, once a CPL value becomes the maximum in the frontier list, there are at most $O(N)$ time units before the all arcs having this CPL value are scheduled. Thus, the worst case time for the program generated is $O(N)$ times the maximum CPL value in the DAG. In the general case, if the maximum width of the frontier list is M, $M > N$, the worst case is $O(M)$ times the maximum CPL value.

9. Enhancements

Several possible enhancements to the scheduling algorithm are discussed in this section. These include modifying the program DAG to create equivalent, but more easily scheduled, DAG's, and generalizations for handling non-planar graphs.

9.1. DAG Modification

Several features of the DAG can affect the performance of the algorithm. The most obvious is with alu operations that merge three images into one. The order in which the merges are specified may cause a delay in the PIPE program generated. For example, if the operation to merge the three images is commutative then it doesn't matter which pair of images is combined first. However, if the DAG specifies that the left and center images are to be combined first, then all three images cannot be combined in a single stage using the pre-buffer ALU's. If the DAG specifies that the center and right images are to be combined first, then the three images can be combined before the stage buffer.

A more subtle problem is demonstrated by the example in Figure 11. Figure 11a shows a DAG with an initial nbr node having six output arcs. The problem with it is that the output arcs must be distributed among the three output paths. These three paths input the arcs to three consecutive stages. Unfortunately, the six subsequent operations are likely to interfere with each other in the next time unit due to hardware limitations in the stages. The DAG could be modified, however, to contain six redundant nodes instead of the initial node with six outputs. In this case the initial operation would be scheduled concurrently in six separate stages, and the next set of operations could be scheduled without interference in the next time unit. The augmented DAG with the redundant nodes defines an equivalent program, but yields a more efficient PIPE schedule. This DAG is shown in Figure 11b.

Unfortunately, it is not always clear when altering the DAG will produce a more efficient PIPE program. An algorithm for deciding when and how to alter the DAG could assist the scheduler presented in this paper to produce faster PIPE programs.

9.2. Non-Planar Graphs

Generalization of the algorithm to non-planar graphs is the most important outstanding issue. The first approach to this problem is to preprocess a given DAG to create an equivalent planar DAG. For example, this was done by hand to the graph in Figure 12 to create the graph in Figure 3. The graph was altered by adding redundant nodes to replace locations where arcs in the graph crossed. Ideally, the process of altering the graph can be automated. This would produce planar graphs from non-planar graphs as a preprocessing step to our algorithm.

For some graphs, however, altering them may be impractical or even impossible to do automatically. In these cases it will still be necessary to handle non-planar graphs. One solution is to make sure that the frontier list is in left to right order at the beginning of each time unit. The stage computations would be scheduled as before, and then the outputs from the stages would be inserted in a new frontier list in left to right order. To do this arcs may need to cross in the frontier list. If two arcs need to cross each other locally, they may do so using the normal output paths. To do this they must be in the same stage or in adjacent stages. If they are in adjacent stages, one arc can go out the forward path of the left stage, and the other can go out the backward path of the right stage. For non-local crossing arcs, the wildcard busses could be used to move an arc into the correct position. Since there are a limited number of wildcard busses, however, all arcs mat not be able to be reordered correctly in a single time unit. Lower priority arcs must be delayed until there a bus available. In this case the left to right ordering of the frontier list would be temporarily violated. The arcs which are not in their correct relative positions must be ignored in the stage scheduling of the next time unit.

This generalization presumes that there would be some method of determining which arcs to shift or cross and when to do so. How to do this is not immediately clear. Also, the generalization would change the meaning of the RTM and LTM heuristics. New heuristics would need to be designed to capture some the characteristics of non-planar graphs.

References

1. S. R. Sternberg, Parallel architectures for image processing, *Proc. 3rd Int. Computer Software and Applications Conf.*, Chicago, IL, 1979, 712-717.
2. E. W. Kent, M. O. Shneier, and R. Lumia, PIPE — pipelined image-processing engine, *Journal of Parallel and Distributed Computing* **2**, 1985, 50-78.
3. T. Gross, H. T. Kung, M. Lam, and J. Webb, Warp as a machine for low level vision, *Proc. Int. Conf. on Robotics and Automation*, St. Louis, MO, 1985, 790-800.
4. S. S. Wilson, The Pixie-5000 — a systolic array processor, *Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Miami Beach, FL, 1985, 477-483.
5. Vicom Systems Inc., Vicom-VMV 1000 product announcement, 1986.
6. E. G. Coffman, *Computer and Job-Shop Scheduling*, Wiley, New York, 1976.
7. C. E. McDowell and W. F. Appelbe, Processor scheduling for linearly connected parallel processors, *IEEE Trans. Computers* **35**, 1986, 632-638.
8. Digital/Analog Design Associates, *Programming the PIPE*, 1985.
9. J. Canny, A variational approach to edge detection, *Proc. National Conf. on Artificial Intelligence*, Washington, DC, 1983, 54-58.

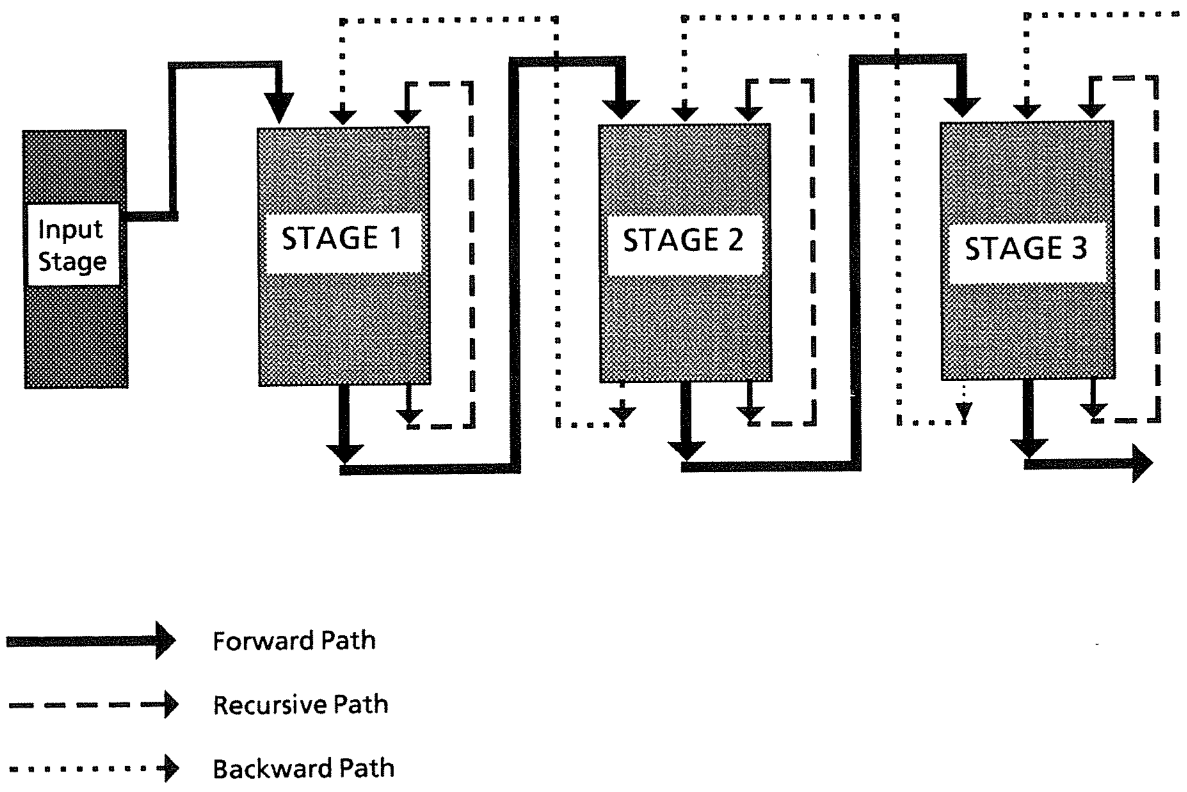


Figure 1. Data communication paths between stages in PIPE.

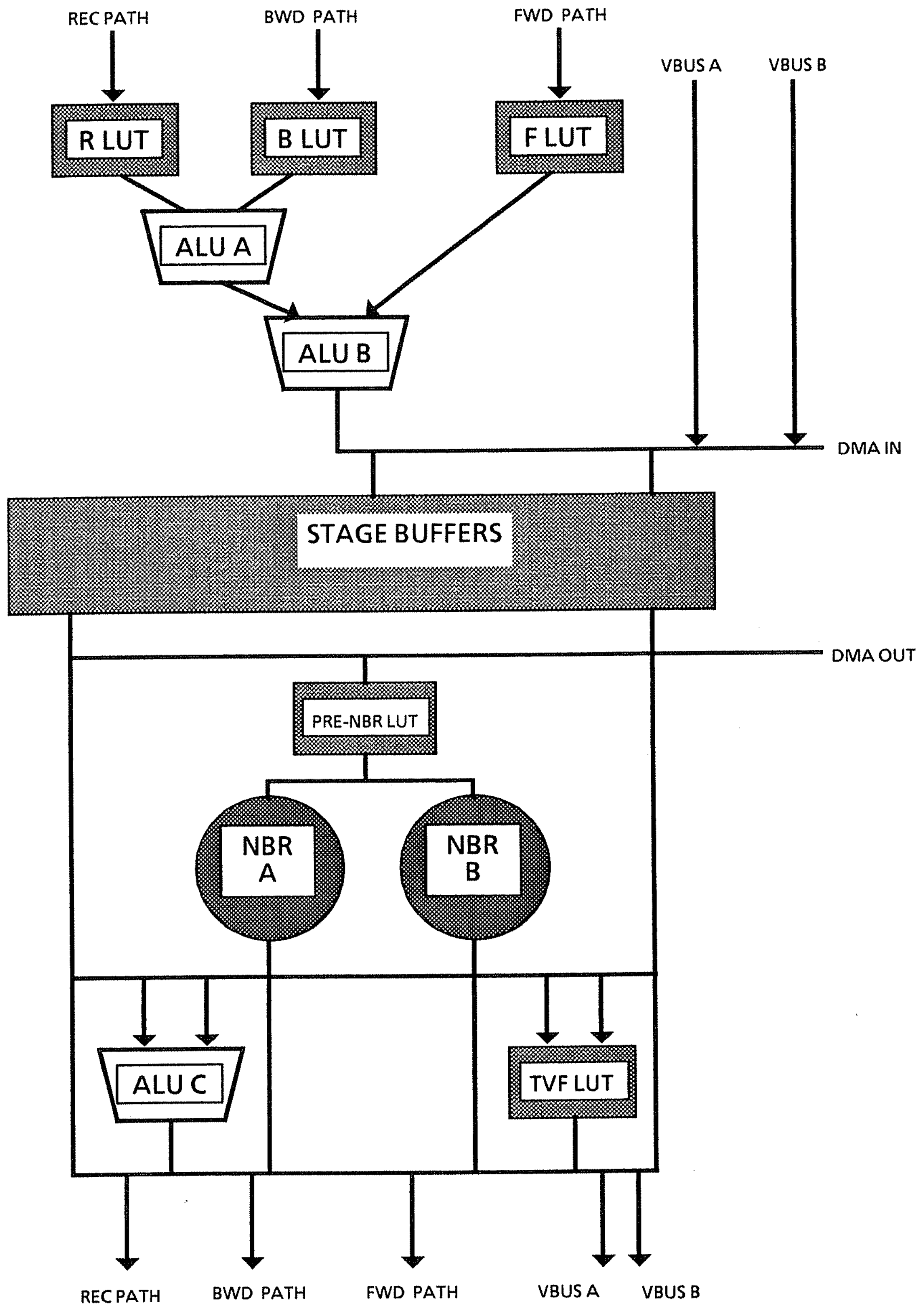


Figure 2. Organization of the buffers and functional units in a stage.

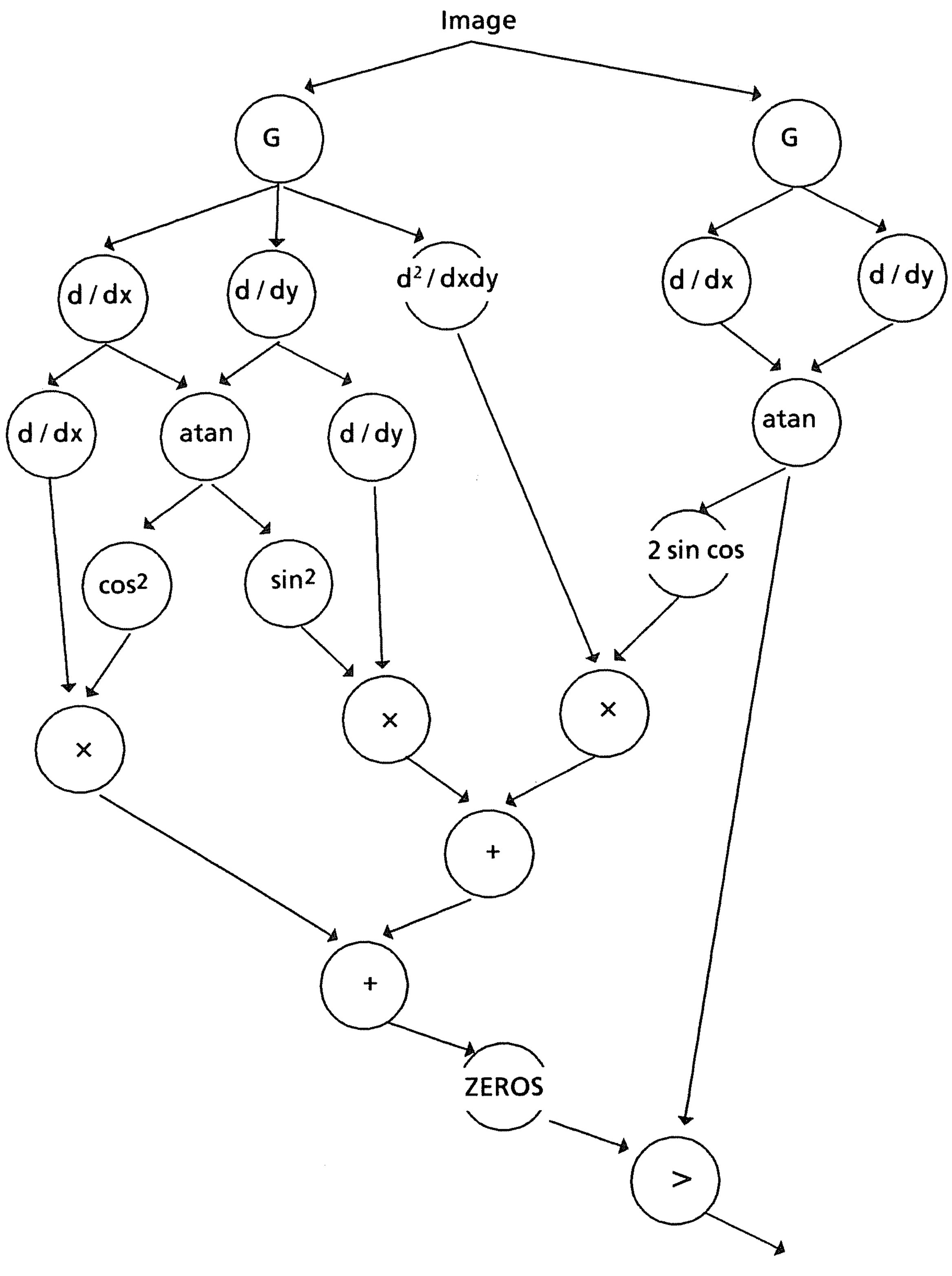


Figure 3. Canny edge detection algorithm.

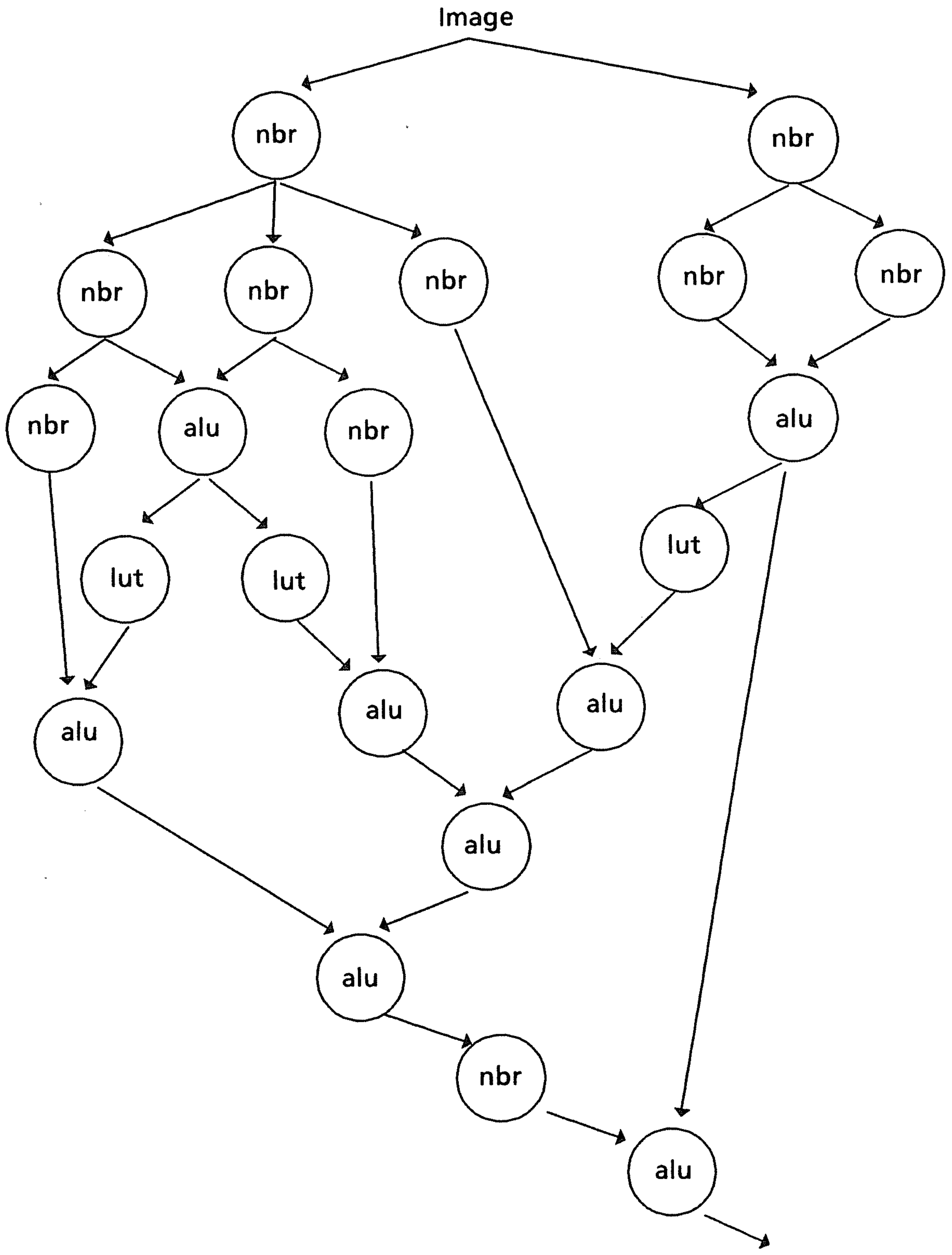


Figure 4. Canny algorithm with generic operations.

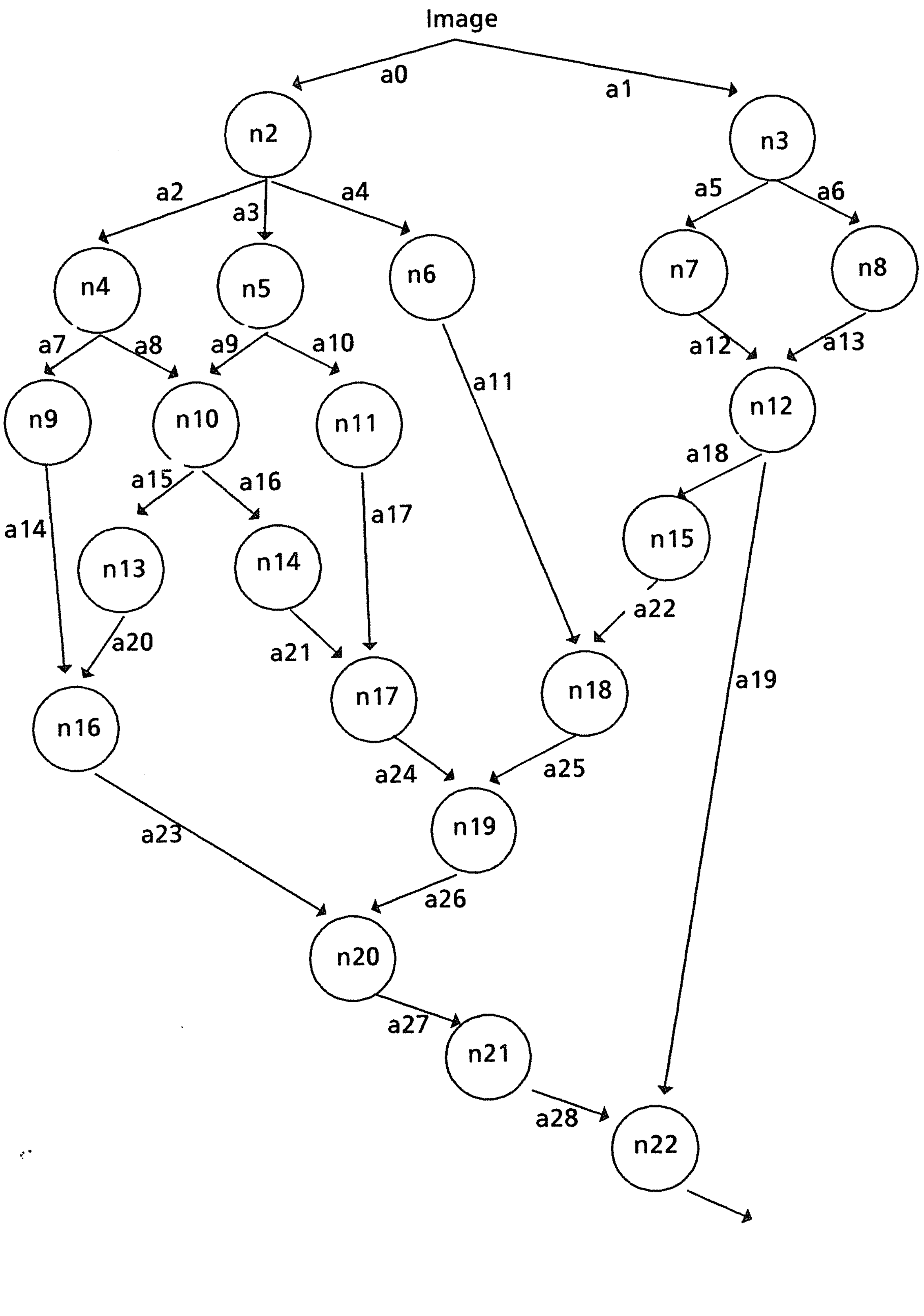


Figure 5. Canny DAG with node and arc labels.

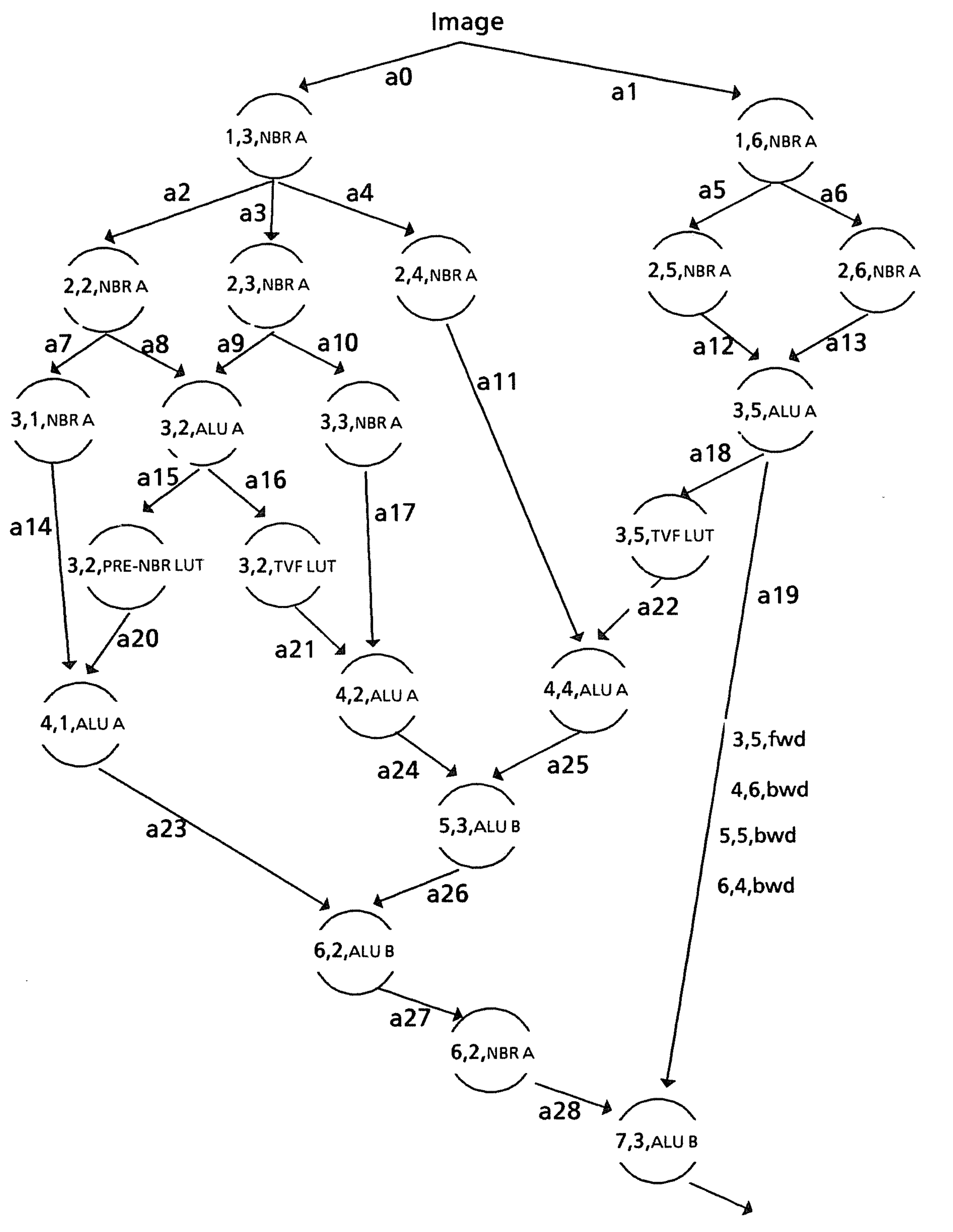


Figure 6. Canny DAG with scheduled operations.

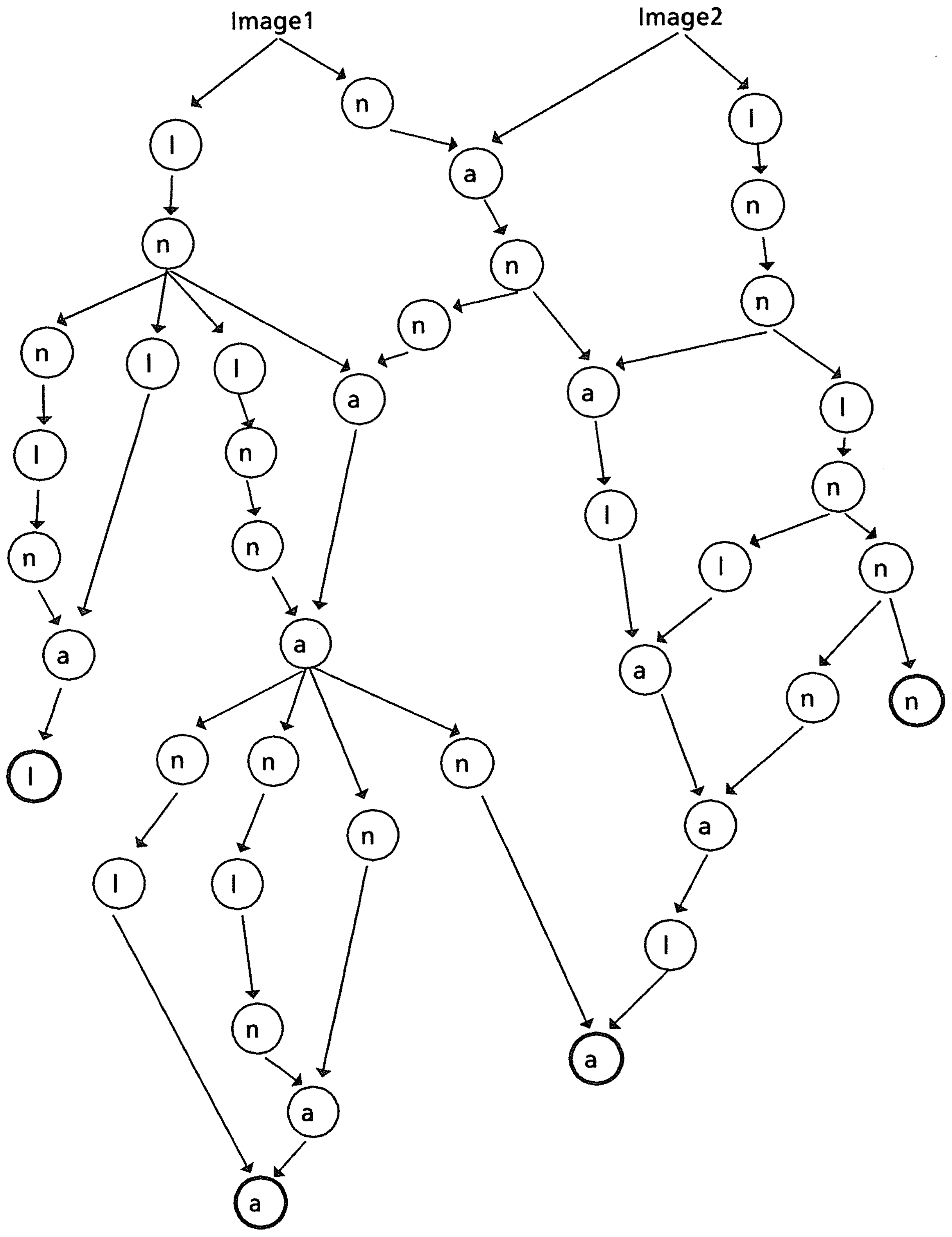


Figure 7. Second Example DAG. Bold nodes are results, nbr nodes are marked "n", lut nodes are marked "l", and alu nodes are marked "a".

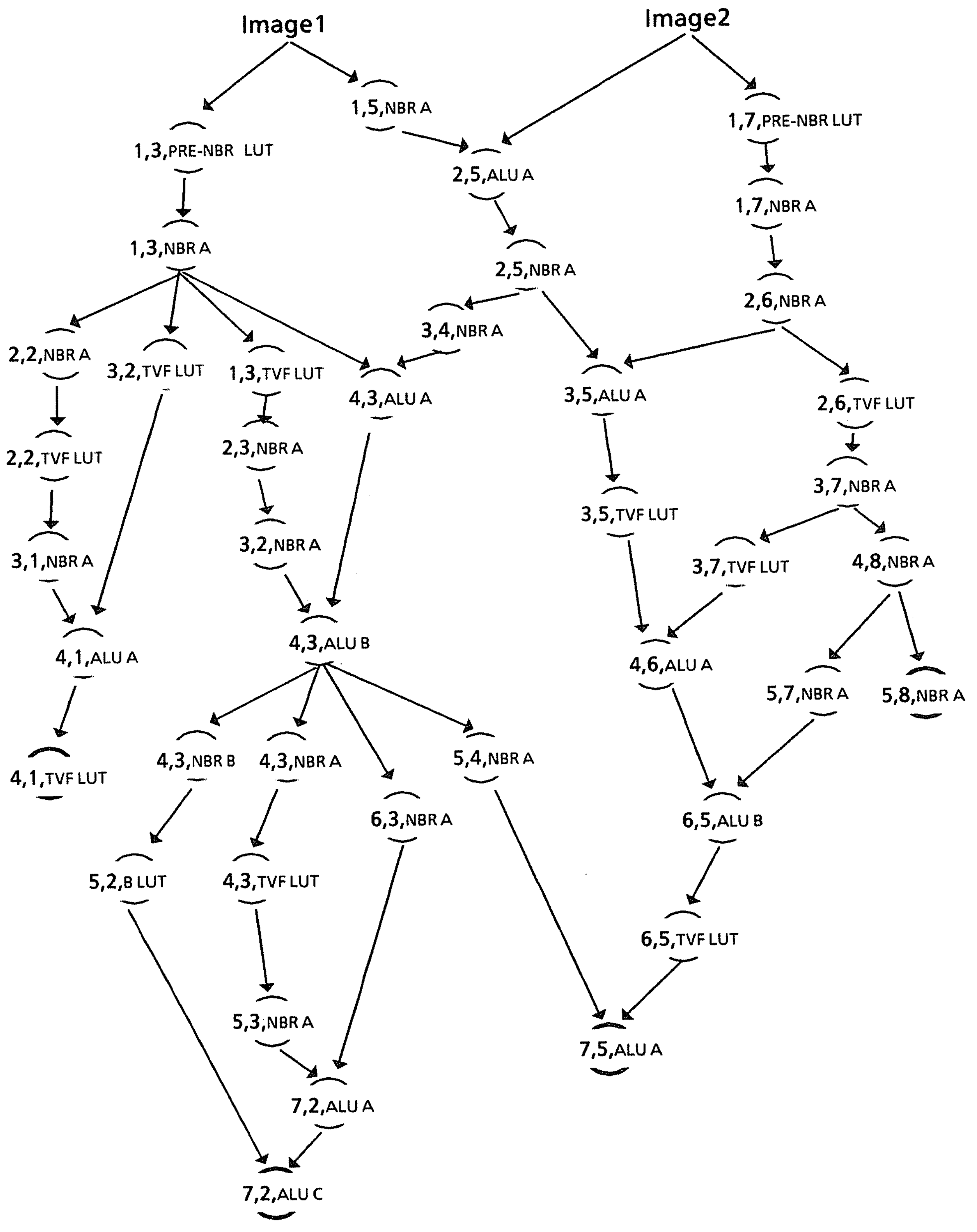


Figure 8. Results of scheduling the DAG in Figure 7.

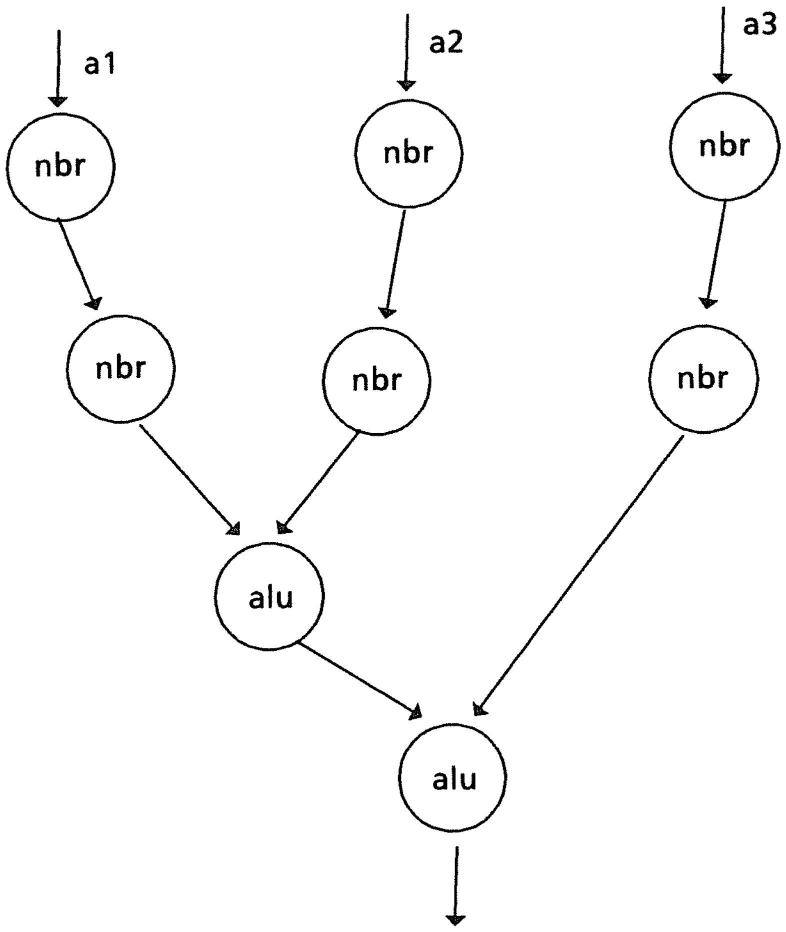


Figure 9. Merging paths in a DAG.

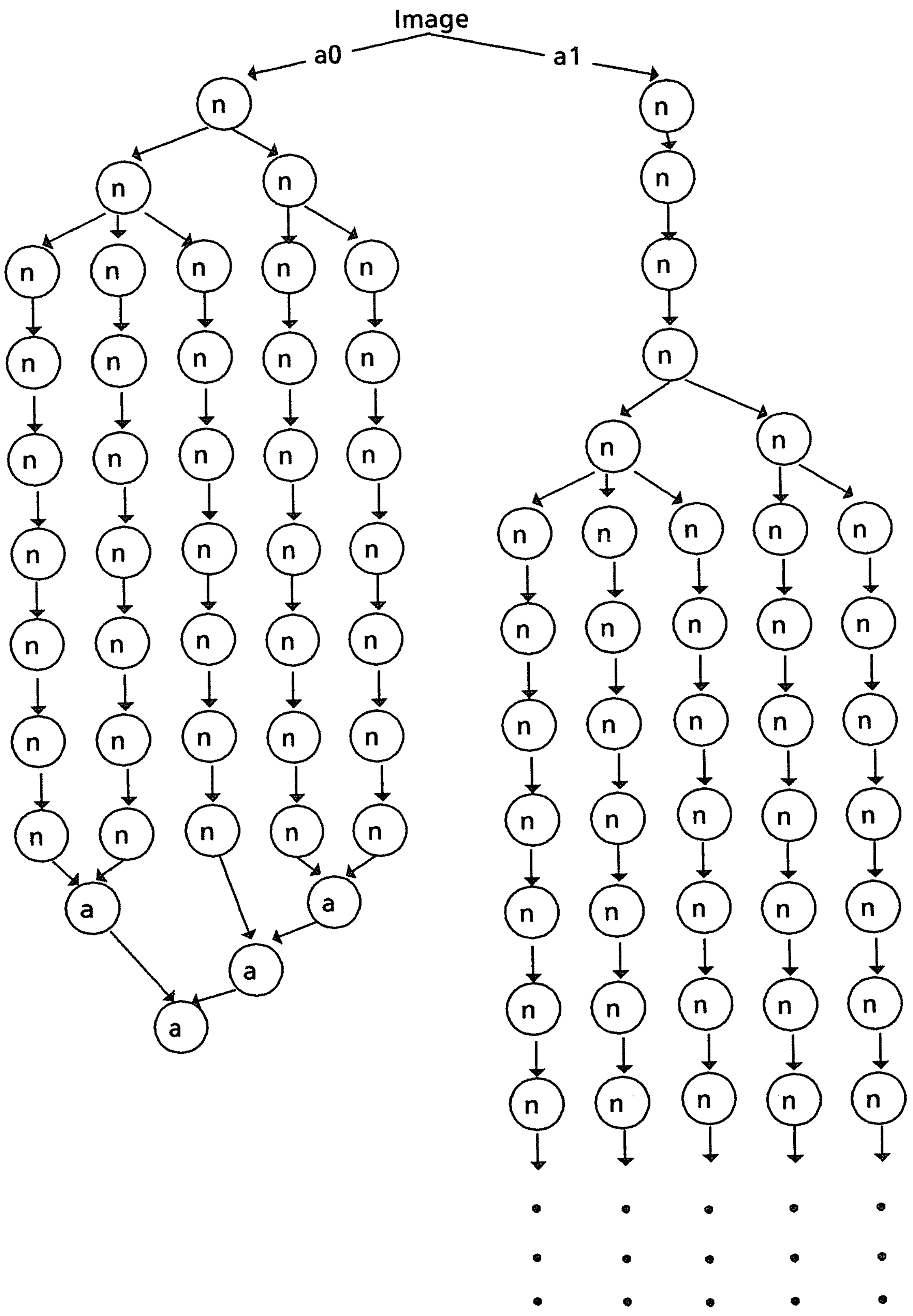
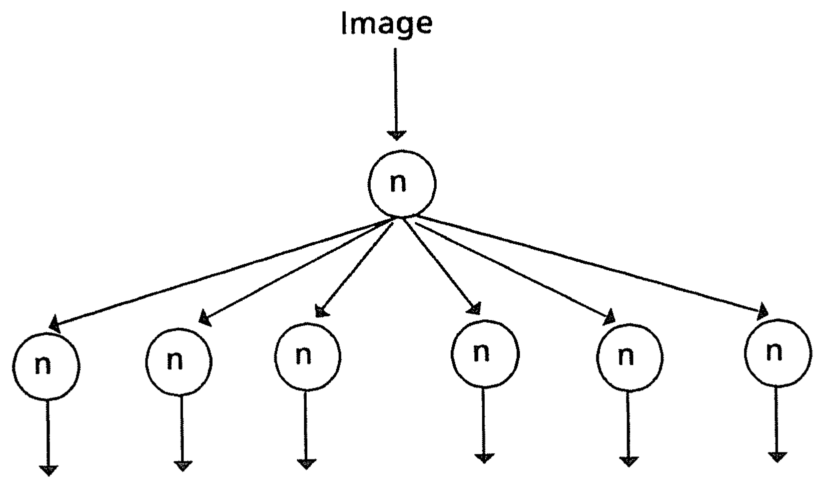
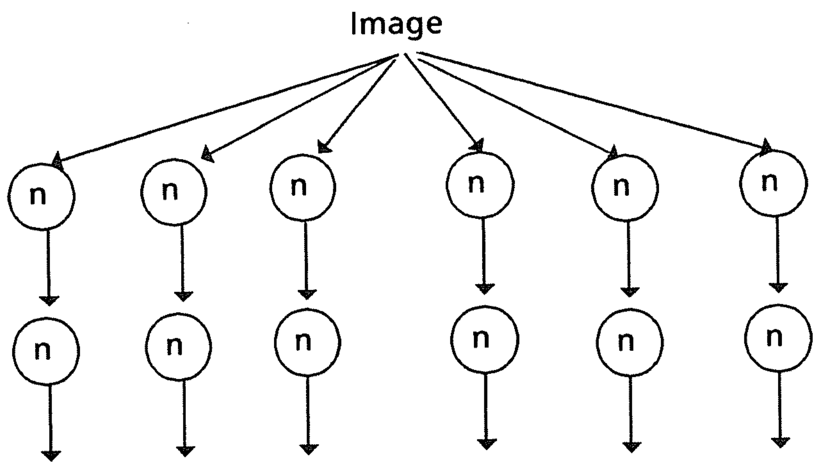


Figure 10. Near worst case DAG.



(a)



(b)

Figure 11. (a) Initial operation with wide fan out.
(b) Initial operation split redundantly. This DAG generates a faster PIPE program.

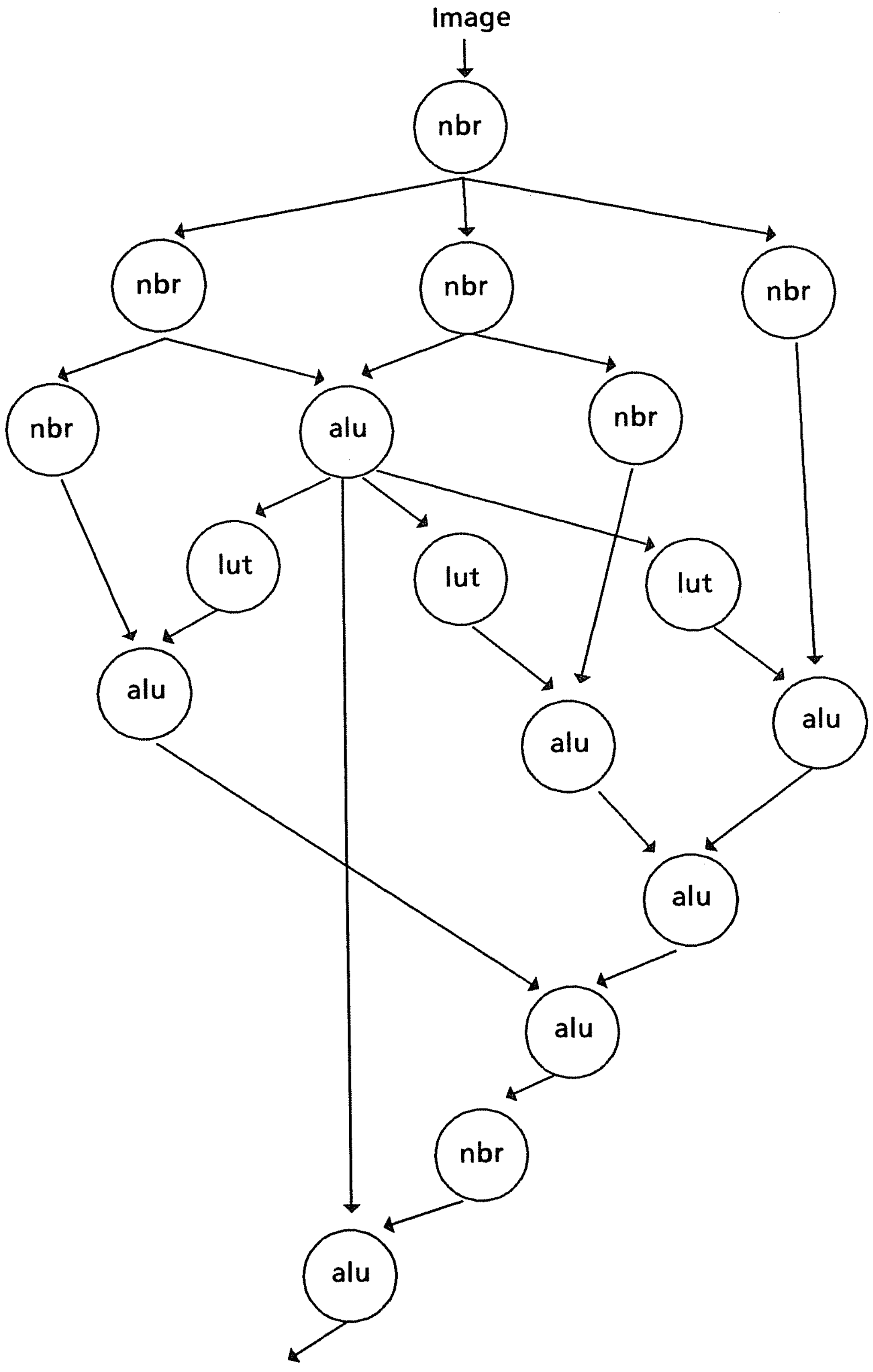


Figure 12. Non-Planar DAG version of the Canny algorithm.