(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 February 2006 (23.02.2006)

PCT

(10) International Publication Number
WO 2006/020001 A2

(51) **International Patent Classification:**
G06T 1/00 (2006.01) ***G06F 17/00*** (2006.01)

(21) **International Application Number:** PCT/US2005/025134

(22) **International Filing Date:** 15 July 2005 (15.07.2005)

(25) **Filing Language:** English

(26) **Publication Language:** English

(30) **Priority Data:**
60/588,538 15 July 2004 (15.07.2004) US

(71) **Applicants** *(for all designated States except US)*: **THE REGENTS OF THE UNIVERSITY OF CALIFORNIA** [US/US]; 1111 Franklin Street, Oakland, CA 94607-5200 (US). **PIXAR** [US/US]; 1200 Park Avenue, Emeryville, CA 94608 (US).

(72) **Inventors; and**
(75) **Inventors/Applicants** *(for US only)*: **OWENS, John, Douglas** [US/US]; 1491 Cedar St., Berkeley, CA 94702 (US). **RIFFEL, Andy** [US/US]; 2043 Picasso Ave., Davis, CA 95616 (US). **LEFOHN, Aaron** [US/US]; 1502 Jackson Street, #402, Oakland, CA 94612 (US). **VIDIMCE, Kiril** [MK/US]; 2060 Sutter St., Apt. #406, San Francisco, CA94115 (US). **LEONE, Mark** [US/US]; 2304 Mallard Drive, Walnut Creek, CA_94597 (US).

(74) **Agents: MAJERUS, Laura, A.** et al.; Fenwick & West LLP, 801 California Street, Mountain View, CA 94041 (US).

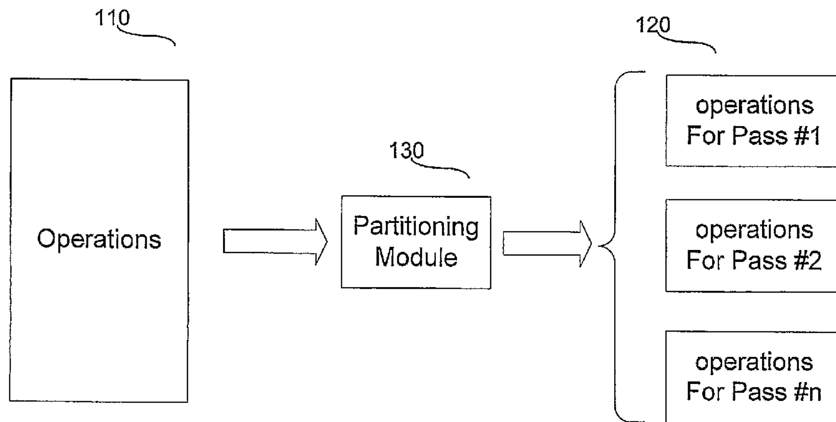
(81) **Designated States** *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) **Title:** FAST MULTI-PASS PARTITIONING VIA PRIORITY BASED SCHEDULING



(57) **Abstract:** The described embodiments of the present invention include a method and system for partitioning and partitioning operations. The operations are first prioritized, then partitioned into one or more partitions.

WO 2006/020001 A2

FAST MULTI-PASS PARTITIONING VIA PRIORITY BASED SCHEDULING

INVENTORS
John Owens, Andy Riffel, Aaron Lefohn, Kiril Vidimce, and Mark Leone

[0001] This application claims priority under 35 U.S.C. § 119(e) to U.S. Provisional Application No. 60/588,538 of Owens et al., filed July 15, 2004, which is herein incorporated by reference.

BACKGROUND OF THE INVENTION

[0002] Recent advances in architecture and programming interfaces have added substantial programability to graphics piplined systems. These new features allow graphics programmers to write user-specified programs that run on each vertex and each fragment that passes through the graphics pipeline. Based on these vertex programs and fragment programs, people have developed shading languages that are used to create real-time programmable shading systems that run on modern graphics hardware.

[0003] The ideal interface for these shading languages is one that allows its users to write arbitrary programs for each vertex and each fragment. Unfortunately, the underlying graphics hardware has significant restrictions that make such a task difficult. For example, the fragment and vertex shaders in modern graphics processors have restrictions on the length of programs, on the number of resource constraints (i.e., temporary registers) that can be accessed in such programs, and on the control flow constructs that may be used.

[0004] Each new generation of graphics hardware has raised these limits. The rapid increase in possible program size, coupled with parallel advances in the capability and flexibility of vertex and fragment instruction sets, has led to corresponding advances in the complexity and quality of programmable shaders. For many users, the limits specified by the latest standards already exceed their needs. However, at least two major classes of users require substantially more resources for their application of interest.

[0005] The first class of users are those who require shaders with more complexity than the current hardware can support. Many shaders in use in the fields of photorealistic rendering or film production, for instance, exceed the capabilities of current graphics hardware by at least an order of magnitude. The popular RenderMan shading language, for example, is often used to specify these shaders, and RenderMan shaders of tens or even hundreds of thousands of instructions are not uncommon. Implementing these complex RenderMan shaders is not possible in a single vertex or fragment program.

[0006] The second class of users use graphics hardware to implement general-purpose (often scientific) programs. This "GPGPU" (general-purpose on graphics processing units) community targets the programmable features of the graphics hardware in their applications, using the inherent parallelism of the graphics processor to achieve superior performance in microprocessor-based solutions. Like complex RenderMan shaders, GPGPU programs often have substantially larger programs that can be implemented in a single vertex or fragment program. They may also have more complex outputs. For example, instead of a single color, they may need to output a compound

data type.

[0007] To implement larger shaders than the hardware allows, programmers have turned to multipass methods in which the shader is divided into multiple smaller shaders, each of which respects the hardware's resource constraints. These smaller shaders are then mapped to multiple passes through the graphics pipeline. Each pass outputs results that are saved for use in future passes.

[0008] A key step in this process is the efficient partitioning of the program into several smaller programs. For example, a shader program may be partitioned into several smaller shader programs. Conventional programs often use the RDS (Recursive Dominator Split) method. This method has two major deficiencies. First, shader compilation in modern systems is performed dynamically at the time the shader is run. Consequently, graphics vendors require algorithms that run as quickly as possible. Given n instructions, the runtime of RDS scales as $O(N^3)$. (Even a specialized, heuristic version of RDS, RDS_h scales as $O(N^2)$.) This high runtime cost makes conventional methods such as RDS undesirable for implementation in run-time compilers. Second, many conventional partitioning systems assume a hardware target that can output at most one value per shader per pass. Modern graphics hardware generally allows multiple outputs per pass.

[0009] There is a need for a partitioning method and system that operates as quickly as possible. There is also a need for a partitioning method and system that allows the output of more than one value from the resulting partitions.

Summary of the Invention

The described embodiments of the present invention include a method and system for partitioning operations. In a preferred embodiment of the present invention, the operations are first prioritized, then placed into one or more partitions. Each of the partitions can then be executed during a plurality of passes.

Brief Description of the Drawings

[0010] The teachings of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings. Like reference numerals are used for like elements in the accompanying drawings.

[0011] Fig. 1 is a block diagram showing operations to be partitioned.

[0012] Fig. 2 is a flow chart showing a method for partitioning operations.

[0013] Fig. 3(a) is a directed acyclic graph in which the nodes are assigned priorities in accordance with a first priority scheme.

[0014] Figs. 3(b)-3(d) show details of additional priority schemes.

[0015] Figs. 3(e) and 3(f) show example of different partitions of the same graph.

[0016] Fig. 4 is a flow chart showing details of a scheduling method that can be used to partition in the method of Fig. 1 in accordance with an embodiment of the present invention.

[0017] Fig. 5 is an example of a ready list using the priority scheme of Fig. 3.

[0018] Fig. 6 is an example of constraints stored in a memory that are specific to particular hardware.

[0019] Fig. 7 is a flow chart showing details of a scheduling method that can be used to partition in the method of Fig. 1 in accordance with an embodiment of the present invention.

[0020] The figures depict embodiments of the present invention for purposes

of illustration only. One skilled in the art will readily recognize from the following discussion that alternative embodiments of the structures and methods illustrated herein may be employed without departing from the principles of the invention described herein.

Detailed Description of Embodiments

[0021] Fig. 1 is a block diagram showing operations 110 to be partitioned. In a described embodiment of the present invention a partitioning module 130 partitions a plurality of operations 110 into a plurality of smaller programs 120 for execution by a processor (or by a plurality of processors (not shown)). Partitioning module 130 preferably contains instructions that can be executed in a data processing system to perform the partitioning operations of the described embodiments of the present invention. The instructions of module 130 are stored, for example, in a memory or appropriate storage media, as are the operations 110 to be partitioned. Partitioning module 130 can be embodied in hardware, software, or firmware. The processor(s) (not shown) can be embodied in, for example, a single data processing system, a general purpose data processing chip, a graphics processing unit, a distributed data processing system, or a networked data processing system. For example, partitioning module 130 may partition a software shader program into smaller programs that are executed by multiple passes through a graphics pipeline.

[0022] Fig. 2 is a flow chart showing a method for partitioning operations. The method is performed, for example, by partitioning module 130 of Fig. 1. As will be understood by persons of ordinary skill in the art, the method can be embodied in instructions stored on a computer readable medium such as a memory, disk, hard drive, CDROM, or a transmission media such as signals on a network connection or signals on a wireless network connection. Element 250 receives operations to be partitioned such as operations 110 of Fig. 1. Element 252 constructs a graph, such as a DAG (Directed

Acyclic Graph) based on the operations using a method known to persons of ordinary skill in the art. The DAG reflects a relationship and dependency between the operations.

[0023] Element 254 determines a priority of the operations of the graph. The determined priority is used to decide an order of traversal of the graph during the partitioning process. The present invention may be used with several priority methods, some of which are described below in connection with Figs. 3(a)-3(d). These priority methods are sometimes called "scheduling methods" herein although they do not actually schedule the operations. Instead, they determine an order in which the nodes of the graph are visited during the partitioning process. Element 256 places the operations into one or more partitions. Each of these partitions may be thought of as one of the smaller programs 120 of Fig. 1. As is described below in more detail, operations are partitioned in accordance with their resource usage and with hard and soft resource constraints of the hardware upon which they will later be executed.

[0024] Fig. 3(a) is a directed acyclic graph (DAG) in which each node corresponds to an operation. This graph represents a data structure or similar construct in memory created by partitioning module 130. In Fig 3(a), the nodes are assigned priorities in accordance with a first priority scheme that employs Sethi-Ullman numbering. Sethi-Ullman numbering is described in, for example, "R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," J. Assoc. Computing Machinery, pp. 715-728, ACM, 1970," which is incorporated by reference herein. Sethi-Ullman numbers are further described in Section 9.10 ("Optimal Ordering for Trees") of "Compilers: Principles, Techniques, and Tools", Alfred V. Aho, Ravi Sethi, and Jeffrey

D. Ullman (Addison-Wesley, 1988), which is incorporated by reference herein.

[0025] In general, this priority scheme orders a graph or tree of operations based on resource usage. A register is an example of a resource and Sethi-Ullman numbers are just one example of a resource estimate. The crux of a priority scheme based on Sethi-Ullman numbers is that performing resource-intensive calculations first frees up resources for later calculations.

[0026] Here, the resource usage of an operation and its children is used to calculate a Sethi-Ullman number for its node. The method labels each node in the graph with a Sethi-Ullman Number (SUN) that indicates a number of registers required to execute the operations in the subtree rooted at that node. In general, for tree-structured inputs, partitioning higher-numbered nodes first minimizes overall register usage. In the example, a node above another node is considered to be a child of that node. Thus, for example, in Fig. 3(a), node #1 is a child of node #3. In the figure, a node above another node is considered to be a predecessor of that node. Thus, for example, in Fig. 3(a), node #1 is a predecessor of node #3. Similarly. Node #3 is a successor of node #1.

[0027] The simple case of Sehti-Ullman numbering involves a node "N" whose children are labeled L1 and L2. Node N represents an operation that requires one register to hold its result. The label of N is determined by:

if (L1 = L2)

then label(N) = L1 + 1

else label(N) = max(L1, L2)

[0028] This method assumes that each operation stores a result in a register.

When both children require M registers, we need a register to hold the result of one child while M registers also are used for the other child, so the total required is M + 1. That extra register is not needed if the children have different register requirements, as long as the child with the bigger resource requirement is run first.

[0029] In the more general case where there are K children:

let N1, N2, ... Nk be the children of N ordered by their labels,

so that label(N1) >= label(N2) >= ... >= label(Nk);

label(N) = max (from i = 1 through K) of label(Ni) + i - 1;

[0030] In the example of Fig. 3(a), the nodes are all assumed to require one output register. Thus, node #3 (as labeled inside the node) has a Sethi-Ullman Number (SUN) of 2 because both of its children have SUNs of 1 (1+1 = 2). Similarly, nodes #4 and #5 have SUNs of 2 since their child (node #3) has a SUN of 2. Similarly, node #11 has a SUN of 3 since its children (nodes #3 and #10) have a SUN of 2 (2+1=3).

[0031] As shown in Fig. 3(a), the SUN values assigned to the nodes result in the following in-order traversal of nodes: 1, 2, 3, 8, 9, 10, 11, 4, 5, 6, 7, 12, 13, 14. The "pre-order traversal" is as follows: 14, 12, 11, 3, 1, 2, 10, 8, 9, 4, 7, 6, 5, 13. The two traversal orders are equivalent traversals: the in-order traversal specifies which operations are scheduled first, whereas the pre-order traversal specifies the order in which the nodes are visited by the scheduling algorithm (i.e. the operations are not scheduled on the way towards the leaves, they are scheduled on the way back to the root).

[0032] In a preferred embodiment, SUNs are assigned to the graph in a first stage and a traversal order is determined during a second stage. The first stage is order

O(n) with the number of input nodes. In the second stage to determine traversal order the method preferably uses a depth-first traversal through the graph, preferably choosing the node with a higher Sethi-Ullman number. Ties are broken in a deterministic and consistent manner. (For example, ties can be broken user a comparison of pointers in the node) This stage is also order O(n) with the number of input nodes.

[0033] Figs. 3(b)-3(d) show details of additional priority schemes. In general, the priority schemes described in this document prefer depth first traversal (i.e., depth first traversal and the ready list method described herein) over breadth first traversal. This preference tends to minimize register usage. Figs. 3(e) and 3(f) shows an example of two possible ways to partition example graphs. A first graph of Fig. 3(e) tries to maximize parallelism in the operations by placing nodes #1, #2, #3, and #4 in the same partition. This approach is often used in conventional methods and results in four pieces of information (from nodes #1, #2, #3 and #4) that need to be passed to a second partition having nodes #5, #6, and #7 therein.

[0034] In contrast, the described embodiments of the present invention tend to minimize register usage. Thus, the graph in Fig. 3(f) partitions nodes #1, #2, and #5 together and partitions nodes #3, #4, #6, and #7 together. This results in only one piece of information that needs to be passed between the two partitions.

[0035] Fig. 3(b) shows an example of a More Predecessors method. In this method, node #3 330 would be given a higher priority than node #5 332 because node #3 has more predecessors (nodes #1 and #2 vs. none).

[0036] Fig. 3(c) shows an example of a More Ready Successors method. In

this method, node #3 340 would be given a higher priority than node #5 342 because node #3 has more ready successors (nodes #4 and #11 vs. just node #11).

[0037] Fig. 3(d) shows an example of a Critical Path priority scheme. Here, all nodes on path 350 have a high priority since path 350 is a longest path and should be prioritized first. Edge weights, representing latencies between operations, can be assigned to each edge, and then used to determine the path length between input and output nodes. Therefore the longest path is not necessarily the path with the most operations.

[0038] Another alternate priority method keeps track of register usage. Specifically, the method keeps track of which operations incur additional register usage (generate) and which operations reduce register usage (kill). Given a choice, operations that kill registers are preferred over registers that generate registers. Note that since Sethi-Ullman numbering accounts for register usage, this priority method is redundant when using SUN.

[0039] Various embodiments of the present invention, uses one or more of the above described priority determining methods. As an example, a preferred embodiment uses a combination as follows: The highest priority nodes are those that reduce register usage, followed by those that leave register usage constant, and finally those that increase register usage. This is the highest priority metric because it most directly affects a number of live registers. The second highest priority metric is to partition operations that will create more ready successors rather than fewer ready successors. The third priority metric is to partition nodes with more predecessors over fewer predecessors and the final priority metric is to partition nodes closest to the critical path.

[0040] Fig. 4 is a flow chart showing details of the method of Fig. 1 in accordance with an embodiment of the present invention. Specifically, Fig. 4 shows a method of partitioning nodes in partitioning module of Fig. 1 using a scheduling algorithm. The method can be used to partition any list of operations that must be partitioned because of resource constraints. Fig. 5 is an example of a ready list data structure 500 stored in an appropriate memory and using the priority scheme of Fig. 3. The following discussion provides an example of partitioning the nodes of the graph of Fig. 3(a) using a ready list scheduling algorithm to determine tree traversal order.

[0041] Elements 402 and 404 correspond to element 254 of Fig. 2, which determines a traversal order. Element 406 adds child nodes to a "ready list" 500. In this example, initially child nodes #1, #2, #8, and #9 are added to the ready list. The remainder of elements form a loop that is executed until all nodes are scheduled into a partition.

[0042] Element 420 chooses a node having a highest priority from the ready list. If the node does not violate any constraints (element 421) the node is added to the current partition 502 and removed from the ready list 500 (element 422). In the example, node #1 is removed from the ready list and placed in the partition 502. (Removal from the ready list is indicated by placing an "x" through the node number in the Figure). A rollback stack in memory is also cleared at this time. If the node violates only soft constraints (such as output constraints) (element 428), the node is scheduled in the current partition anyway and removed from the ready list (element 426). The node is added to the rollback stack. If the node violates an input constraint (element 432) the

node is removed from the ready list without scheduling it in this stage (element 430). If the node violates neither input nor output constraints (element 432) then an operation count constraint or a temporary register count constraint (i.e., a hard constraint) has been violated and the ready list is cleared (element 434). This causes a rollback in element 408.

[0043] In the example, a hard constraint is violated when the number of operations exceeds 8 at time 531. At this time, the partition is rolled back (elements 410, 412, 414) to a time 536, which, in the example, was the most recent time that all hard and soft constraints were met. In the example, at this time, only nodes #1, #2, and #3 are in the partition 502.

[0044] Element 424 is executed after a node is schedule in either element 422 or 426. Element 424 adds new ready operations to the ready list and execution continues with element 408. In the example, when node #1 is removed from the ready list and added to the partition, its parent nodes #3 is not added to the list. Node #3 becomes ready and is added when its other child node #2 is added to the partition. In other words, a node preferably is added to the ready list when all of its children have been added to the partition

[0045] In the example, the number of outputs 506 is a soft constraint and the number of operations in the partition is a hard constraint. These are used for the purpose of example only. In general, soft constraints are metrics that can potentially rise or fall with the addition of more operations to the partition. In contrast, hard constraints can only rise with more operations. A critical resource is a resource that has reached its

constraint in the current partition. When a soft constraint is violated, there is a possibility that it will not remain in a state of violation in the future, while a hard constraint will continue to be violated. Both constraints must be met at the close of a partition. Other embodiments can use additional or other hard and soft constraints 510 and 512. Examples of hard constraints include, but are not limited to, a number of operations currently in a partition (as in the example) and a number of temporary registers used. Examples of soft constraints include, but are not limited to, a number of textures (stored in global memory), whether a varying input is used, uniforms, a number of constants, and a number of outputs (as in the example). The method allows the usage of operations that temporarily overuse constraints such as the number of outputs with the hope that future operations will return the schedule to compliance.

[0046] In one embodiment, nodes that do not use a critical resource are assigned a higher priority "on the fly."

[0047] Fig. 6 is an example of constraints stored in a memory that are specific to particular hardware. These values are evaluated each time a node is added to the partition. Exactly which types of values are hard constraints and which are soft constraints will vary with the type of hardware on which the partitioned operations will be executed. Thus, the constraints used and their designation as hard or soft will vary depending at least on the target hardware.

[0048] Sethi-Ullman numbers are just one example of a resource estimate that can be used as part of a priority scheme. Multipass partitioning can use other types of priority schemes. For example, the number of texture units can be used as a criteria

instead of a number of output registers. In general, these resource estimates can be combined (for example, using a weighted sum) to direct the depth-first partitioner toward the most resource-intensive operations.

[0049] Partitioning also can be performed with a depth-first traversal of the DAG. Directed depth-first scheduling is a solution to the multi-pass partitioning problem (MPP) that relies on a pre-pass to compute resource usage information followed by a depth-first traversal that is guided by those resource estimates. A method using directed depth-first scheduling is described below and shown in 7.

[0050] The depth-first traversal is performed as follows

[0051] - The traversal starts at the root (output) of the operation dependency tree or DAG (element 702). In Fig. 3(a), the root node is node #14.

- At each step, the child requiring the greatest number of resources is visited (element 704).

- If there are no children, or all the children have been visited, and the current operation can be scheduled without violating any constraints, the current operation is added to the current partition (element 706). The operations are then partitioned traversing the DAG in in-order traversal, using the pre-order traversal determined by the depth first method. One implementation uses a recursive algorithm to implement this method.

- The current partition can be finalized as soon as an operation is encountered that violates a constraint. The next partition can then start with the current operation (which is guaranteed to be ready because its children have already been scheduled) (element 708).

[0052] - Alternatively, the traversal can skip operations that violate constraints and continue to consider other operations (element 710). This might be desirable if other operations might be scheduled because of differing resource constraints. For example, resources like texture units might be exhausted before other resources.

[0053] Multipass partitioning also can use other kinds of resource estimates instead of register usage. For example, the number of texture units required to execute a partition could be used. In general, these resource estimates can be combined (for example, using a weighted sum) to direct the depth-first scheduler toward the most resource-intensive calculations.

[0054] Although the present invention has been described above with respect to several embodiments, various modifications can be made within the scope of the present invention. Accordingly, the disclosure of the present invention is intended to be illustrative, but not limiting, of the scope of the invention, which is set forth in the following claims.

WHAT IS CLAIMED IS:

1. A method of partitioning operations, comprising:

determining respective priorities for a plurality of operations in a ready list containing operations to be partitioned;

choosing, from the ready list, an operation that has a highest priority;

adding the chosen operation to a current partition unless a hard constraint is violated; and

adding successor operations that no longer have predecessor operations to the ready list.

2. The method of claim 1, wherein the priority for an operation is determined by assigning a priority based on a number of registers required to execute a subtree of the operation.

3. The method of claim 1, wherein the priority for an operation is determined by assigning a priority based on a Sethi-Ullman number of the operation.

4. The method of claim 1, where the method is performed for graphics operations usable in a graphics processing unit (GPU).

5. The method of claim 1, further including:

allowing partitioning of a operation that temporarily overuses a number of outputs per partition.

4

1 6. The method of claim 1, further including:

2 if a hard constraint is violated, performing a rollback to a point where all

3 hard constraints are met; and

4 adding at least one of the rolled back operations to a new ready list.

5

1 7. The method of claim 1, further including keeping track of resources and

2 registers used by each operation.

3

1 8. The method of claim 9, wherein resources include at least one of the

2 following: slots in a graphics shader operation memory used by each operation, a

3 number of constant and varying inputs, number of textures accessed, number of

4 internal registers used, and number of allowed outputs per pass.

5

1 9. The method of claim 1, where determining respective priorities attempts to

2 minimize a number of passes in the partitioned operations.

3

1 10. The method of claim 1, further including saving multiple intermediate results

2 instead of recomputing them between passes.

3

1 11. The method of claim 1, further comprising:

2 determining that a partition has been completed when hard constraints are

3 violated.

4

1 12. The method of claim 1, wherein partitioning further comprises:
2 differentiating between hard and soft resource limits, hard resource limits
3 being limits that, once reached, make it impossible to partition more operations;
4 and soft limits being limits that, once reached, may possibly allow more
5 operations to be partitioned.
6

1 13. A method of partitioning operations, comprising:
2 determining respective priorities, for a plurality of operations to be
3 partitioned, in a data structure store in memory that represents dependencies between the
4 operations, the priorities assigned in accordance with the operation's register usage,
5 whether the operation creates more ready successors, and a number of predecessors to the
6 graphics operation;
7 choosing, a graphics operation that has a highest priority;
8 adding the chosen graphics operation to a current partition unless a hard
9 constraint is violated; and
10 partitioning successor graphics operations that no longer have predecessor
11 graphics operations.
12

1 14. The method of claim 13, wherein the priority for a operation is determined by
2 assigning a highest priority to operations that reduce register usage, assigning a next
3 highest priority to operations that create more ready successors than fewer ready
4 successors, assigning a next highest priority to operations with more predecessors than

5 fewer predecessors and assigning a next highest priority to operations closest to a critical
6 path.
7

1 15. The method of claim 13, further including:
2 allowing partitioning of an operation that temporarily overuses a number
3 of outputs per pass.
4

1 16. The method of claim 13, further including:
2 performing a rollback to a point where all hard constraints are met; and
3 scheduling at least one of the rolled back operations in a next stage.
4

1 17. The method of claim 13, where the method is performed for graphics
2 operations usable in a graphics processing unit (GPU).
3

1 18. The method of claim 13, further including keeping track of resources and
2 registers used by each partitioned operation.
3

1 19. The method of claim 18, wherein resources include at least one of the
2 following: slots in graphic shader operation memory used by each partitioned
3 graphics operation, a number of constant and varying inputs, number of textures
4 accessed, number of internal registers used, and number of outputs.
5

1 20. The method of claim 13, where determining respective priorities attempts to
2 maximize a number of operations per pass.
3

1 21. The method of claim 13, further including saving multiple intermediate results
2 instead of recomputing them between passes.
3

1 22. The method of claim 13, further comprising:
2 determining that a partition has been completed when hard constraints are
3 violated.
4

1 23. The method of claim 13, further including use of a depth-first method of
2 traversing the operations.
3

1 24. The method of claim 13, further including use of a Sethi-Ullman based
2 method of traversing the operations.
3

1 25. A method, performed by a data processing system, of partitioning a plurality
2 of operations, represented by an operation dependency graph in a memory,
3 comprising:
4 for an operation visited during the traversal, visiting its child operation
5 that requires a greatest number of resources;

6 if an operation has no children, or all the children have been visited, and
7 the current operation can be scheduled without violating any constraints, adding
8 the operation to the current partition ;
9 finalizing the current partition when an operation is encountered that
10 violates a constraint; and
11 starting a next partition with the with the operation that violated the
12 constraint .
13
1 26. The method of claim 25 further comprising:
2 during traversal of the operations dependency graph in the memory,
3 skipping operations that violate predetermined constraints while continuing to consider
4 other operations.
5
6

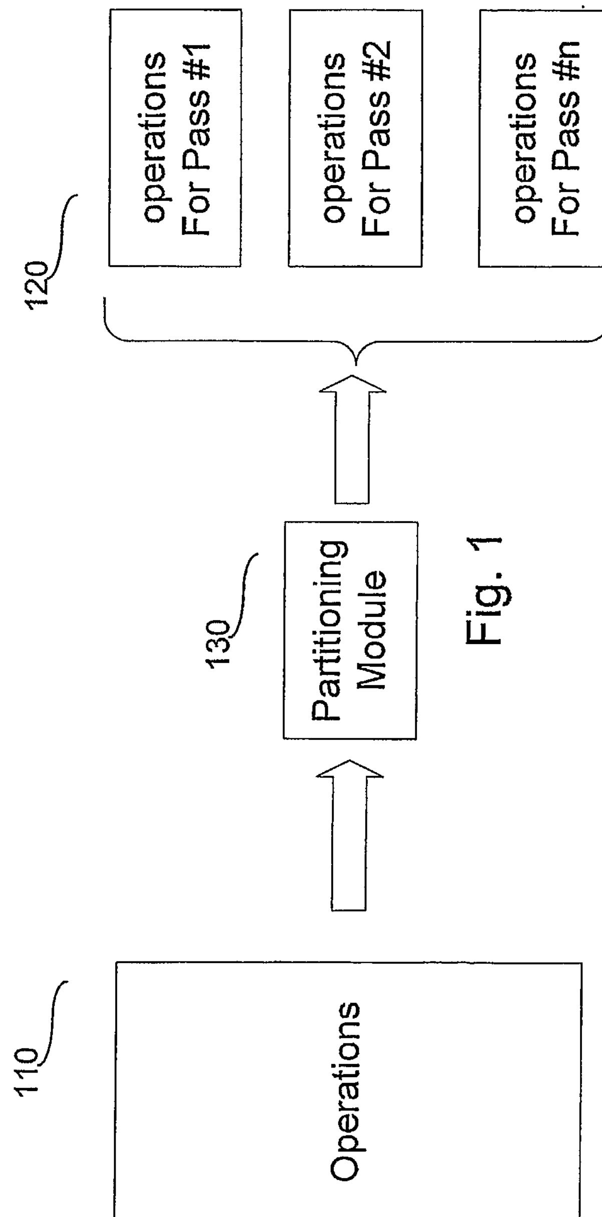


Fig. 1

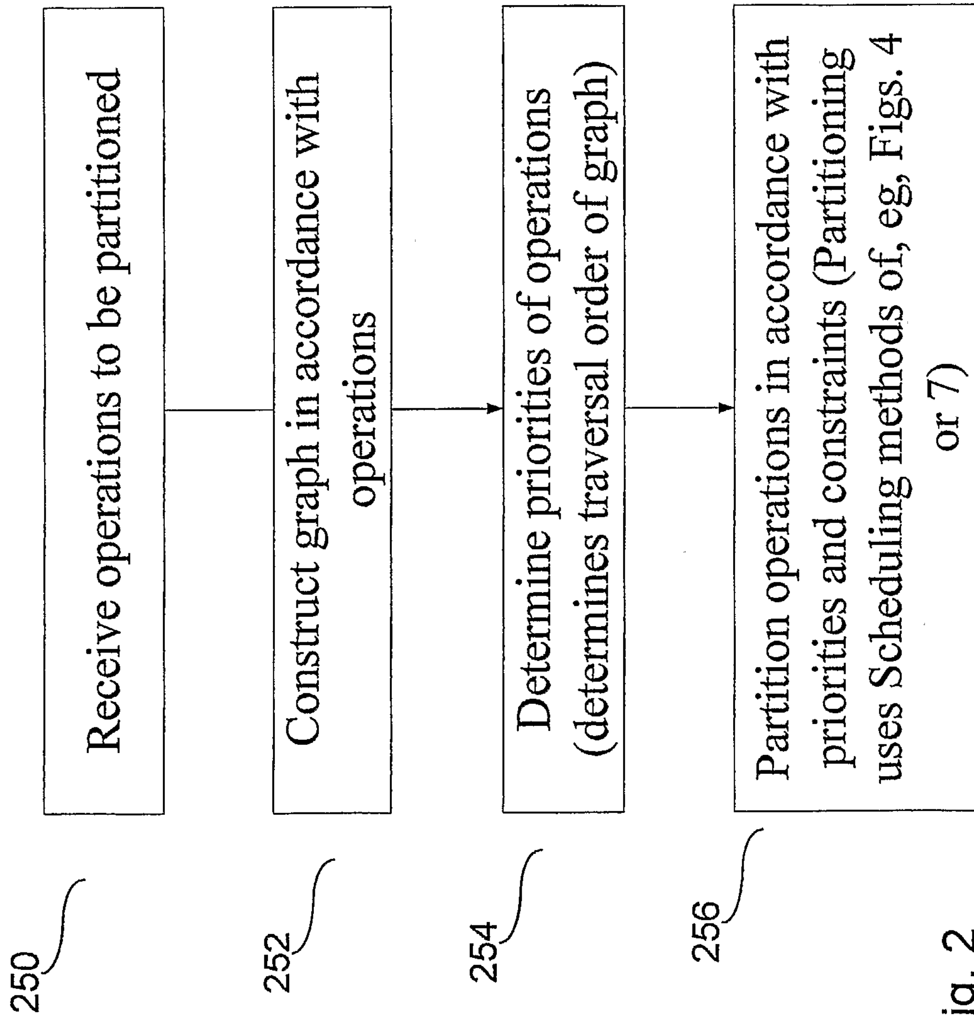


Fig. 2

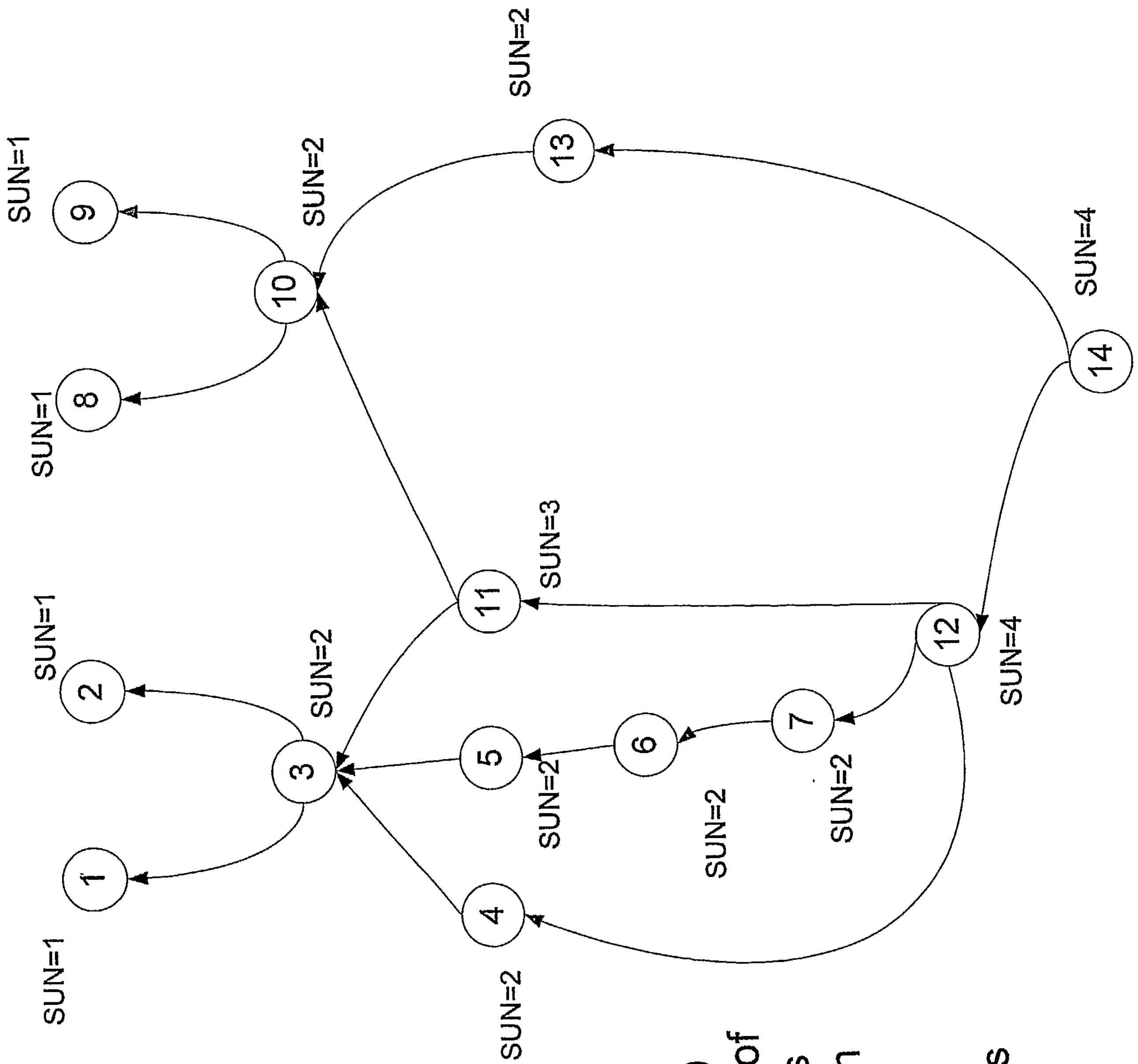


Fig. 3(a) Example of Priorities based on Sethi-Ullman Numbers

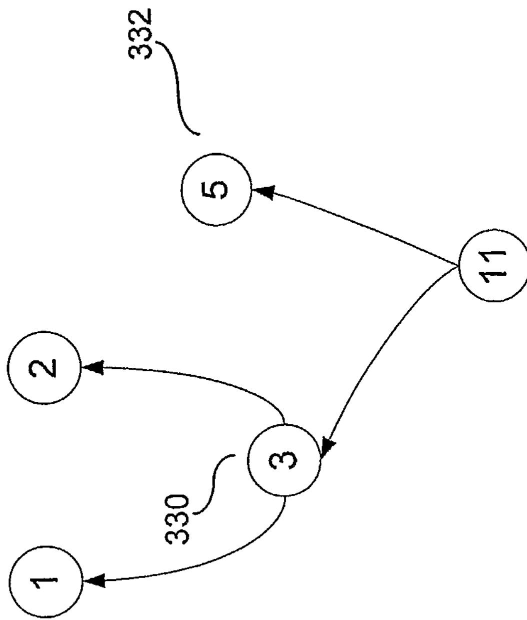


Fig. 3(b) Example of More Predecessors (For Priority Scheme)

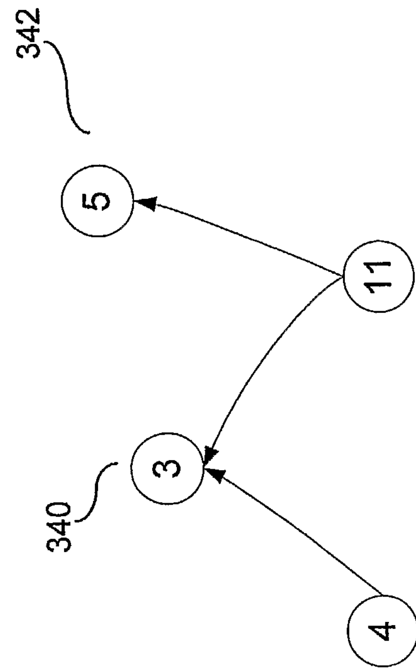


Fig. 3(c) Example of More Ready Successors (For Priority Scheme)

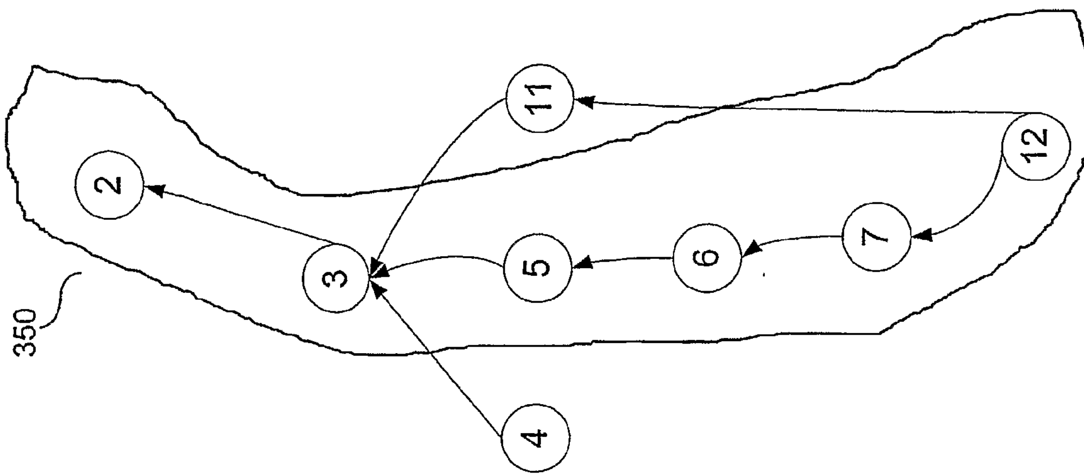


Fig. 3(d)
Example of Critical Path (For Priority Scheme)

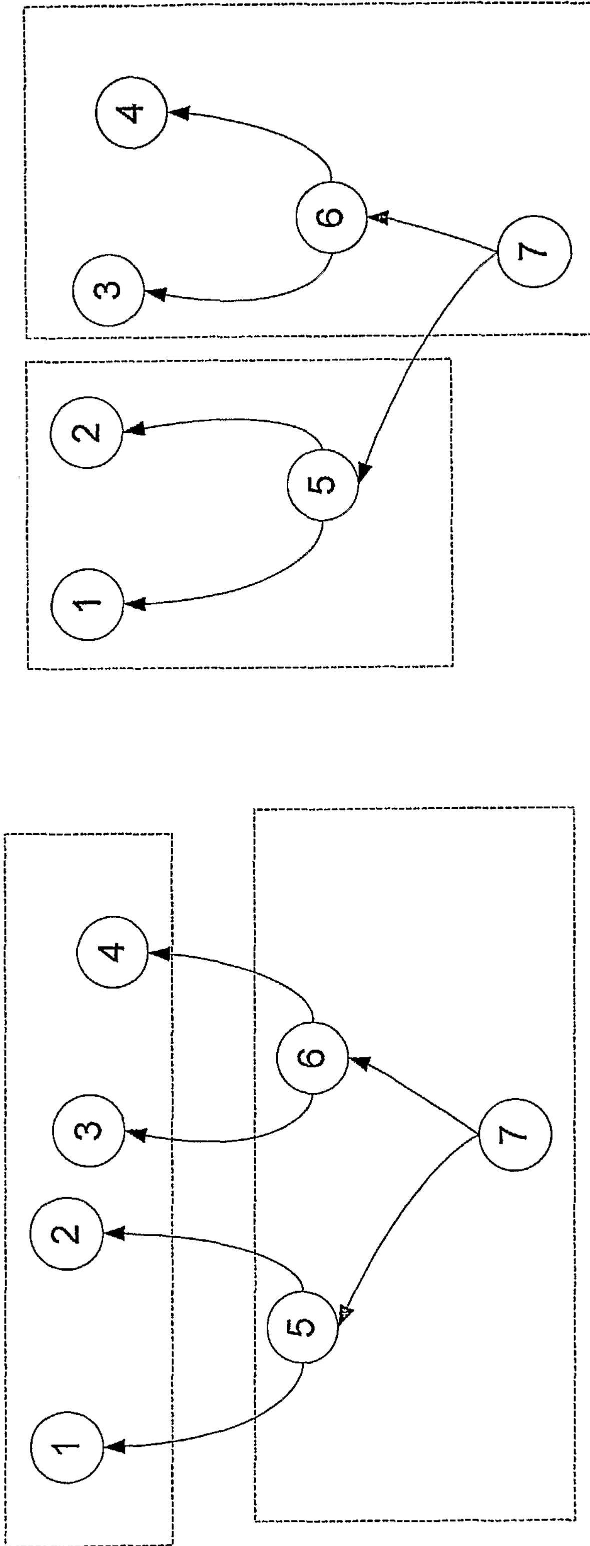


Fig. 3(f)

Fig. 3(e)

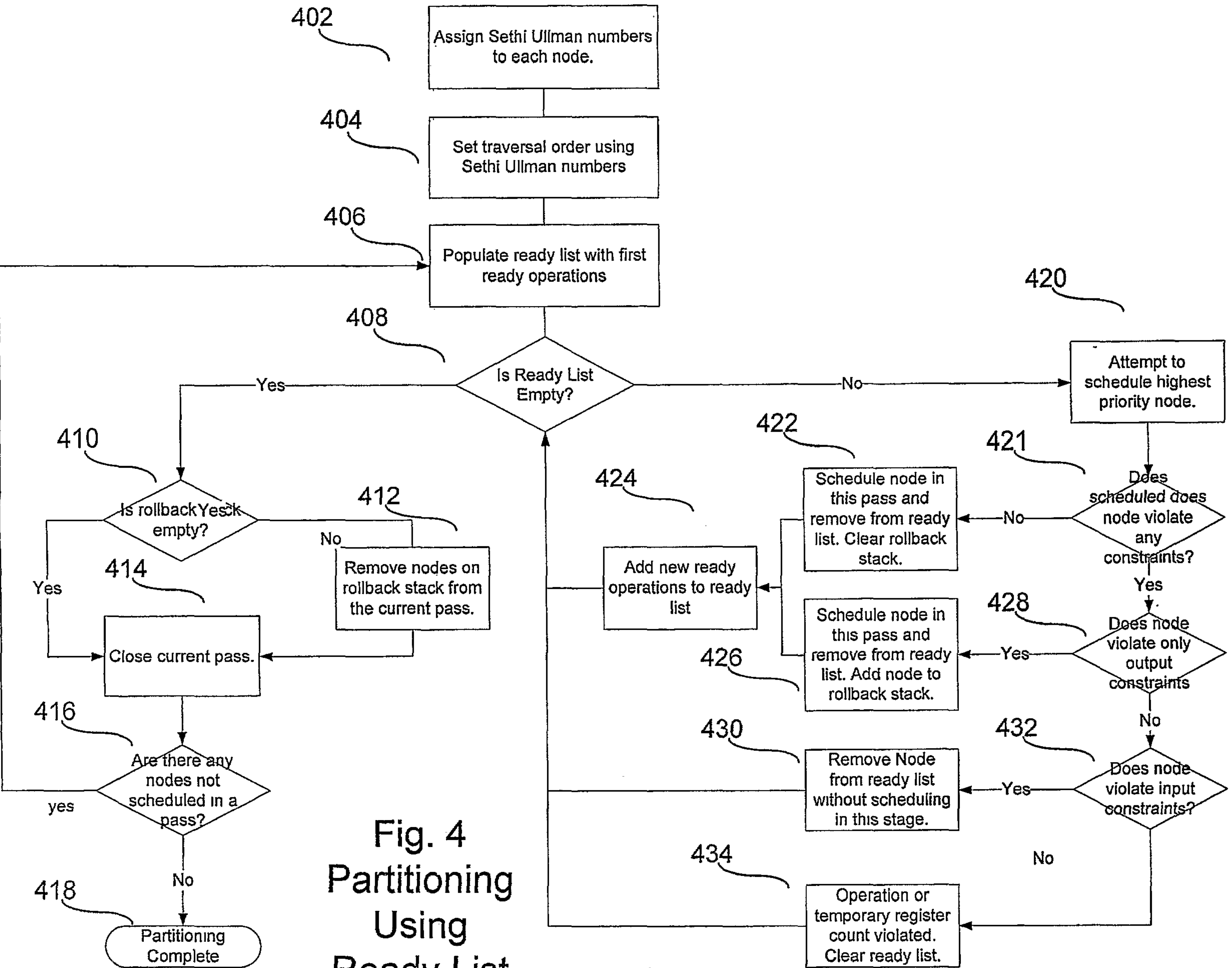


Fig. 4 Partitioning Using Ready List Scheduling

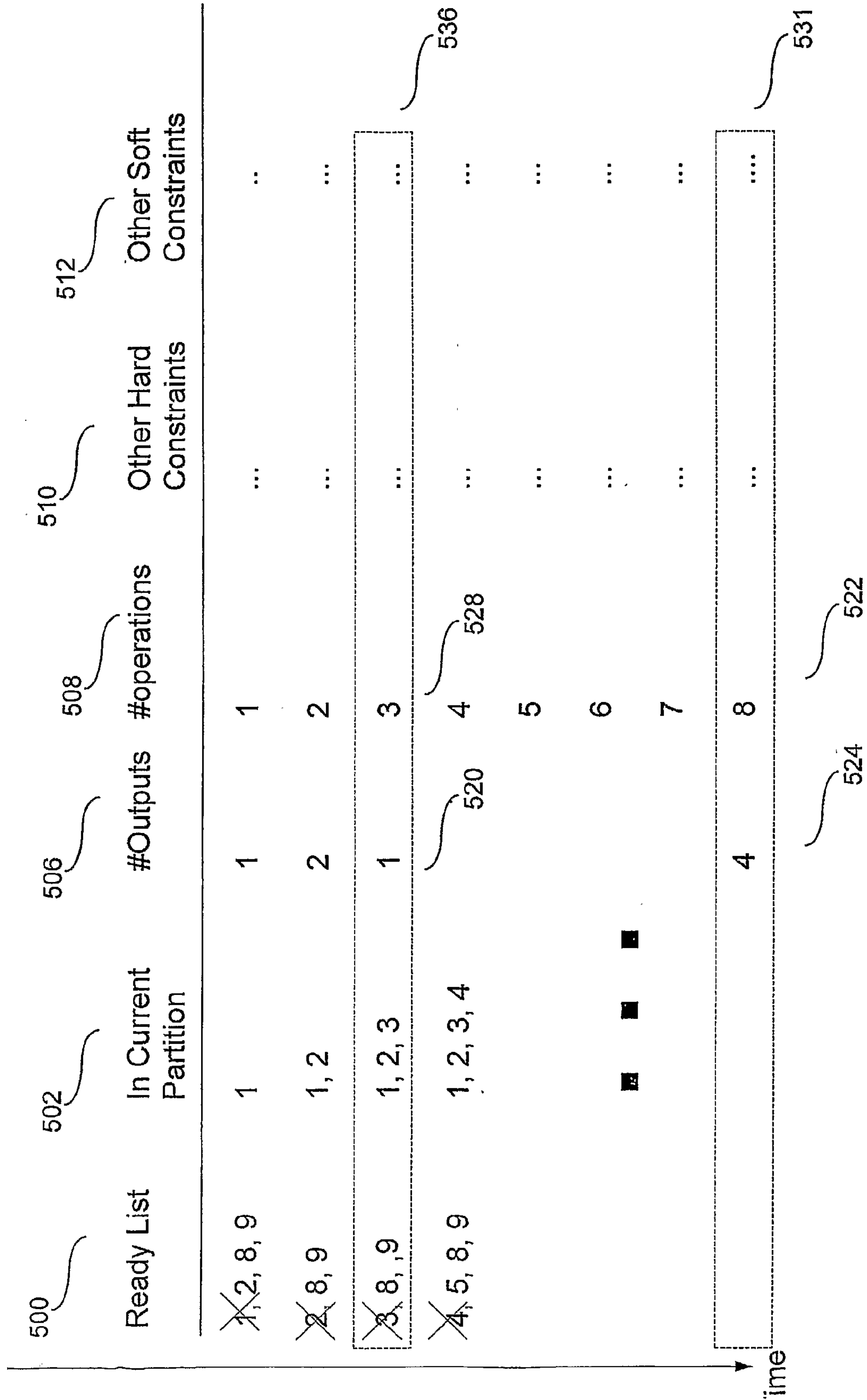


Fig. 5

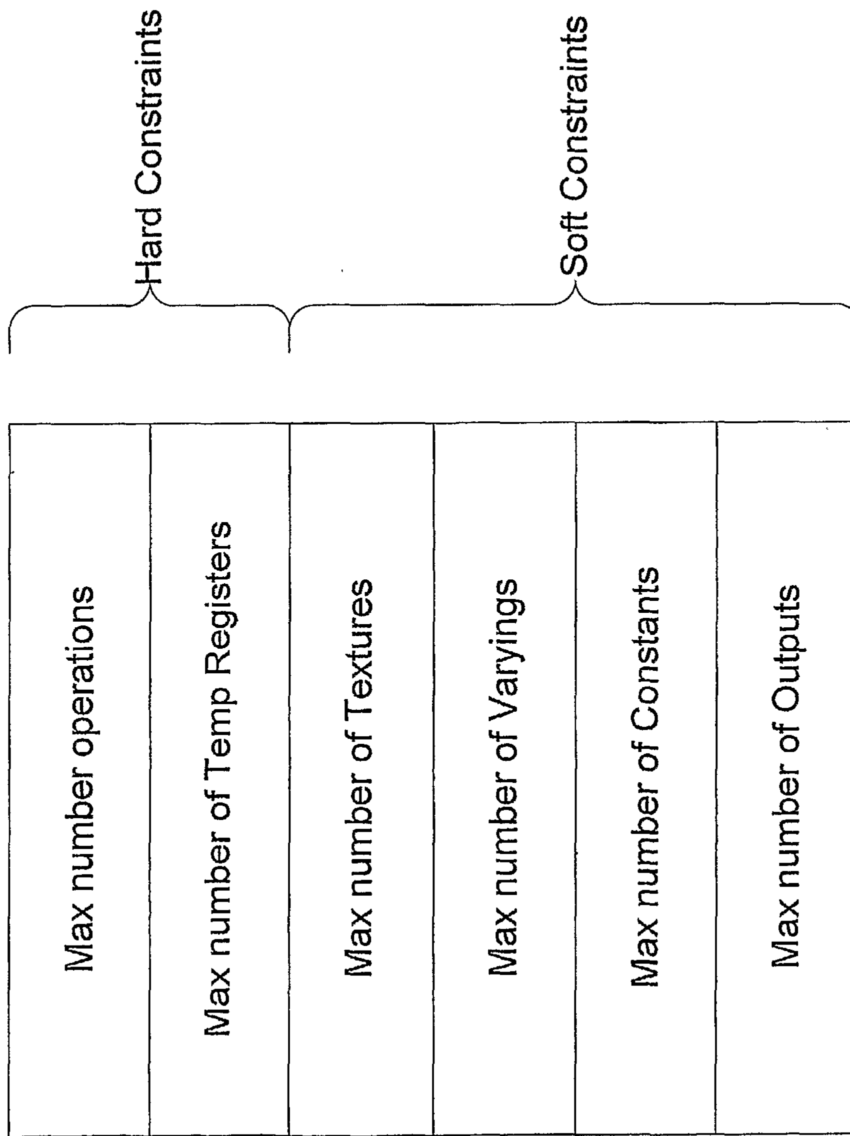


Fig. 6
Constraints for Particular Hardware

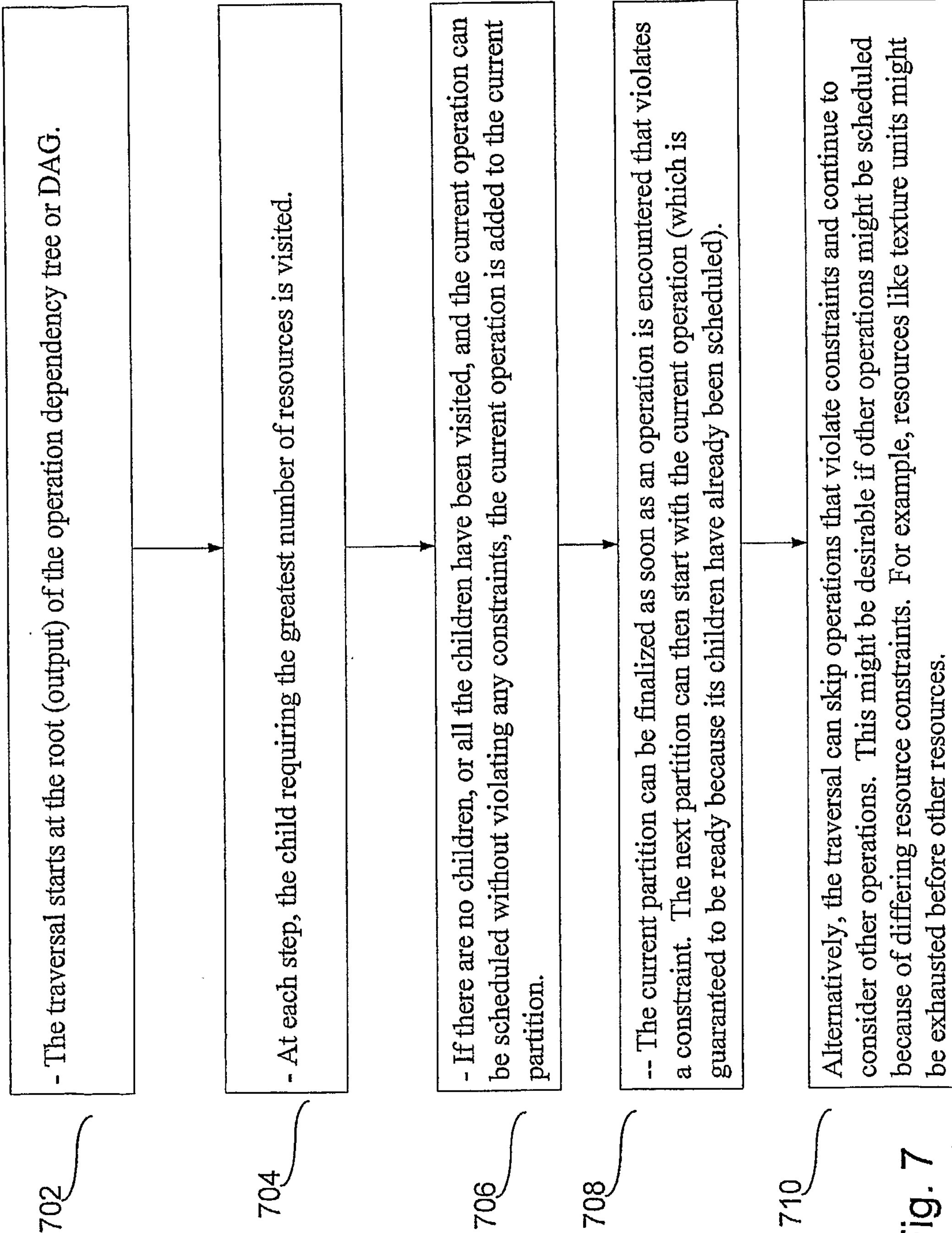


Fig. 7 Partitioning Using Depth First Scheduling