On-Line Control and Deadlock-Avoidance in a Page-Parallel Multiprocessor Rasterizer

Yitzhak Birk, *Member, IEEE*

***Abstract*—A rasterizer converts a document described in some page-description language into a sequence of full-page bitmaps (pagemaps), which can then be printed or displayed. The Page-Parallel rasterizer harnesses multiple processors to work on the same document, thereby permitting cost-effective high-speed rasterization of complex documents. Any given page is processed by a single processor, hence the name. For performance reasons, it is desirable to permit out-of-order rasterization as well as to share memory and computation results among the processors. However, this can result in deadlock. This paper presents on-line algorithms for controlling the rasterizer so as to avoid deadlock without being overly restrictive. We show that previously-proposed approaches for deadlock-avoidance cannot be applied directly due to a special form of nonexclusive allocation of shared resources. We then present a solution, thereby extending the applicability of deadlock-avoidance. We expect our approach to be useful in a variety of similar situations that may occur in other applications.**

***Index Terms*— Deadlock-avoidance, multiprocessor rasterizer, on-line algorithms, parallel computing, parallel rendering.**

I. Introduction

A. The Page-Parallel Batch Rasterizer

A BATCH rasterizer receives a description of a document's contents in some page-description language and converts it into two-dimensional arrays of pixel values (intensity and color), each such array containing the values for a single page. We refer to the input form as *datastream* and to the final result (per page) as a *pagemap*. Batch rasterizers are used primarily with laser printers. (An incremental rasterizer may receive an existing pagemap along with a list of required modifications.)

The last several years have brought about a dramatic increase in the sophistication of page-description languages and their use by application programs. This has resulted in a growing gap between the rate at which pagemaps can be printed, which has been in excess of 200 pages per minute for over 15 years (e.g., IBM 3800) and the rate at which pages can be rasterized. To date, there are no microprocessor-based rasterizers that can match those printing rates for complex pages. Although microprocessors are becoming faster with time, the amount of computation per page will also increase with the introduction of color, multiple intensity levels, and higher resolution.

The Page-Parallel rasterizer architecture harnesses multiple microprocessors to achieve high rasterization throughput in a cost-effective manner. This rasterizer consists of a set of processor-memory elements as well as shared memory. It employs a "Page-Parallel" approach, wherein each page is rasterized by a single processor, which is referred to as a "Page Imaging Processor" (PIP). The input datastream is "scanned" sequentially to detect page boundaries, page size and some other information. The task performing this operation is referred to as the *Scanner*. The conversion of each page from datastream to pagemap is then carried out by a PIP in two stages: i) conversion into an intermediate command stream (ICS) and ii) conversion of the latter into a pagemap. The tasks performing the two steps are referred to as *Imager* and *Builder*, respectively. Any given page is processed by a single instance of Imager and a single instance of Builder. For simplicity of exposition, each (Imager, Builder) pair can be thought of as running on the same processor. The Page-Parallel architecture is depicted in Fig. 1.

The rasterization of a page usually requires the repeated use of *objects*, the most prominent of which are text characters. Whenever preparing these objects requires significant processing and they are reused numerous times, the processed versions are cached and shared among the PIP's. (E.g., raster versions of scalable text characters.) We refer to the processed version as *processed* objects, as opposed to *source*. The collaboration among processors in preparing objects and the use of shared memory to store the results introduced coupling among tasks processing different pages.

"Source" objects are always available to the rasterizer when it needs them. This is attained either by installing them permanently in the rasterizer (ROM, cartridges, etc.), by the rasterizer not deleting them until specifically instructed in the datastream and verifying that they are no longer needed for pages represented by earlier parts of the datastream, or by permitting the rasterizer to request such objects from an external server in which they reside permanently.

Based on the above, it appears that "processed" objects may be deleted at the discretion of the rasterizer since they can be regenerated from the source versions. However, several decisions made in the specification of the Page-Parallel rasterizer architecture affect the handling of cached "processed" objects:

- The need for object preparation is discovered by an Imager, at which time the object is either prepared while the Imager waits or else a request is enqueued in a global object-preparation queue. In either case, the Imager requests a chunk of shared memory for the processed object, waits for it to be allocated, and registers (in a shared table) the location of the allocated memory as the

Manuscript received July 27, 1990; revised August 2, 1991.

The author is with the Electrical Engineering Department, Technion–Israel Institute of Technology, Haifa 32000, Israel.

IEEE Log Number 9204623.

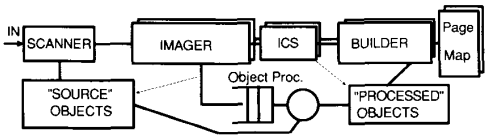

Fig. 1. The Page-Parallel architecture. Any given page is processed by a single Imager and a single Builder. "Source" objects are set aside by Scanner. Imagers request the preparation of "processed" objects and a memory allocation for them. The ICS contains direct pointers to locations of "processed" objects, for use by Builders.

eventual location of the processed object.

- In order to reduce the amount of memory required for ICS and the amount of data-copying that takes place, the ICS contains pointers to the objects rather than copies of the objects themselves. The actual objects are stored in shared memory for use by all processors. Moreover, in the interest of performance and simplicity, the ICS usually contains direct pointers to the location of the cached objects.
- The decision whether to allocate memory for an object may depend on the page for which it is being requested. However, once memory is allocated and the location is registered, any Imager may generate ICS that points to that location and expect the memory not to be recycled until that ICS is consumed by the corresponding Builder.
- A Builder may not request the preparation of an object. (The ICS points to characters by address, not by name, so a Builder does not even know what character to request.)
- An Imager and a Builder may be preempted, but should never have to repeat work. Therefore, their output may never be discarded before it is used, so there is no preemption of processed-object memory. (Object-preparation is not included in this restriction.)

An important implication of the foregoing decisions is that although the "processed" objects can always be regenerated from the "source" versions, they may not be deleted so long as they may be referenced by existing ICS. (This strange flavor of shared resources creates problems, as will be shown later.)

Irrespective of the handling of "source" objects, rasterizer memory management can never involve them in a deadlock situation. In the remainder of this discussion, we are therefore concerned only with "processed" objects, which will simply be referred to as "objects" unless stated otherwise.

We assume the following memory organization for the rasterizer: each PIP has some private workspace and possibly private memory for a pagemap. There is also shared memory, which is partitioned into pagemap memory, memory for processed objects ("object memory") and memory for other purposes. For simplicity of exposition, the partitions will be assumed fixed and the "memory for other purposes" will be assumed unlimited. Nevertheless, our results also hold for finite total memory and flexible partitions.

B. Potential Deadlock

For deadlock to occur, the following must all be true [1]:

- There must be shared resources which are held on an exclusive basis.
- Some task must be holding on to some shared resource (exclusively) while waiting for some other shared resource.
- The graph representing the hold/wait dependencies among tasks must contain a cycle.
- Preemption that frees resources which are part of the cycle must not be permitted.

Violating any of the above suffices to guarantee a deadlock-free system.

The foregoing design decisions give rise to the possibility of deadlock, as illustrated by the following example. Consider a 2-PIP system with a single pagemap per PIP and 3 pagemaps in shared memory, and a 10 page document in which the processing (time) required for rasterizing p.1 is 50 times greater than for any other page. As long as PIP 1 is busy with page 1, no pages can be printed, so PIP 2 processes the remaining pages. Pages 2, 3, and 4 are rasterized and placed in the shared memory pagemaps, page 5 remains in PIP 2's local pagemap, and pages 6 through 10 are stored in shared memory in ICS form. Suppose now that p.1 needs more object memory but none is available. Until it gets more object memory, p.1 cannot be printed, so no pagemaps can be freed and the ICS of pages 6 through 10 cannot be consumed. Consequently, no object memory can be freed, and the system is deadlocked. The fact that ICS is involved in the deadlock cycle, even if ICS memory is infinite, proves that the problem would occur even if we had dynamic partitioning of (finite) shared memory.

C. Organization of the Paper

In Section II, we briefly review various approaches to solving deadlock problems. We then carefully analyze the rasterizer problem, casting it into general terms and identifying the key problems, and present our method of deadlock avoidance in such situations. Sections III and IV describe the use of this method in controlling the rasterizer: Section III spells out the exact rules for controlling the rasterization so as to avoid deadlock and enforce some additional policies without overly restricting progress, and Section IV translates these rules into explicit algorithms and analyzes their complexity. Section V concludes the paper.

II. Avoiding Deadlock in the Page-Parallel Rasterizer

A. Deadlock Avoidance

There are three ways of treating deadlock: recovery, prevention and avoidance. However, recovery is precluded since it would require that we "roll back" Imager or Builder. Prevention and avoidance are similar, in that both are preemptive actions. With prevention, one effectively considers the "static" worst case scenario for all contending tasks in deciding whether to allocate resources to a given task. In other words, it is always assumed that a task may concurrently need all the resources that it ever uses. Avoidance, on the other hand, takes into account things such as the order in which resources are requested by a task and whether some are freed before others are required. This careful, dynamic evaluation of the situation

results in more permissive policies and consequently in higher utilization of the resources and higher performance. We opt for a dynamic form of deadlock-avoidance, both to improve performance and because some of the resource-requirements are not known in advance, so prevention could effectively result in the processing of one page at a time.

Deadlock-avoidance usually works as follows [1]. Given *a priori* information about the amount of resources of each type that may be requested by each task, a deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. This state is characterized by the number of available and allocated resources, and the maximum demands of the tasks. A state is safe if the system can allocate resources to each task (up to its maximum) in some order and still avoid deadlock. More formally, a system is in a safe state if and only if there is a sequence of tasks $(p_1, \cdots, p_n)$ such that if the allocation being considered is made the tasks can subsequently be allocated all the required resources in the order specified by the sequence, and such allocation results in all of them completing their jobs. Note that it suffices that the resources required by some task, say p_i, become available only after prior tasks in the sequence have completed their work and released their resources.

The first deadlock-avoidance algorithm to follow the foregoing approach was the "Banker's algorithm" due to Dijkstra [2]; it dealt with a single resource type. (The name is due to the fact that bankers must always make sure that they can meet the demands of all customers in some order.) This was later extended to multiple resource types by Habermann [4]. Holt was the first to formalize the notion of deadlocks in terms of a graph theoretical model that could be used to represent our approach [5], [6]. These approaches are further extended in [7], [8]. An overview of the deadlock problem appears in [9], and [12] contains a classifying bibliography. A comparison among concurrency-control methods using analytical models is presented in [10]. For a characterization of situations that permit the allocation of an available unit of resource to any requesting processor without running the risk of deadlock, see [11].

B. Application to the Page-Parallel Rasterizer

The resources involved in the deadlock-causing cycle are pagemap memory and object memory. For facility of exposition, let us assume that each is partitioned into fixed units, to which we refer as pagemap and obmap (object map), respectively.

Pagemap memory is a shared resource. Each page requires exactly one pagemap, which is held on an exclusive basis until the page is printed. It is then freed.

Object memory is best thought of as being dynamically partitioned into a pool of blank obmaps (this is the free portion) and a set of labeled obmaps. Each labeled obmap holds a specific object at any given time. A request for object memory is really a request for a shared "read lock" on an obmap with a particular label. If an obmap with the desired label (object) is not present, a blank one is requested and, if granted, it is removed from the pool, labeled, locked, and used to house the specific obmap that was requested at the outset. Additionally, some mechanism would be implemented to prevent reading before the obmap content is in place.

The granting of a blank obmap from the pool may be subject to various policies which are beyond the scope of this paper and do not affect its results. (We will use availability of memory as the criterion.) There is, however, no restriction on addition of read locks to labeled obmaps (piggyback). In fact, even an Imager whose request for a blank obmap had earlier been refused may place a read lock on that same obmap if it has been allocated to another Imager which uses it for the same object. An obmap is returned to the pool only if there are no locks on it. Since the ICS contains direct pointers to obmaps, locks can only be removed once the ICS is consumed. Attached to each obmap or a set of obmaps is a count of the number of locks placed on it, and the ICS contains lock-removal commands. The decision as to which of the unlocked obmaps should be recycled is left to the memory management system.

Our deadlock-avoidance strategy hinges on the fact that once memory for a pagemap is allocated to the Builder of a given page (in private or shared memory), all the ICS generated by that page's Imager can be consumed even if the Imager has not completed its work on the page. This, in turn, removes all the obmap locks placed by that page. Although we have no control over which page points to which objects and it is perhaps even too expensive to keep track of this at every obmap, it is still true that if we allocate pagemap memory for all pages whose Imaging has started then all ICS can be consumed, all locks will be removed and object memory can be freed as required.

Having described the deadlock situation in the context of the rasterizer, we now proceed to cast it in more general terms in order to facilitate its application in other situations.

C. Abstraction of the Problem and its Solution

Our system receives a sequence of "jobs" (pages). A given job, say J_i, must be processed by a sequence of two tasks, I_i and B_i; the output of I_i may be piped to B_i before I_i is completed. (In practice, the number of instances of a task would equal the number of active jobs, since a completed job frees up the tasks that processed it.) The incoming jobs may be processed in any order, but job completion (specifically, the freeing of resources by type-B tasks) must occur in job-sequence order. A task of type I requires a specific set of labeled chunks of resource R^I, which is not known in advance. A task of type B requires a single unit of resource R^B. Finally, the allocation of a unit of R^B to B_i permits the consumption of the entire output that has been generated by I_i, thereby releasing the latter's hold on any chunks of R^I without need to acquire additional such chunks in the process.

A chunk of resource R^I may be shared among several tasks of type I provided that they use it for the same purpose. Thus, such tasks may acquire a shared read lock on chunks of R^I. The initial granting of a "blank" chunk of R^I to a requesting task, I_i, may be governed by various policies. Once granted, however, any other task, say I_j. may add its lock to that chunk,

and a chunk may only be freed and reallocated (for a different purpose) once all locks have been removed from it. A task of type B requires one unit of R^B when it begins, holds it on an exclusive basis, and frees it once it and all $B_j, j < i$ have completed.

The problem of managing R^I thus combines features of databases, wherein one is interested in a specific record, and memory allocation where memory is granted by quantity without requirement of a specific cell. The "database" portion alone would be trivial (unlimited shared read locks), and the memory aspect alone is not new either. However, the combination of the two along with the lack of any advance knowledge of the requirements of any given type-I task for R^I, the unrestricted placement of read locks, and the inability to recycle a chunk of R^I until all locks are removed complicates matters and prevents a direct application of previous approaches.

Instead of directly involving the management of R^I in our deadlock-avoidance scheme, we control the permission for type-I tasks to begin work on a new job and the allocation of R^B to type-B tasks. This is used to guarantee that if task I_i is permitted to begin processing job J_i then task B_i can eventually receive a unit of R^B without having to grant any more locks or allocations of R^I to any type-I task in this process. In other words, we are able to guarantee that the system is in a safe state at all times.

III. Resource-Allocation Rules

Having sketched our approach to deadlock avoidance, we now apply it to the on-line control of the rasterizer. We begin by spelling out the exact rules for granting an Imager permission to commence work on a page and allocating a pagemap to a Builder. These rules can be used by any scheduler as an oracle which determines whether a desired scheduling step is legal. This can serve both to pick a legal scheduling option from those being considered and as a pacing mechanism for a chosen processing order.

A. Preliminaries

Because of the printing order constraints, we are guaranteed that if some Imager is blocked due to a request for object memory that cannot be granted then either the problem resolves itself or else all Imagers will eventually be blocked. For the purpose of deadlock avoidance, it is therefore sufficient to consider the case of all Imagers blocked, and to guarantee that when this happens all ICS can be consumed without need to generate any new ICS (and pointers to objects) in the process. Before proceeding, we introduce some more details of the system which must be taken into account in making allocation decisions. We are not introducing additional resources.

Throughout the discussion, we will assume that when a pagemap request for a page is being considered, the pagemap sizes of all lower-numbered pages are known. These sizes would easily be determined by the scanner, which encounters the pages in sequence and before any other task does. (Pagemap size is a simple function of paper size and properties of the printing device; it is not altered by the content of the page.)

Due to properties of the printing mechanism as well as performance considerations, it is sometimes required to guarantee pagemaps for a number, say N_p, of "priority" pages. These have the lowest page numbers. (More precisely, they are the first ones in the printing order.) Pagemaps can only be allocated to later pages once pagemaps have been allocated or reserved for the priority pages. For example, in a duplex printer in which both impressions are made in one pass, N_p is at least two. We refer to this as the *pagemap priority policy*. Note that this does not preclude out-of-order processing, so long as the priority pages have their guarantees. Shared memory contains at least $N_p \cdot maxSize$ pagemap memory, where *maxSize* is the maximum size of a pagemap. (This may be relaxed in special cases, such as the use of large paper only for job separation.) The notation used in stating the rules is as follows:

- M_t Total amount of pagemap memory in bytes.
- i Lowest page number that has yet to print.
- N_p Number of contiguous pages, beginning with i, for which pagemaps must be reserved. (Priority pages.)
- M_p Aggregate pagemap memory size of the N_p priority pages.
- N_c The number of contiguous pages, beginning with i, which have been completed by their Imagers (no additional object memory will be needed for them). If printing is in singles and in page-number order, these can all be printed and their pagemaps can then be freed.
- N_{ac} Number of "active" pages. A page becomes active when its Imaging is begun, and ceases to be active once it becomes one of the N_c pages.
- $P^i_{\max}$ Highest page No. whose Imaging has begun.
- $P^p_{\max}$ Highest page No. which has been allocated a pagemap.
- j Denotes an arbitrary page.

Additional notation will be introduced as required.

We now state necessary and sufficient conditions for allocating a pagemap for a page, e.g., page number j, and for permitting its Imaging to begin. Although the latter always precedes the former, the pagemap allocation is discussed first since the ability to allocate a pagemap influences the decision whether to permit Imaging of j to begin. For each set of conditions, we will show that they are necessary for deadlock-avoidance or for satisfying the pagemap priority policy. We will also prove that the two sets are jointly sufficient.

B. Pagemap Allocation

A pagemap may be allocated to a given page, say j, if and only if *all* the following conditions are satisfied:

- A-1 there is sufficient free pagemap memory (obvious),
- A-2 the allocation leaves enough pagemap memory for those of the N_p priority pages which have not yet received a pagemap, and
- A-3 the allocation to j can be shown not to lead to a future violation of the pagemap priority policy.

The last condition differs from the second one whenever page size is variable as well as when certain printing orders are used.

The following examples illustrate the necessity of the different rules.

Example 1: Direct violation of the pagemap priority policy; $N_p = 2; M_t = 5$. (Rule A-2.)

Page Number	1	2	3	4	5	6
Imaging begun	+	+	+	+	+	+
Imaging completed	+	+	–	–	–	–
Pagemap allocated	+	–	+	+	+	?
Page size	1	1	1	1	1	1

Allocating a pagemap to p.6 would take up the last free pagemap, thereby preventing p.2 from receiving one until p.1 is printed. This would violate the policy since $N_p = 2$, and the violation could lead to deadlock (e.g., if pages 1 and 2 must be printed together on different sides of the same sheet).

Example 2: Eventual violation of the pagemap priority policy; $N_p = 2; M_t = 8$. (Rule A-3.)

Page Number	1	2	3	4	5	6
Imaging begun	+	+	+	+	+	+
Imaging completed	+	+	+	+	–	–
Pagemap allocated	+	–	–	–	+	?
Page size	1	1	5	1	1	5

Allocating a pagemap to p.6 would not directly violate the policy, since the total amount of allocated pagemap memory would become 7, leaving 1 for p.2. However, once p.1 was printed and p.3 became one of the preferred pages, the amount of free pagemap memory would be 1 (p.2 would still have a pagemap at this time), whereas p.3 would need a pagemap of size 5. The problem here is due to the fact that the amount required by p.3 is larger than that released by p.1. The eventual violation, when it occurs, can also result in deadlock. The foregoing rules are thus clearly necessary to avoid deadlock while adhering to the priority policy.

C. Permitting Imager to Begin

Imaging of a page, say j, may begin if and only if *all* the following conditions are satisfied:

- I-1 if all Imagers were blocked and Building and printing continued as much as possible (pagemaps of printed pages would be freed and allocated for Building of subsequent pages and so on until no more pages could be printed), there would be sufficient pagemap memory for all remaining pages whose Imaging had started, and
- I-2 all said pagemap allocations would conform to the pagemap-allocation rules.

It should be noted that if Imaging of pages is begun in printing order, satisfying the first condition guarantees that the second one is also satisfied.

The following examples illustrate the necessity of the conditions and expose some subtleties.

Example 3: j cannot receive a pagemap; $N_p = 2; M_t = 3$. (Rule I-1.)

Page Number	1	2	3	4	5
Imaging begun	+	+	+	?	–
Imaging completed	–	+	–	–	–
Pagemap allocated	+	+	+	–	–
Page size	1	1	1	1	1

If Imaging of p.4 were permitted to begin and Imagers were subsequently blocked, there would be no printable pages. Therefore, pages 1, 2, and 3 would never free their pagemaps and p.4 would not be allocated one. As a result, p.4 would still have references to objects, which would thus have to remain in memory, possibly preventing Imaging of p.1 from progressing.

Example 4: Permitting j to start may violate the pagemap priority policy; $N_p = 2; M_t = 3$. (Rule I-2.)

Page Number	1	2	3	4	5
Imaging begun	+	–	–	+	?
Imaging completed	–	–	–	+	–
Pagemap allocated	+	–	–	+	–
Page size	1	1	1	1	1

If Imaging of p.5 were permitted to start and Imagers were subsequently blocked, there would be enough pagemap memory for all active pages, including p.5. However, allocating a pagemap to p.5 would violate the priority policy, since there would be no pagemap for p.2 when it needed it. This example proves that it is not sufficient to consider the pagemap priority policy when deciding whether to allocate a pagemap; it must also be factored into the decision whether to permit the beginning of Imaging of a page.

Example 5: Permitting j to start would cause an indirect problem; $N_p = 2; M_t = 3$. (Rule I-1.)

Page Number	1	2	3	4	5
Imaging begun	+	+	?	–	+
Imaging completed	–	–	–	–	–
Pagemap allocated	+	–	–	–	–
Page size	1	1	1	1	1

Initially, it appears that Imaging of p.3 should be allowed to begin, since a pagemap may be allocated to p.3 when desired (pages 1 and 2 are the preferred ones, but there are 3 unallocated pagemaps. Also, p.3 has priority over p.5). However, if all Imagers were blocked, p.5 would never receive a pagemap since p.1 would not be printed. Thus, p.5 would prevent resources from being freed and deadlock could occur. This example shows why the rule must be that "it must be possible to give a pagemap to every active page" rather than

merely to j. It can also clearly be observed that this problem would never occur if Imagers were started in page-number order.

Proposition 1: The five conditions for pagemap allocation and permission to start Imaging of a page are jointly necessary and sufficient for deadlock avoidance and enforcement of the pagemap-allocation policy.

Proof: Given that deadlock must be avoided without violating the pagemap priority policy, the five foregoing examples prove that the conditions are necessary.

If we adhere to the rules for permission to begin Imaging, we are guaranteed that all ICS is consumable; this, in turn, breaks the apparent hold/wait cycle, thereby violating one of the necessary conditions for deadlock. The rules for allocating a pagemap, which form part of the test for beginning Imaging, further guarantee that the pagemap priority policy is never violated, even during the allocation of pagemaps for the purpose of releasing object memory to avoid deadlock. □

IV. Algorithms for Implementing the Rules

In Section II, we characterized a safe state. In this section we present efficient algorithms for testing the safety of the state following the allocation of a pagemap or the permission to an Imager to commence working on a page. Throughout the discussion, we use the term "page number," implicity assuming that pages are printed in page-number order. Nevertheless, other printing orders can be accommodated by simply interpreting "page number" as the printing order, provided that the pagemap priority policy is also stated in terms of the printing order. The algorithms presented shortly constitute a precise implementation of the rules. The examples that served to prove the necessity of the different rules and exposed their subtleties also help understand the algorithms.

A. Pagemap Allocation

The willingness to consider out-of-sequence pagemap allocations is important. For example, a good operating policy may be to request a pagemap for a page only when the size of the ICS that has been generated for it approaches the size of the pagemap (no savings by keeping it in ICS form, possibly pinning numerous objects) or its time to print is approaching. As another example, consider a case wherein the amount of processing required for pages is known in advance. (This would happen if the document had been rasterized in the past and is archived in datastream form along with the relative rasterization time of each page.) It would be useful to begin work on a difficult page as early as possible, but this may require a pagemap.

Pages (and therefore pagemaps) may vary in size, and this introduces a possibility of deadlock whenever several small pages are followed by large pages. A conservative solution would be to reserve $M_p = N_p \cdot maxSize$ pagemap memory for the priority pages. However, assuming that the Scanner discovers the sizes of pages it scans, that pages are scanned in page-number order, and that a page is never allocated a pagemap before the Scanner discovers its size, we can do better. The general idea is to first determine how much of the free pagemap memory must be set aside in order to avoid violations of the priority policy, be they immediate or eventual, and to subtract that from the amount of free pagemap memory. If the result is greater than the amount requested by j, a pagemap is allocated. Otherwise it is not.

The test is carried out by considering a sliding window containing N_p consecutive page numbers. In its initial position, the window contains pages i through $i + N_p - 1$, and sliding stops when it contains pages $j - N_p$ through $j - 1$. For each position of the window, we sum up the sizes of the pages contained in it and refer to the results as the total memory size of that window. This is denoted by $M_{wt}(k)$, where k is the lowest page number in the window. We then compute the maximum of $M_{wt}(k)$ over k in the range i through $(j - N_p)$ and use that in the test for allocating a pagemap to j.

The above approach would prevent deadlock, but is overly restrictive in two ways:

- Although $M_{wt}(k)$ indeed represents the required amount of pagemap memory for the k-window, some of the pagemaps for this window may have already been allocated. No free memory need be reserved for those pages.
- Before $(k, k+1, \cdots, k+N_p-1)$ becomes the priority-page window, all pagemaps for pages with numbers smaller than k will have been freed. The pagemap memory already allocated to those can be relied upon for pages in the k-window.

Therefore, the real amount of memory that still needs to be reserved for the k-window, denoted $M_{wr}(k)$, is given at the bottom of this page. We now slide the window and obtain the amount of unallocated pagemap memory that must be reserved for earlier pages than j as

$$M_r(j) = \max\{M_{wr}(k)\}, \qquad k = i, i+1, \cdots, j - N_p. \quad (1)$$

For any given k. $M_{wr}(k)$ can be negative (more memory will be freed than is needed). However, we clearly cannot allocate

---

$$\begin{aligned} M_{wr}(k) &= M_{wt}(k) \\ &\quad - (\text{pagemap mem. already allocated to pages } k, \cdots, k+N_p-1) \\ &\quad - (\text{pagemap mem. already allocated to pages } i, i+1, \cdots, k-1) \\ &= M_{wt}(k) \\ &\quad - (\text{pagemap mem. already allocated to pages } i, i+1, \cdots, k+N_p-1). \end{aligned}$$

memory that is not free (the first condition). Therefore, j can be allocated a pagemap if and only if

$$M_{\text{free}} - \max\{M_r(j).0\} > \text{size}(j).$$

Note that when printing is in page-number order, $M_{ur}(i)$ is never negative, since there are no earlier pages than i in the system.

Proposition 2: The pagemap-allocation algorithm precisely implements the pagemap-allocation rules.

Proof: We use $P^p_{\max}$ to denote the maximum page number that has been allocated a pagemap.

Correctness: (Enforcement of the policy.)

The proof is by induction on events. The events are:

1) printing of page i and freeing of its pagemap
2) allocating a pagemap to page $j, i \le j < i + N_p$
3) allocating a pagemap to page $j, j \ge i + N_p$.

- *Initial conditions:* Assuming that there is at least $N_p \cdot maxSize$ pagemap memory, and that it is initially free, there is clearly no problem at the outset.
- *Induction step:*
  1) The new "critical" window consists of pages $(i+1, i+2, \cdots, i+N_p)$. If $P^p_{\max} > i+N_p$, the requirements of this window were already accommodated whenever the allocation of a pagemap to $j > i+N_p$ was being considered. (The eventual freeing of i's pagemap and its availability for later pages was possibly relied upon when allocating pagemaps to later pages.) If $P^p_{\max} \le i+N_p$, no pagemaps have been allocated to pages beyond the "critical" ones, so the assumption of sufficient pagemap memory for those suffices.
  2) This reduces the amount of free memory, but also reduces the requirements for any window that includes j. Later windows are unaffected, since j's pagemap will be freed before they need it. Earlier j-less windows do not exist since $j < i + N_p$.
  3) We verify that there is sufficient memory for early windows before allocating, so no new problem can be created.

Tightness: A formal proof is omitted. The idea is to relax each of the constraints and show one example of deadlock in each case. Examples 1 and 2 can serve for this purpose. □

It readily follows that any available pagemap memory may always be allocated to a page once all lower-numbered pages have been allocated a pagemap.

B. Pagemap Allocation—Extensions

Pagemap allocation decisions are sometimes complicated by two dynamic changes: i) pagemaps are compressed once completed, and ii) the total amount of memory that may be allocated for pagemaps is allowed to change dynamically. We extend our algorithm as follows:

1) Compression of Pagemaps in Shared Memory: Unlike page-size, the compression ratio is only determined once a pagemap is completed and compressed, and thus is not known to the Scanner. Nevertheless, the algorithm used for variable size pagemaps can be adapted easily to handle this problem. The idea is to initially use the noncompressed size, and to free up excess memory once compression has been carried out. This approach is correct provided that a compressed pagemap is no larger than a noncompressed one.

2) Variable Amount of Pagemap Memory:

Increase in pagemap memory: Change the total pagemap memory size and the amount of free memory. Next, see if any pending requests can be granted, and proceed as usual.

Reduction in pagemap memory size: This must be done carefully, since some of the free memory may have been implicitly committed by letting the Imaging of some page begin. A careless reduction of the free memory could thus result in deadlock. The algorithm is as follows:

- Create a dummy page
  - *pageSize* = desired reduction in pagemap memory
  - *pageNumber* = $P^i_{\max} + 0.5$
- Whenever a page is printed, attempt to allocate a pagemap to the dummy page using the usual algorithm, with the following differences:
  - Give the dummy page priority in the sense that its request is evaluated before those of other pages; the granting is still subject to the regular policy for a page number $P^i_{\max} + 0.5$.
  - The pagemap may be allocated in pieces. (Whenever some memory is allocated to the dummy page, its size is reduced by the amount allocated to it; the dummy page thus never owns any memory.)
- Whenever the dummy page receives memory, reduce M_t by that amount.
- Once the dummy page receives the entire requested amount, delete it.
- In deciding whether to permit the Imaging of a page, say j, to begin, distinguish between two cases:
  - if $j < P^i_{\max}$, there is no change.
  - if $j > P^i_{\max}$, subtract the current size of the dummy page from M_{free} (possibly causing it to become negative). Other than this, ignore the dummy page in the calculations.

Proposition 3: The extended pagemap-allocation algorithm is correct and tight

Proof:

Correctness: The assignment of a page number higher than $P^i_{\max}$ to the dummy page, along with the use of this number in the pagemap-allocation decisions and with the fact that the dummy page does not affect the decision to start Imaging of lower-numbered pages, guarantees that allocating pagemap memory to this page cannot cause deadlock involving pages numbered $P^i_{\max}$ and below. In deciding whether to permit Imaging of later pages to begin, the adjustment of M_{free} to correspond to the situation after the completion of the memory reduction guarantees that Imaging of such pages can begin only if deadlock can be avoided without relying on memory that is to be removed. Finally, no later pages can receive pagemaps before the dummy page is removed, so pagemap allocation to later pages is not a source of concern.

Tightness: The priority given to the dummy page in consideration for receiving pagemap memory (the granting is per the rules, to avoid deadlock), along with the permission to allocate memory to it incrementally, guarantee that the requested memory reduction is achieved at the earliest possible time. (An earlier allocation would violate the pagemap-allocation rules, which have already been shown to be tight, and could thus result in deadlock). □

C. Permission to Start Imager

With a general printing order, it is impractical to implement the rules in closed form. Instead, our algorithm assumes the form of a program that *simulates* the blocking and printing process as well as the allocation and freeing of pagemaps. If the program succeeds in allocating a pagemap to all active pages, including j, then j's Imaging may begin. Else it may not. Clearly, the regular allocation rules must be obeyed. Following is a skeleton of such a simulation program.

```
Permission to begin Imaging (j)
   Stop all Imagers; /*this is the worst
                      case*/
   REPEAT
      Print all Printable pages and free
         their pagemaps;/*in specified
                         order*/
      Allocate free pagemaps to next in
                       line;
      Build all pages whose Imaging has
         been completed and which have a
         pagemap;
   UNTIL (no new pages Built);
   IF (possible to allocate a pagemap
         to all active pages including j)
            RETURN (OK); /*to start j*/
   ELSE RETURN (not OK);}
```

Comments:

- The loop is equivalent to "print all eventually-printable pages and free their pagemaps."
- The allocation test is conducted as discussed earlier, including all the generalization, so those need not be addressed again.
- It is possible to insert the test for allocation to all active pages into the loop and exit if OK, potentially saving iterations.
- Whenever pages are printed in bunches of b (e.g., in some duplex printers $b = 2$), we slide the window in steps of b pages, visiting only the feasible positions. (b may differ from N_p.)
- In specific implementations with a known printing order and page size properties, a simpler equivalent of this program will usually be easy to construct.

D. Incremental Computation and Complexity

Pagemap allocation: Two events alter the information on which this decision is based: i) pagemap allocation, and ii) release of a pagemap following the printing of a page. The affected variables are $M_r(j), M_{wr}(k)$ and M_{free}.

We store $M_{wr}(k)$ for all values of k representing pages that are in the system. For every event involving a page, say j, (freeing a pagemap, allocating a pagemap, etc.), $M_{wr}(k)$ needs to be updated only for $j - N_p < k < j + N_p$. This is a fixed amount of work. Given that there are exactly two such events per page, it also represents a fixed amount per page.

The situation with $M_r(m)$ (m is an arbitrary number) is more complicated, since a change in $M_{wr}(k)$ can ripple through and influence $M_r(m)$ for all pages $m > k$. Our approach is to store $M_r(m)$ for all pages in the system, and to update it whenever a pagemap is allocated or freed. The expression for $M_r(m)$ in (1) can be computed incrementally as

$$M_r(m) = \max\{M_r(m-1), M_{wr}(m - N_p)\}. \quad (2)$$

The worst-case cost is thus on the order of the number of pages in the system (per page). The operations are very simple, so the cost is low even if there are hundreds of pages in the system. Moreover, the process need not propagate beyond the first page whose value remains unchanged.

If there is a guarantee that a pagemap is never requested for a page with number greater than $(i + \text{constant})$, the direct approach may be more economical.

Permission to begin Imaging: This information must be updated whenever one of the previous events happens as well as whenever Imaging of a page is completed. The updates consist of two elements: stepping the "printing simulator," and determining whether it would be possible to allocate a pagemap for every remaining active page.

Stepping the simulator clearly requires constant work per step, since there is exactly one candidate for printing and one to receive the next pagemap that is freed. Moreover, storing the simulator's state enables it to only make forward progress, so there is a fixed number of steps per page. Along with the simulator state, we keep a data structure identical to the one for pagemap allocation. The information in this structure, however, reflects the printing state of the simulator, not of the real machine. This data structure is used to determine whether all active pages, including the page for which beginning of Imaging is requested, will be able to receive pagemaps in due course without violating the pagemap-allocation policy.

Repeated requests (of either type) by the same page pose no problem, as a request would only be processed upon its original submission and, if refused at that time, will eventually be granted as a side-effect of responding to the events of freeing a pagemap, granting one or permitting the Imaging of some page to start. (One can also verify that it never becomes possible to grant a request due to a pagemap allocation. However, we ignored this and other small refinements in order to retain the simplicity of description and analysis.)

In summary, the complexity of the algorithms (per page) is at worst linear in the number of pages in the system, with small constants and very simple operations. In practice, we expect the complexity to be essentially constant.

V. Conclusions

The decisions to permit out-of-order processing, share object and pagemap memory, and have the intermediate command stream point at cached raster characters by location (not by their full name), give rise to the possibility of deadlock in the Page-Parallel rasterizer architecture. We studied this deadlock problem and provided on-line algorithms for flexibly controlling the rasterizer while avoiding deadlock. The algorithms are executed on-line as the printing progresses. Since the number of events per page is small and the updates at each event are incremental, the computation required for the algorithms is a small fraction of that required for the actual rasterization. It is important to observe that the reliance of our approach on the fact that all ICS for a given page can be consumed once a pagemap has been allocated to that page makes this approach inapplicable to intra-page parallel rasterization with out-of-order generation of ICS within a page.

The issues addressed here should not be confused with scheduling. Rather, our algorithms can be viewed as an oracle that tells the scheduler whether or not a requested phase for a given page may commence. The algorithms give the scheduler the greatest possible freedom while avoiding deadlock.

One of the resources which were part of the deadlock cycle combines features of database records, which are requested specifically but can be read concurrently, and memory which is requested by quantity. This, along with the lack of advance knowledge of requirements for this resource and the permission to share it in a certain way, created special problems. We showed how to solve these without even trying to directly control the use of the problematic resource. Instead, we were able to guarantee that all tasks using this resource would be able to release it in return for a better-behaved resource. We believe that this indirect approach extends to other processes with additional resources or a longer release-chain, and should probably be considered before giving up on deadlock avoidance and resorting to prevention or recovery.

Acknowledgment

S. Scott observed that once a pagemap is allocated to a page its ICS can be consumed. S. Scott, W. Plouffe, A. Strietzel, D. Ehlers, J. Lotspiech, and S. Nin participated in various discussions on this topic. S. Nin and J. Stamos reviewed early versions of this manuscript.

[1] J. L. Peterson and A. Silberschatz, *Operating System Concepts*, 2nd ed. Reading, MA: Addison-Wesley, 1986.
[2] E. W. Dijkstra, "Cooperating sequential processes," Tech. Rep. EWD-123, Technological Univ., Eindhoven, The Netherlands, 1965; reprinted in *Gen68*, pp. 43–112.
[3] F. Genuys, Ed., *Programming Languages*. London, England: Academic, 1968.
[4] A. N. Habermann, "Prevention of system deadlocks," *Commun. ACM.*, vol. 12, pp. 373–377, 1969.
[5] R. C. Holt, "On deadlock in computer systems," Ph.D. dissertation, Cornell Univ., 1971. (Also CSRG Tech. Rep. 6, CSD, Univ. Toronto.)
[6] ——, "Some deadlock properties of computer systems," *Comput. Surveys*, vol. 4, pp. 179–196, 1972.
[7] D. B. Lomet, "Subsystems of processes with deadlock avoidance," *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 297–304, 1980.
[8] T. Minoura, "Deadlock avoidance revisited," *J. ACM*, vol. 29, no. 4, pp. 1024–1048, 1982.
[9] S. S. Isloor and T. A. Marsland, "The deadlock problem: An overview," *IEEE Comput. Mag.*, vol. 13, pp. 58–78, 1980.
[10] K. C. Sevcik, "Comparison of concurrency control methods using analytical models," *Inform. Processing*, pp. 847–858, 1983, R. E. A. Mason, Ed.
[11] M. C. Chen and M. Rem, "Deadlock-freedom in resource contentions," *Acta Informatica*, vol. 21, pp. 585–598, Springer-Verlag, 1985.
[12] D. Zöbel. "The deadlock problem: A classifying bibliography," *Oper. Syst. Rev.*, vol. 17, no. 4, pp. 6–15, 1983.

Yitzhak Birk (S'82–M'86) received the B.Sc. (cum laude) and M.Sc. degrees from the Technion–Israel Institute of Technology, Haifa, in 1975 and 1982, respectively, and a Ph.D. degree from Stanford University in 1987, all in electrical engineering.

From 1976 to 1981, he was project engineer in the Israel Defense Forces. From 1986 to 1991, he was with IBM at the Almaden Research Center, where he worked on parallel architectures, computer subsystems and passive fiber-optic interconnection networks. He is presently on the faculty of the Electrical Engineering Department at the Technion. His current research interests include computer subsystems, communication networks, distributed systems, and multiprocessor architectures.