A Speculative Approach to Deadlock Handling

Ankur Sahai
University of Southern California
Los Angeles, CA 90089
asahai@usc.edu

ABSTRACT

This paper summarizes a detailed survey of deadlock handling approaches and suggests an ingenious approach to handle deadlocks. This approach is based upon storing the access patterns of the resources in a system (consisting of processes and resources and their abstract relations) and using these to make predictions about the future behavior of the system, in particular, to avoid imminent deadlocks. The author also investigates the tools and techniques that can be used to implement this approach. Further, a suggestion is made to use knowledge-bases in place of the traditional resource wait-for-graphs to handle the deadlocks.

Keywords:
Deadlock, Bayesian inference, Knowledge bases, Virtual edges, Virtual cycles,

Outline

1. Introduction

1.1 *What is a deadlock?*

A deadlock (or *deadly embrace*) is said to have occurred when a process ends up waiting indefinitely (or *starving*) for its completion because, it waits for another process in the system to release a requested resource; at the same time, holding a resource requested by that process. This can occur among more than two processes in a system where the processes are waiting for each other (to release a requested resource) in a cyclic fashion, as in Fig 1(b), thus forming a *deadlock cycle*. Hence, none of these processes can proceed until one of them is explicitly terminated.

The resource *wait-for-graphs* (WFG)[18], defined in section 2.4, are used to depict and detect the deadlocks in the system.

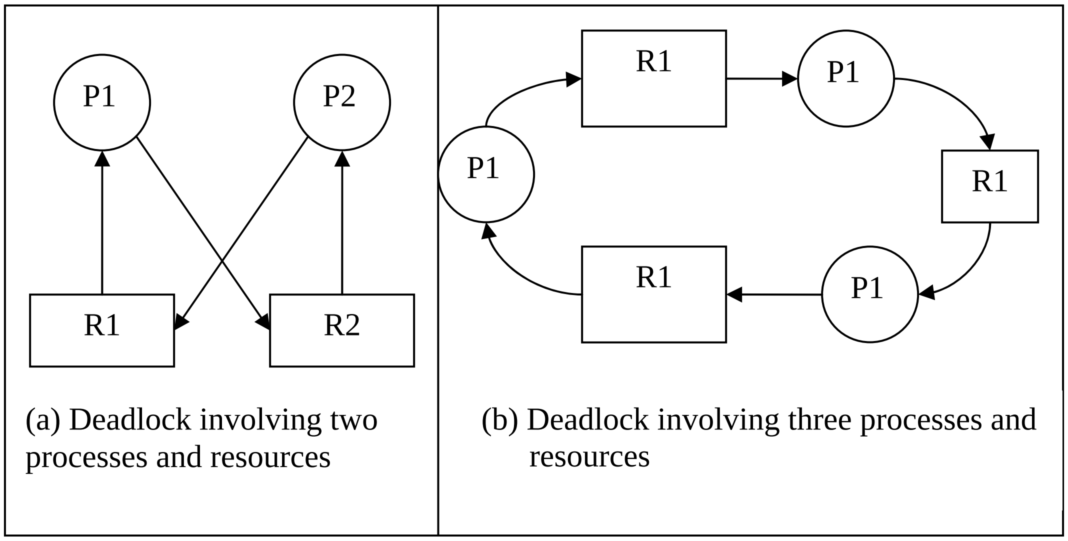
P1
P2
R1
R2
(a) Deadlock involving two processes and resources
R1
P1
P1
R1
R1
P1
(b) Deadlock involving three processes and resources

Fig.1 Resource-wait-for-graphs depicting deadlocks in the system involving (a) two and (b) three processes.

The necessary conditions for a deadlock to occur in a system were identified by E.G. Coffman[17]. They are as follows:

- Mutual Exclusion: a resource can be accessed by only one process at a time
- Hold and wait condition: a process that already holds a resource may request for another resource
- Non-preemption: a process holding a resource cannot be forced to drop a resource and only the process itself may release it
- Circular Wait: two or more processes wait for a resource that the next process in a cycle holds thus forming a circular chain (or *deadlock cycle*)

The reader is also encouraged to go through this paper by Levine[13] that attempts to define deadlocks comprehensively.

1.2 Distributed Deadlock

Distributed deadlock is a deadlock that occurs in a distributed system wherein, the processes and the requested resources involved in the deadlock may be spread across the distributed system instead of being present in the same local node. They may even be several miles apart. This paper will, for the most part, refer to the issue of distributed systems and distributed deadlocks as systems and deadlocks respectively.

There are different ways to handle such a deadlock depending on the structure of the distributed system itself. This paper will discuss *distributed, hierarchical, cluster based* and *centralized* deadlock handling approaches as these are the most common types of existing system organizations and have been analyzed in extensive research.

1.3 Relevance of distributed deadlocks

Typically, deadlocks occur in systems which can be broken down into abstractions of the type: processes that acquire resources with exclusive access rights. Processes here may range from light-weight processes (*threads*) running within a node to processes running at different nodes, that may be miles apart, in a distributed system. And resources may be: memory (*critical section*), CPU, input/output devices, files, other processes, buffers etc. Deadlocks degrade the performance of the system.

Deadlock has been widely studied in many fields of computer science, notably in communications, database, and operating systems. The issue of deadlocks has become even more relevant with the growing size of these systems. Deadlocks also show up in cellular robotics networks, sensor networks and wormhole routing networks

1.4 Deadlock handling

The deadlock handling approaches can be classified into the following categories.

1.4.1 Prevention

Deadlock prevention technique *serializes* the execution of the processes so that, deadlocks do not occur in the system. This requires prior knowledge of all the resource requests of the processes in the system. It overlooks the dynamic behavior of the system wherein the processes and resources may enter and leave the system arbitrarily

1.4.2 Avoidance

Deadlock avoidance involves checking the system to see if it will end up in a *safe state* if, a resource request is granted. So, only such a set of requests are granted that do not put the system in an *unsafe state*. However, this is very expensive because it requires keeping track of the state information of the system which includes number of instances of each resource and which processes hold and request which resources. It also makes implicit assumptions about the system like the maximum number of resources a process may acquire thus, ignoring the dynamic behavior.

Classification of deadlock handling approaches into prevention and avoidance is an arguable topic. There have been papers[4] attempting to collapse the boundary between these two approaches.

1.4.3 Detection and Recovery

Detection is the most commonly employed approach that eliminates the overhead involved in the prevention and avoidance techniques. It detects a deadlock only when one actually occurs in the system. Note that, it is a more pragmatic approach as; it doesn't make any assumptions about the state of the system.

Once a deadlock has been detected, a recovery mechanism is executed to bring the system to a deadlock-free state. This essentially requires forcing one or more processes to relinquish their resources; otherwise killing them to break the deadlock.

1.4.4 Drawbacks of current approaches

As discussed in the previous sections, prevention is impractical in most cases as, it requires prior knowledge of the system state which is not possible for most distributed systems because of their dynamic behavior.

On the other hand, Avoidance is very costly apart from the fact that it makes some assumptions about the system state.

In case of detection, we have to explicitly drop the resource requests or kill the processes in midst of their execution which is not the best approach. Further, the problem of detecting deadlocks accurately in a system has been proven to be NP-Complete. Moreover, we could overcome this by detecting the deadlocks before they actually occur in the system, at least the most obvious ones, as proposed by the *Speculative approach* proposed by the author in this paper.

1.4.5 A Speculative approach to handling deadlocks

In this approach proposed by the author the deadlocks will be detected before they actually occur in the system; at least the most obvious ones i.e. the ones that have the highest Bayesian probability of occurring given their previous occurrences as the priori. As a result, it bypasses the effort involved in subsequent detection and recovery of those deadlocks. However, ideally this approach will be most effective when run on top of deadlock detection and recovery mechanisms, so that, it can be tuned for the optimal throughput making it a *throughput-oriented approach*.

This is different from both prevention and avoidance techniques as, it neither requires any prior information of the system state; nor does it check for the *safe state*.

2. A survey of deadlock handling techniques

This section will study deadlocks in different fields and at different levels of granularity starting from the *critical section problem* up to large-scale distributed systems and effective techniques to tackle them. *Process synchronization* techniques are closely related to broader class of *prevention techniques,* although, the motivation here is to avoid data inconsistency by serializing the execution of the processes and sharing of resource is allowed here. The *critical section* problem and *monitors* are prime examples of places where *deadlocks* and *race conditions* may occur within a system because exclusive access to the codes is required. Elegant algorithms have been proposed to tackle these problems and are discussed in the following paragraphs.

The *Ostrich algorithm* is a design strategy that advocates overlooking deadlocks if they are known to occur rarely and is used by most modern operating systems like UNIX and Microsoft Windows. It is similar to the *Great Big Lock* algorithm used for the protection of critical sections which uses a lock that prevents anything from occurring at the same time as anything protected. These are lazy and inefficient design strategies.

- ***Deadlock prevention and avoidance* techniques:**

One or more of the necessary conditions must be negated to prevent or avoid deadlocks. Concurrent programming problems and solutions were proposed by Dijkstra[6] and later on improved by Lamport[7]. *Critical section* is that part of the code that can be accessed by only one process at a time. The working of *Bakery algorithm*[7] can be better understood by drawing analogy to a bakery (critical section) with customers (threads) waiting in a queue to be served. A machine at the entrance of the bakery assigns to each customer a coupon before he enters. The number on this coupon is incremented by one every time a customer enters the bakery and a global counter displays the coupon number of the customer being served to all customers. Customers wait in a queue until the baker finishes serving the current customer and the next number is displayed on the global counter. Here the number on the coupon can be envisaged as the priority associated with the corresponding thread; a lower value indicating a higher priority and vice versa.

The *One-shot* algorithm[8] is a trivial deadlock prevention strategy that requires each process to request all its resources simultaneously and then allocates all these resources to the process before it begins execution. *Hierarchical* algorithm[9] a slightly more complex strategy that refuses the resource request by the process either if the resource requested is being held by another process or if any of the resources being held by the requesting process is of higher priority than the one requested. *Repeated One-shot* algorithm and *Hierarchical algorithm with waiting* are tweaks to these techniques meant for more effective implementations. Dijkstra's Banker's Algorithm discussed in the next paragraph is also used for deadlock prevention.

Banker's algorithm used for deadlock prevention and avoidance is also a largely theoretical approach requiring the prior knowledge of resource usage limit. The algorithm works by continuously trying to keep the system in a *safe state*. A system is in a *safe-state* if there exists a *safe sequence:* which is an ordering on the processes <P1,...,Pn> such that, for any process Pi in the ordering, its resource requests can be satisfied with the resources that are currently available in addition to the resources that are held by Pj, where j < i. If no such sequence exists, then the system is said to be in an *unsafe state* which hints to but does not necessarily imply a deadlock.

Lee et al[12] talk about a deadlock avoidance algorithm (DDA) involving explicit *Deadlock Avoidance Unit* (DAU) that keeps track of the system state and communicates with the processes to avoid deadlocks. Iordache et al[13] talk about deadlock prevention *supervisors* using *Petri-nets* that are mathematical representations of a discrete distributed system; graphically depicting their structure as directed bipartite graphs with annotations. Andrews et al[14] talk about *on-the-fly* deadlock prevention using *Communicating Sequential Processes*[15] to serialize the processes. *Strict two-phase locking[16]* is yet another deadlock avoidance technique used in *database systems* to serialize the transactions that relies on two principles: If a transaction T wants to read or write a resource, it must request a shared or exclusive lock on the resource respectively. All locks held by transaction T are released only when T commits.

- ***Deadlock detection***:

Deadlock detection involves detecting the deadlocks that have already occurred in the system. To detect deadlocks in a distributed system, it is necessary to know the global system state. A Global *Resource wait-for-graph (RWFG)* is the most commonly used construct for this. It is necessary to understand the notion of resource-wait-for-graphs first; the following paragraph defines it. Majority of the currently used techniques use the RWFGs to detect deadlocks by checking for *cycles* in this graph. This paper will use the term wait-for-graphs (or WFGs) to refer to the resource-wait for graphs.

A Resource-wait-for-graph (shown in Fig.1.) is defined as a directional graph G= (V, E) which depicts the entire state of the system, where,

V ← set of nodes representing processes or resources; the nodes indicating the processes are represented by circles and resources by rectangles

E ← set of directed edges representing whether the resource is being held or requested by a process. If the edge is directed towards a process node then the resource pointed to by the tail of the edge is being held by the process pointed to by the head of the same edge. Whereas, if the edge points to a resource node then the resource pointed to by the head of the edge is being requested to by the process pointed to by the tail of the same edge.

Distributed system are classified into AND, OR and AND-OR[19] models in the context of deadlocks. In the AND model (or *multiple-resource model*), a process is allowed to make more than one resource request, and it is blocked until all of these requests are granted. So, the processes can be involved in several deadlock cycles at once. In the OR model (*or communication model*), a process makes more than one resource requests and is blocked until any one of them is granted. The AND-OR hybrid model allows a combination of these request types, such as a request for a resource X and either Y or Z. This paper only considers AND model; it being the more natural one.

- ***Deadlock detection* techniques:**

  a. *Distributed deadlock detection*
     a.1 Obermarck's Path-pushing
     a.2 Chandy, Misra and Haas's Edge Chasing

  b. *Hierarchical deadlock detection*
     b.1 Menasce and Muntz's algorithm
     b.2 Ho and Ramamoorthy's algorithm

Singhal[20] has studied these categories in detail. Hierarchical algorithms are best suited for systems where the nodes are organized in a tree structure as compared to distributed ones which envisage nodes to be spread randomly across the system.

Distributed deadlock detection techniques are generally classified as *Path-pushing* and *Edge-chasing* techniques. In the *Path-pushing* techniques the WFG is disseminated into paths that are sequence of edges. Since, these paths are disseminated across the distributed system, a deadlock is declared if a node detects a local cycle (a closed sequence of these paths).

Edge chasing is a more elegant and widely used efficient technique. It is closer to the *end-to-end arguments principle*[1] as it requires lesser functionality within each node compared to the *path-pushing* techniques that have the additional overhead of managing the paths apart from detecting local cycles. Here, a signal called *probe* is generated at a particular node initiated by a process which wants to detect global deadlocks in the system. The *probe* is sent from one node to another if; a process running at one node holds a resource physically present at another node. This *probe* when received by the other node is circulated within its local WFG and is sent out to another node if the probe hits upon a local process which hold a resource physically present at some other node and so on. This is repeated at each node of the system that receives the *probe*. If the initiator process successfully receives the probe generated by it, a deadlock is declared in the system. Note that, the *probe* travels only along one direction and dies off if it cannot proceed. As discussed in section 1.3.5, it is not a complete approach since; accurate detection of deadlocks in a system is NP-Complete.

Let us examine the specific algorithms more closely. In *Obermarck's Path-Pushing algorithm[21],* each site maintains local WFGs, the process nodes of which are classified into the nodes for local processes and "Pex" node representing external processes. For example, a path at the local node may look like: Pex1 → P1 → P2 → P3 → Pex2. Now, if a node Ni detects a cycle without a "Pex" node it declares a local deadlock. If the cycle has a "Pex" node then there is a possibility of global deadlock. To check this possibility of deadlock, node Ni sends a message containing its detected path to all other sites preferably only when Pex1>Pex2 to decrease the network traffic. If a site Nj receives such a message, it updates its local WFG and reevaluates it possibly by pushing a path again. The two drawbacks of this approach are that it assumes a unique global identifier for a process and may report false deadlocks.

The Chandy, Misra and Hass's Edge-chasing algorithm[23] works as follows. When a process has to wait (or block) for a resource, it sends a *probe* to the process holding that resource. Here, processes are allowed to request and wait for multiple resources simultaneously. *Probe* contains 3 values: ID of the process that blocked on the resource, ID of process sending the *probe* and ID of process the *probe* was sent to, though; the latter two identifiers seem obsolete here. When a blocked process receives this *probe*, it propagates the probe to the processes holding the resources that it has requested but not before updating the latter two fields of the *probe.* If the blocked process receives its own probe, it declares a deadlock.

Performance evaluations of these algorithms have been carried out earlier[40, 42,43] and the following facts have been determined. In case of Obermarck's *Path-pushing* algorithm, only half of the sites involved in a deadlock send *probes* on average. Every blocked site sends messages to all other sites, thus $O(n^2)$ messages are required to detect a deadlock. For n sites, the size of this message is $O(n)$. In case of Chandy, Misra, and Haas's algorithm, given n processes in a system, a process holding a resource may be blocked by up to $n-1$ processes, the next process may be blocked by another $n-2$ processes and so on. So, if there are more number of nodes than processes, in the worst case, the number of messages is $O(n^2)$; size of each *probe* message being 3 integers.

Menasce and Muntz' *hierarchical* deadlock detection algorithm[24] that assumes a hierarchical tree-structured organization of the nodes (or *controllers*) in a distributed system works as follows. *Leaf controllers* manage the resources by maintaining a local WFG whose resource nodes represent only its own resources. *Interior controllers* are responsible for detecting deadlocks in the WFG which is constructed from the union of the WFGs of its children. The changes in the WFGs are propagated upwards continuously or periodically.

In Ho and Ramamoorthy's *hierarchical* deadlock detection algorithm[25] the sites are grouped into disjoint *clusters*. Periodically, a site is chosen as a *central control site* which chooses a *control site* for each *cluster* like a soft-state. This *control site* then collects status information from the nodes in its *cluster* and tries to detect deadlocks in that cluster. All the *control sites* then forward their status information and WFGs to the *central control site*, which combines this information into a *global WFG* and searches it for cycles. Thus, *control sites* detect deadlocks within *clusters* whereas, *central control site* detects deadlock between *clusters*

Several techniques have been proposed that try to improve upon these drawbacks. Lee et al[26] propose a parallel detection algorithm using deadlock detection units(DDU) having time complexity of $O(min(m, n))$, where m is the number of resources and n is the number of processes in the system. They have also proposed

Operating System designs based on this approach achieving significant speed up[27]. This is a significant improvement over the algorithm proposed by Kim[28] with time complexity of *O(mn)*. Bracha et al[29] came up with an efficient detection algorithm for AND-OR models in the mid 90s. Huang[30] proposed a detection algorithm for CSP-like communication that overcomes most of these drawbacks. *Self-stabilizing* deadlock detection algorithms were first proposed by Flatebo et al[31]. Martinez et al bring up the interesting topic of effect of buffer sizes[32] and propose an efficient detection algorithm for *wormhole networks*[33]. Kshemkalyani et al[34] noted flaws in the earlier proof techniques of detection algorithms and proposed *Invariant-based verification technique*. Mendivil et al[35] propose a *Syntactic approach* using Automata theory with *wait-strings*. Lee[36] proposed an efficient algorithm for centralized detection and resolution. Krivokapic et al[37] proposed a detection algorithm based on dynamic *deadlock detection agents* (DDA). Farajzadeh et al[38,39] propose a *history-based edge chasing* algorithm that resolves the deadlock as soon as detects it without waiting for the probe to return back; thus reducing the average persistence time of the deadlock.

Estimating performance of a deadlock detection algorithm requires the following facts to be accounted for. It is usually measured as the *number of messages* exchanged to detect deadlocks. This is deceptive since; messages are also exchanged when there are no deadlocks. Size of the messages, being insignificant is not accounted for. It should also measure the *deadlock persistence time*; a measure of *resources wastage* that has a tradeoff with the *communication overhead*. *Storage overhead* of WFGs and other deadlock detection constructs has to be accounted for. *Processing overhead* to search for cycles and *time to recover* from deadlocks also has to be accounted for. The reader is encouraged to go through this performance analysis study by Lee et al[40].

- ***Deadlock resolution:***

As discussed earlier in the paper, deadlock resolution involves aborting at least one process called *victim* in the cycle and granting its resources to others. There are some standard efficiency issues with deadlock resolution. It should be *fast* i.e. after deadlock is detected the victim should be selected quickly. It should be *minimal* in the sense that it should abort minimum number of processes and ideally abort *least expensive* ones, with respect to: completed computation, consumed resources, etc. It should be *complete* i.e. after a victim has been aborted the information about it should be quickly removed from the system to avoid *phantom deadlocks. Phantom deadlocks* are those deadlocks that have been already resolved by the time they are detected. This may either be due to *delay in transmission* of system information or failure to update the system state. It should avoid *starvation* of a process which may occur due to repeated selection of the same process as the victim and its subsequent abortion.

The following are the main problems that show up during resolution of deadlocks. Detecting processes (on local nodes) may not have enough information about the victim; on the other hand propagating sufficient information makes detection expensive. Multiple nodes may simultaneously detect a global deadlock. Since the global state information is distributed across the system, removing information about the victim and updating the global information takes time. Finally, *livelocks*[41] can be a menace where the state of processes involved in deadlock keeps changing with respect to each other.

3. A Speculative approach to Deadlock Handling

3.1 Introduction

One can maintain salient patterns of the behavioral history of the system by taking *distributed snapshots*[2] of the system at regular intervals. This information can be effectively used to make decisions (using Bayesian inference) on the allocation of resources to the processes in the system to avoid impending deadlocks. This technique should be implemented ideally on top of existing detection mechanisms to be most effective. Extent of usage of this technique will be guided by the throughput (due to deadlocks) of the system making it a *throughput-oriented approach*.

3.2 Outline of the algorithm

Knowledge of the access patterns of the processes is maintained and used effectively by the *prediction tools* running in the background. Using this information, new edges called *virtual edges* are added to the WFG speculatively. Note that these are *virtual edges* and are not considered while declaring deadlocks in the system by the actual detection algorithm. The WFG is built upon speculatively by adding virtual edges.

The goal now is to detect cycles containing one or more *virtual edges* called *virtual cycles* in the Global WFG which is left to an efficient detection algorithm. Note that, this is in addition to the detection and resolution of actual deadlocks in the system. Once a *virtual deadlock cycle* is detected, necessary steps are taken to avoid this cycle from occurring in real-time in the future. This can be achieved by denying (or delaying) those resources to the processes corresponding to the virtual edges the virtual cycles.

3.3 Prediction tools

Prediction tools are used to decide when and which virtual edges to create by maintaining the information about the access patterns efficiently. The author will analyze different tools that can be used for this purpose and.

3.3.1 Using data–structures in the memory

In this section, the author suggests data-structures to store this information and doesn’t delve into their implementation details. To cope up with the dynamic behavior of the system, linked-lists that grow and shrink can be used to implement structures like priority queues. Trees can be more suitable for a hierarchical organization. These data-structures should store the *process id*, the corresponding *resource ids* each having a *status bit* (held/blocked) and an associated *priority value* that is a measure of the priori probability. More information can be stored to better the predictions like the type of resources most used by the process i.e. whether it is CPU or I/O intensive etc.

Since there will be an upper bound to the available memory space, *refreshing algorithms* have to be used to make the optimal use of this space. The algorithm suggested here will refresh the memory at fixed time intervals when a *snapshot* is made. While updating the information, if the algorithm comes across an already existent entry (process-resource pair representing an edge in the WFG) it increases the associated priority value instead of selecting a victim to replace which it will do otherwise, when there is no free space left. This algorithm selects an entry with lowest priority value corresponding to the edge in WFG when a new resource comes to be held or blocked by an existent process. In the case of new process, the algorithm evicts the process with the lowest cumulative value of priorities of the resources associated with it. The worst case space complexity is $O(p*r*c1*c2)$~$O(p*r)$ where, p is number of processes, r is number of resources, $c1$ is for flag bit and $c2$ is for the priority value.

3.3.2 Machine learning

Machine learning techniques, can be used to learn and predict the behavior of the system over time. Especially those that try to avoid the need for human intuition in the analysis of the data will be more useful because, it is not possible for a human to monitor this data. However, they need sufficient time for training and are quite costly.

3.3.3 Knowledge Bases

The author suggests the use of Knowledge bases not only as detection tools but also as prediction tools. Knowledge bases provide means for efficient storage, organization and retrieval of information. They can be used by exploiting their logical deductive reasoning. One can use the system information stored as atomic formulae to build a compound formula, using logical operators, to detect deadlocks like: if *(((a process p1 is holding a resource r1) and (is requesting for a resource r2)) and ((a process p2 is holding r2) and (requesting resource r1)))*, declare a deadlock.

3.4 Throughput-oriented iterative improvement algorithm using prediction classes

The deadlock predictions can be classified into different classes based on their *strength*. The author suggests two methods to decide the *strength* of a prediction. One method is to take into account only the number of virtual edges involved in the virtual cycles with the cycle with lesser number of virtual edges having higher strength and vice-versa. It is suitable for more dynamic systems where the priori probabilities associated with virtual edges are less dependable because, they are not accounted for by this method. The other method would be to consider the priorities (priori probabilities) associated with the virtual edges involved, for example, by multiplying them. This method will be ideal for more deterministic systems where the priori probabilities are more dependable as it uses them to decide the strength of the predictions. The predictions with same strength are put under the same prediction class.

The idea here is to use these prediction classes to maximize the throughput of the system (countering deadlocks). Measuring the throughput countering deadlocks is a tricky step that can be done using heuristics, although, the ideal way of measuring it would be by making all other factors invariant (or nullifying them). A *throughput-oriented iterative-improvement algorithm* is proposed where first, only the highest class of predictions is applied to the system and the resultant throughput is compared to throughput of the system without using this technique. If there is an improvement in the throughput of the system the next higher class of predictions is considered; this procedure is repeated until there is a prediction class that lowers the previous throughput. Thus in the worst case, the system uses none of these predictions and in the best case it uses all of them. Note that, this approach assumes that the behavior of the system is not totally arbitrary.

4. Summary and Conclusions

This paper surveys deadlocks and deadlock handling techniques: prevention, avoidance and detection (and recovery). It studies the issue of deadlocks at different levels of granularity. The author analyses some standard deadlock handling techniques and brings up their merits and demerits. The conclusion is that the deadlock prevention and avoidance approaches are impractical because they do not take into account the dynamic behavior of the system and require some prior knowledge of the system. The biggest demerit of the currently used deadlock detection and resolution mechanisms, however efficient they may seem, is that they can only detect and resolve deadlocks that have already occurred in the system.

The author proposes a novel Speculative approach to deadlock handling that will be most effective when run on top of the deadlock detection and recovery mechanisms, so that, it can be tuned for the optimal throughput making it a throughput-oriented approach. An iterative improvement algorithm based on *prediction classes* is also presented to achieve this. It is based on the crucial assumption that there are some general access patterns in the system that can be captured by taking periodic *snapshots* of the system. This information can be analyzed to make resource allocation decisions to avoid imminent deadlocks thus overcoming the overhead of having to detect and resolve these deadlocks subsequently. The author also examines tools for making these predictions. It improves upon the detection mechanisms by using Bayesian inferences based on the previous system information (as priori).

Use of Knowledge bases is suggested as a construct for detection and further prediction of deadlocks as an alternative to the WFGs.

5. Scope for future work

More efficient criteria and tools that account for the higher level details of system may be thought of to classify and use the predictions. Efficient data-structures and algorithms to improve the overall performance of this technique can be explored. Further, use of Knowledge bases for detection and prediction can be analyzed.

References

1. J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end Arguments In System Design ", *ACM Transactions on Computer Systems*, ACM Press, vol. 2 , no. 4 , 1984 , pp.277-288

2. K. Mani Chandy and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, vol. 3, no. 1, February 1985, pp. 63-75.

3. G.N. Levine, Defining deadlock, ACM SIGOPS Operating Systems Review Volume 37 , Issue 1 (January 2003), Pages: 54 – 64, Year of Publication: 2003 ISSN:0163-5980, ACM Press

4. G.N. Levine, The classification of deadlock prevention and avoidance is erroneous, ACM SIGOPS Operating Systems Review**,** Volume 39, Issue 2 (April 2005) , Pages: 47 - 50 , Year of Publication: 2005, ISSN:0163-5980, ACM Press

5. Davidson and MacKinon, Ostrich Algorithm, 1993

6. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (Sept. 1965), 569.

7. Lamport, L., A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17,* 8 {Aug. 1974), 453-455.

8. Baer J.L., Russel E.C, Modelling and scheduling of computer programs for parallel processing systems, Winter Simulation Conference, Proceedings of the second conference on Applications of simulations**,** New York, New York, United States, Pages: 278 – 281, Year of Publication: 1968, Winter Simulation Conference

9. Harder H., Rothermel K., Concurrency control issues in nested transactions, The VLDB Journal — The International Journal on Very Large Data Bases**,** Volume 2 , Issue 1 (January 1993) Pages: 39 – 74, Year of Publication: 1993 ISSN:1066-8888, Springer-Verlag New York, Inc

10. Dijkstra, E.W. Cooperating sequential processes. In *Programming Languages,* F. Genuys, Ed., Academic Press, New York, 1968, 43-112.

11. Brinch Hansen, P. *Operating System Principles. P*rentice-Hall, Englewood Cliffs, N J, 1973.

12. Jaehwan Lee, Vincent John Mooney, Software and hardware techniques for performance optimization of embedded applications: A novel deadlock avoidance algorithm and its hardware implementation

13. Iordache, M.V.; Moody, J.; Antsaklis, P.J.; Synthesis of deadlock prevention supervisors using Petri nets; Robotics and Automation, IEEE Transactions on, Volume 18, Issue 1, Feb. 2002 Page(s):59 - 68 , Digital Object Identifier 10.1109/70.988975

14. Gregory R. Andrews, Gary M. Levin, On-the-fly deadlock prevention, August 1982, Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing, ACM Press

15. C. A. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, vol. 21, no. 8, August 1978, pp. 666-677

16. K. P. Eswaran, J.Gray, R. Lorie, and I. Traiger, "TheNotion of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, vol. 19, pp. 624–633, 1976.

17. Coffman E.G., Elphick M., Shoshani A., System Deadlocks, ACM Computing Surveys (CSUR) Volume 3 , Issue 2 (June 1971), Pages: 67 - 78 , Year of Publication: 1971, ISSN:0360-0300, ACM Press

18. Cheng J., Task-wait-for graphs and their application to handling tasking deadlocks, Annual International Conference on Ada**,** Proceedings of the conference on TRI-ADA '90, Baltimore, Maryland, United States, Pages: 376 - 390 , Year of Publication: 1990, ISBN:0-89791-409-0, ACM Press

19. Knapp E., Deadlock detection in distributed databases, ACM Computing Surveys (CSUR)**;** Volume 19 , Issue 4 (December 1987); Pages: 303 – 328; Year of Publication: 1987; ISSN:0360-0300, ACM Press

20. Singhal M (1989) Deadlock detection in distributed systems. IEEE Comput 22(11): 37–48

21. Obermarck R (1982) Distributed deadlock detection algorithm. ACM Trans Database Syst 7(2): 187–208

22 C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

23. Chandy K. M., Misra , J., and Haas, L. M. 1983. Distributed deadlock detection. ACM Trans. Comput. Syst. 1,2 (May), 144-156.

24. Menasce, D. and Muntz, R. 1979. Locking and deadlock detection in distributed databases. IEEE Trans. Softw. Eng. SE-5,3 (May).

25. Ho, G. S., AND Ramamoorthy, C. V. 1982. Protocols for deadlock detection in distributed database systems. IEEE Trans. Softw. Eng. SE- 8, 6 (Nov.), 554-557.

26. Jaehwan John Lee, Vincent John Mooney, An *o*(*min*(*m*, *n*)) parallel deadlock detection algorithm; July 2005, ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 10 Issue 3, ACM Press

27. Lee, J.J.; Mooney, V.J., III, Hardware/software partitioning of operating systems: focus on deadlock detection and avoidance, Computers and Digital Techniques, IEE Proceedings-Volume 152, Issue 2, Mar 2005 Page(s):167 - 182 Digital Object Identifier 10.1049/ip-cdt:20045078

28. Ju Gyun Kim; An algorithmic approach on deadlock detection for enhanced parallelism in multiprocessing systems; Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium; 17-21 March 1997; Page(s):233 - 238 ; Digital Object Identifier 10.1109/AISPAS.1997.581669

29. Gabriel Bracha, Sam Toueg, A distributed algorithm for generalized deadlock detection; August 1984, Proceedings of the third annual ACM symposium on Principles of distributed computing, ACM Press

30. S.-T. Huang, A distributed deadlock detection algorithm for CSP-like communication, January 1990; ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 12 Issue 1; ACM Press

31. Mitchell Flatebo, Ajoy Kumar Datta, Self-stabilizing deadlock detection algorithms, April 1992, Proceedings of the 1992 ACM annual conference on Communications, ACM Press

32. Martinez, J.M.; Lopez, P.; Duato, J.; Impact of buffer size on the efficiency of deadlock detection, High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On, 9-13 Jan. 1999 Page(s):315 - 318 ; Digital Object Identifier 10.1109/HPCA.1999.744385

33. Lopez, P.; Martinez, J.M.; Duato, J.; A very efficient distributed deadlock detection mechanism for wormhole networks; High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on; 1-4 Feb. 1998 Page(s):57 - 66
Digital Object Identifier 10.1109/HPCA.1998.650546

34. Kshemkalyani, A.D.; Singhal, M.; Invariant-based verification of a distributed deadlock detection algorithm; Software Engineering, IEEE Transactions on Volume 17, Issue 8, Aug. 1991 Page(s):789 - 799 ;Digital Object Identifier 10.1109/32.83914

35. Gonzalez de Mendivil, J.R.; Garitagoitia, J.R.; Syntactic approach to the deadlock detection problem; CompEuro '92 . 'Computer Systems and Software Engineering', Proceedings. 4-8 May 1992 Page(s):515 - 519 Digital Object Identifier 10.1109/CMPEUR.1992.218430

36. Soojung Lee; Fast, centralized detection and resolution of distributed deadlocks in the generalized model; Software Engineering, IEEE Transactions on Volume 30, Issue 9, Sept. 2004 Page(s):561 - 573 Digital Object Identifier 10.1109/TSE.2004.51

37. Natalija Krivokapić, Alfons Kemper, Ehud Gudes; Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis; October 1999; The VLDB Journal — The International Journal on Very Large Data Bases, Volume 8 Issue; Springer-Verlag New York, Inc

38. Farajzadeh, N.; Hashemzadeh, M.; Mousakhani, M.; Haghighat, A.T.; An Efficient Generalized Deadlock Detection and Resolution Algorithm in Distributed Systems; Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on 21-23 Sept. 2005 Page(s):303 - 309 Digital Object Identifier 10.1109/CIT.2005.69

39. Hashemzadeh, M.; Farajzadeh, N.; Haghighat, A.T.; Optimal Detection and Resolution of Distributed Deadlocks in the Generalized Model; Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on 15-17 Feb. 2006 Page(s):133 - 136 Digital Object Identifier 10.1109/PDP.2006.54

40. Soojung Lee; Kim, J.L.; Performance analysis of distributed deadlock detection algorithms; Knowledge and Data Engineering, IEEE Transactions on Volume 13, Issue 4, July-Aug. 2001 Page(s):623 - 636 Digital Object Identifier 10.1109/69.940736

41. [Lo 80] D.B. Lomet / Subsystems of Processes with Deadlock Avoidance / IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, 297-304

42. O. Bukhres, Performance Comparison of Distributed Deadlock Detection Algorithms, Proc. IEEE Eighth Int'l Conf. Data Eng.,pp. 210-217, 1992.

43. A.N. Choudhary, Cost of Distributed Deadlock Detection: A Performance Study, Proc. Sixth Int'l Conf. Data Eng., pp. 174-181, Feb. 1990.

44. B I. Galler and L. Bos, A Model of Transaction Blocking in Databases, Performance Evaluation, vol. 3, pp. 95-122, 1983.

45. K. Min, Performance Study of Distributed Deadlock Detection Algorithms for Distributed Database Systems, PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1990.

46. K.H. Pun and G.G. Belford, Performance Study of Two Phase Locking in Single Site Database Systems, IEEE Trans. Software Eng., vol. 13, no. 12, pp. 1311-1328, Dec. 1987.

47. S.-C. Shyu and V.O.K. Li, Performance Analysis of Static Locking in Distributed Database Systems, IEEE Trans. Computers, vol. 39, pp. 741-751, June 1990.

48. George Coulouris, Jean Dollimore, and Tim Kindberg; Distributed Systems : Concepts and Design (3th edition); Publishers: Addison Wesley

49. Rakesh Agrawal, Michael J. Carey, David J. DeWitt; Deadlock detection is cheap; January 1983; ACM SIGMOD Record, Volume 13 Issue 2; ACM Press

50. Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne; Operating System Concepts (6th edition)