Parallel Scheduling in Data Base Systems

Dynamic Action Scheduling in a Parallel Database System

Paul W.P.J. Grefen   Peter M.G. Apers
University of Twente, The Netherlands

Abstract

This paper describes a scheduling technique for parallel database systems to obtain high performance, both in terms of response time and throughput. The technique enables both intra- and inter-transaction parallelism while controlling concurrency between transactions correctly. Scheduling is performed dynamically at transaction execution time, taking into account dynamic aspects of the execution and allowing parallelism between the scheduling and transaction execution processes. The technique has a solid conceptual background, based on a simple graph-based approach. The usability and effectiveness of the technique are demonstrated by implementation in and measurements on the parallel PRISMA database system.

1 Introduction

In general, (parallel) database systems are meant for high performance data processing. High performance can be seen as a combination of low response times and high throughput. To obtain these properties, a good scheduling of the actions on the database is essential. We distinguish between scheduling of the actions in one single transaction, and scheduling of the actions in different transactions.

The scheduling of actions in one single transaction to obtain low response time is called *transaction optimization* here. The technique is based on dependencies between the actions in a transaction, and takes the dynamic execution characteristics of the transaction into account, using availability of resources and feedback from the execution layer of the system. As such, the scheduling technique is based on a dynamic model of a multi-user machine, and can be seen as the complement of traditional query optimization techniques, which transform actions in a transaction based on a static model of a single-user machine. Transaction optimization is especially important in an environment with complex transactions, either user-defined or system-generated. The latter case occurs in distributed systems with fragmented relations [3] and in systems performing integrity constraint enforcement through transaction modification [6, 7].

The scheduling of actions of different transactions is traditionally called *concurrency control.* This technique tries to optimize throughput under the condition that concurrently executing transactions are 'unaware' of eachother. Concurrency control can be based on dependencies between the involved actions of concurrent transactions, much alike the dependencies mentioned above.

Transaction optimization and concurrency control can be integrated into one single mechanism that operates on a global graph representing the dependencies between actions. This leads to a system architecture with fully centralized action scheduling. Decentralizing this task can now be modelled conceptually by splitting up the graph into several subgraphs, operated upon by concurrency control and decentralized transaction management processes.

The approach to action scheduling presented in this paper has a number of characteristics that distinguish it from other proposals. In the first place, the approach makes use of high level dependencies between actions; other approaches, like [5], use lower level dependencies representing the actual flow of data in a system. The use of high level dependencies enables simple scheduling mechanisms, causing little overhead and allowing for distribution of and parallelism in the scheduling task. In the second place, the scheduling technique presented in this paper performs the dependency analysis dynamically at transaction execution time, thereby allowing parallelism between transaction control and execution. Compiler-based techniques perform the analysis at transaction definition time [9, 2]. The approach described here requires no compilation of transactions and is therefore suited for ad hoc transactions as well. Finally, the approach described in this paper has been implemented and tested in a real-world parallel database machine. Other conceptually oriented approaches often lack this practical test, whereas most database machine projects pay little attention to scheduling of concurrent multi-action transactions.

This paper starts with a short discussion of some basic notions. Section 3 discusses the concept of order dependency and transaction optimization. Section 4 presents the concept of resource dependency and its use in concurrency control. Order dependency and resource dependency are combined in Section 5 to obtain a global dependency notion. Section 6 discusses architectural issues for an implementation of the techniques. Section 7 describes the practical usage of the techniques in the PRISMA parallel database system, and presents measurements to demonstrate the effectiveness of the proposed techniques. Conclusions are drawn in Section 8.

2 Basic notions

Informally, a transaction is a unit of work executed against a database state. More formally, a transaction T can be seen as an operator that transforms a database state D into another state $T(D)$:

$$D \xrightarrow{T} T(D)$$

Further, the execution of T can have side effects, such as the production of output. The execution of a transaction should satisfy three important properties: *atomicity* of the execution, *serializability* with respect to concurrent transactions, and *correctness* with respect to a set of integrity rules. In the context of this paper, only the serializability property is of interest. This property requires that the effect of the

begin	*begin*	update	*upd*(x, y)
assignment	$x = op(y_1, \cdots, y_n)$	commit	*commit*
output	$?x$		

Table 1: Elementary action types

concurrent execution of two transactions T_1 and T_2 must always be the same as the effect of some serial execution of the same transactions:

$$D \overset{(T_1,T_2)}{\rightarrow} T_1(T_2(D)) \quad or \quad D \overset{(T_1,T_2)}{\rightarrow} T_2(T_1(D))$$

The transformation of the database state and the generation of side effects by a transaction T are specified as a sequence of elementary actions: $T = (a_1; a_2; \cdots; a_n)$. The actions a_i are described using some database language, e.g. a language based on the relational algebra. We make an abstraction from such a language to obtain a small set of elementary action types to specify transactions. The notation of these action types is shown in Table 1. The *begin* action indicates the start of a transaction. The *assigment* action assigns the result of a relational operation to a new and implicitly defined relational variable; the semantics of the operation (join, union etc.) are of no significance in this context. The *output* action delivers the contents a relational variable to the user of the database system; how the output is produced is irrelevant; the output action models side effects of a transaction. The *update* action changes the current state of the database; the first parameter is the variable (relation) to be changed, the second parameter is a variable used as source for the change; the type of update (insert, delete or modify) is of no significance. The *commit* action indicates the end of a transaction. Note, that this set of actions can be used to model more complex actions, such as actions with nested operations. Other choices of elementary action types are possible; the choice is not of great importance for the techniques presented in this paper, however.

3 Order dependency and Transaction optimization

This section starts with the discussion of the dependencies between actions in a transaction with respect to execution order. These dependencies can be represented by means of a graph. This graph is used for the scheduling of actions to obtain an optimized execution of a transaction.

3.1 Order dependency

Actions in a transaction can be dependent with regard to the order in which they have to be executed; this *order dependency* relation is defined below.

Definition 3.1 Given are two transactions $T = (a_1, \cdots, a_i, \cdots, a_j, \cdots, a_n)$ and $T' = (a_1, \cdots, a_j, \cdots, a_i, \cdots, a_n)$. Transaction T' is obtained from T by interchanging actions a_i and a_j. Now action a_j has an *order dependency* with respect to action a_i in

	begin	$v = op(w_1, \cdots, w_m)$	$?v$	$upd(v, w)$	*commit*
begin	-	-	-	-	-
$x = op(y_1, \cdots, y_n)$	*true*	$y_i \equiv v$	*false*	$y_i \equiv v$	-
$?x$	*true*	$x \equiv v$	*true*	$x \equiv v$	-
$upd(x, y)$	*true*	$x \equiv w_i \vee y \equiv v$	$x \equiv v$	$x \equiv v \vee x \equiv w \vee y \equiv v$	-
commit	*true*	*true*	*true*	*true*	-

Table 2: Order dependency matrix

T if at least one of the following two statements holds. **(1)** T and T' model different transformations of at least one database state D: $(\exists D)(T(D) \neq T'(D))$. **(2)** The side effects of the execution of T and T' on some database state D are different; in particular, T and T' either produce different output, or produce the same output in a different order. The fact that a_j is order dependent on a_i is denoted as $od(a_j, a_i)$. □

This definition gives a conceptual view on order dependency, which is hard to use in a practical situation; a more operational approach is developed in the sequel of this paper. If the order dependency relations between actions of a transaction are to be analyzed, a minimal set of relations is preferable. Therefore, the definition above is restricted in the definition of *direct order dependency* below.

Definition 3.2 Given is transaction $T = (a_1, \cdots, a_i, \cdots, a_j, \cdots, a_n)$. Now action a_j has a *direct order dependency* with respect to action a_i if $od(a_j, a_i)$ and no action a_k exists with $i < k < j$ such that both $od(a_j, a_k)$ and $od(a_k, a_i)$. The fact that a_j is direct order dependent on a_i is denoted as $dod(a_j, a_i)$. So, we have:

$$dod(a_j, a_i) \Leftrightarrow od(a_j, a_i) \wedge (\nexists a_k)((od(a_j, a_k) \wedge od(a_k, a_i))$$

□

In a transaction consisting of a sequence of the elementary action types, order dependency between two actions a_j and a_i exists in the following cases. If a_i defines a variable that is used as an operand by a_j, the actions cannot be interchanged without changing the effect of the transaction; this kind of order dependency is called *definition dependency*. Definition dependency models the *dataflow* between operations and is therefore also called *dataflow dependency*. If a_i updates a variable that is used as an operand by a_j, interchanging the actions causes a_j to 'see' a wrong value of the variable; this kind of dependency is called *value dependency*. If both actions a_i and a_j have side effects (produce output), interchanging the actions changes the order of the side effects; this kind of dependency is called *output dependency*. Every possible ordered pair of action types can be analyzed to obtain the conditions under which the two actions have an order dependency relation; these conditions are listed in Table 2[1].

The set of all direct order dependencies between the actions in a transaction can be described by means of a graph. The technique to represent sets of dependencies by means of a graph is commonly used in compiler construction, both for general

[1]In this table the symbol $\equiv$ is to be interpreted as 'is the same variable as'.

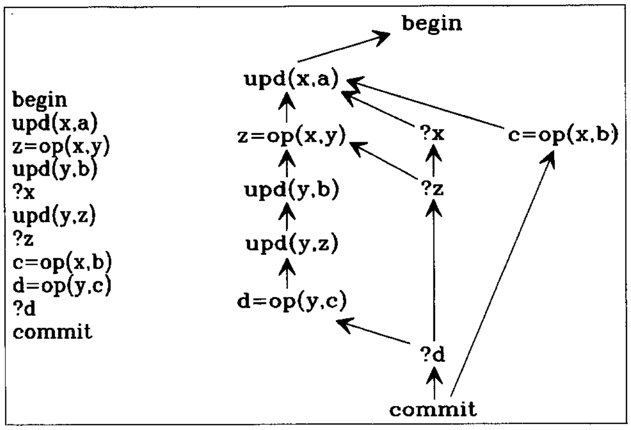


Figure 1: Example transaction and O-graph

purpose programming languages [1] and for database programming languages [9]. The definition below describes a graph representing direct order dependencies. Figure 1 shows an example transaction and the corresponding graph.

Definition 3.3 Given is transaction $T = (a_1, \cdots, a_n)$. The *Direct Order Dependency Graph* or O-graph of T is a directed graph $G = \langle V, E \rangle$. The set of vertices V corresponds with the actions a_i in T: $V = \{v_1, \cdots, v_n\}$ and v_i is labeled with a_i. The set of edges E corresponds with the direct order dependencies that exist between the actions a_i of T: $E = \{\langle v_1, v_2 \rangle \in V \times V \mid dod(action(v_1), action(v_2))\}$. □

3.2 Dynamic transaction optimization

Dynamic transaction optimization is the technique of scheduling the execution of individual actions within a transaction such, that the overall transaction execution response time is reduced, under the condition that the semantics of the transaction are not affected. Reduction of response time is obtained by scheduling the execution of the actions as early as possible. This implies monitoring the execution of the actions and acting on the occurring events; transaction optimization is thus a dynamic process. Transaction optimization can be seen as a form of global query optimization [12], complementary to the usual query optimization techniques dealing with expression rewriting, common subexpression elimination and such [3].

Scheduling the actions of a transaction to reduce transaction response times implies deviating from a fully sequential execution of the actions in the order specified by the transaction. A number of practical situations exist in which such a deviation is beneficial. In case of *parallel processing* possibilities, several actions can be scheduled to be executed in parallel. Take example transaction $T1$ in Table 3. The two update actions are fully independent, so they can be executed in parallel. In case of *unavailable resources* necessary for some actions, the execution order of actions can be changed. Resources are mostly database data (relations). This case is illustrated by transaction $T2$ in Table 3. If some other transaction holds a lock on resource a at

T1	T2	T3
begin	begin	begin
upd(a,b)	upd(a,c)	upd(a,b)
upd(c,d)	?c	upd(c,d)
commit	commit	alarm(op(a))
		commit

Table 3: Example transactions

the start of the execution of $T2$, the first action cannot be executed, but the second can; therefore, changing the execution order will reduce transaction execution time. In case of actions that may cause a *transaction abort*, these actions can be scheduled earlier to avoid unnecessary work in case of an abort. This situation is common in a system that handles integrity constraints through transaction modification [7]; using this technique, constraints are translated into an *alarm* operator that can trigger a transaction abort if some condition holds on its operand. Transaction $T3$ in Table 3 performs two updates on relations a and c. At the end of the transaction, an integrity constraint is evaluated over relation a[2]. If this constraint is violated, the transaction will be aborted and the update on c has been superfluous work. Scheduling the *alarm* before the second update will avoid this.

Scheduling the actions of a transaction may not affect the semantics of the transaction. This implies that it must be guaranteed that the optimized execution of a transaction has exactly the same effect as the fully serialized execution of the transaction, both in terms of database transition and side effects. The concept of order dependency is used for this purpose: actions in a transaction may not be scheduled such, that two actions having an order dependency are executed in an other way than sequentially[3] in the order as indicated by the transaction. As shown before, the dependencies in a transaction can be represented conveniently using an O-graph. Therefore, this graph will be used as the basis for the transaction optimization algorithms discussed below.

3.3 Scheduling using the O-graph

Transaction optimization is performed by scheduling algorithms operating on an O-graph as follows. A new vertex is added to the graph when a new action in the transaction is submitted to the system. Adding a new vertex implies adding all edges that originate from this vertex, using a decision matrix as depicted in Table 2. When a new resource becomes available, the resource administration of the transaction is updated. This administration is used to check when all resources for an action are available. The action associated with a certain vertex is submitted

[2] If relation a has a domain constraint c on attribute i, the constraint enforcement action is $alarm(\sigma_{\neg c(i)}a)$. This construct triggers a transaction abort whenever the result of the selection is non-empty, i.e. whenever tuples in a violate c.

[3] As described later in this paper (Section 7.2), some form of parallelism can be allowed between two actions that have a dataflow dependency. This is an operational aspect, however, that is not of importance on the conceptual level discussed here.

for execution when the action is order executable (see below), and all necessary resources are available to the transaction. A vertex is removed from the graph when the execution of the action associated with this vertex has been completed. Removing a vertex implies removing all vertices ending in this vertex. The concept *order executable* is used to indicate wether an action can be executed with respect to its order dependencies. This concept is defined formally as follows:

Definition 3.4 An action a_i of a transaction T is *order executable* if the vertex that corresponds with a_i has no outgoing edges in the O-graph $G = \langle V, E \rangle$. So:

$$oexec(a_i) \Leftrightarrow \{\langle v_1, v_2 \rangle \in E \mid action(v_1) = a_i\} = \emptyset$$

□

4 Resource dependency and Concurrency control

This section discusses the dependencies between actions belonging to different transactions that make use of the same resources. These *resource dependencies* can be represented by means of a graph, usable for the scheduling of the actions to obtain a concurrency control protocol.

4.1 Resource dependency

Below the definitions of resource dependency and direct resource dependency are given; these are analogous to the definitions of order dependency and direct order dependency given before. In short, two actions are *resource dependent* if an execution of these actions other than sequential and in the specified order, may violate the serializability property of the transactions the actions belong to.

Definition 4.1 Given are two transactions $T_1 = (a_1, \cdots, a_i, \cdots, a_m)$ and $T_2 = (b_1, \cdots, b_j, \cdots, b_n)$. Now action a_i has a *resource dependency* with respect to action b_j if the execution of a_i in T_1 requires resources that are obtained or will be obtained by T_2 and that cannot be released before the execution of b_j has been completed. The fact that a_i is resource dependent on b_j is denoted as $rd(a_i, b_j)$. □

Definition 4.2 Given are transactions T_1 and T_2 as shown above. Now action a_i has a *direct resource dependency* with respect to action b_j, if $rd(a_i, b_j)$ and no action c_k exists in any transaction T_x being executed by the system, such that both $rd(a_i, c_k)$ and $rd(c_k, b_j)$. The fact that a_i is direct resource dependent on b_j is denoted as $drd(a_i, b_j)$. So we have:

$$drd(a_i, b_j) \Leftrightarrow rd(a_i, b_j) \wedge (\nexists c_k \in T_x)(rd(a_i, c_k) \wedge rd(c_k, b_j))$$

□

The set of all direct resource dependencies between the actions of transactions being executed by the system at a given time can be described by means of a graph, called R-graph. This graph is similar to the Wait-For-Graph (WFG) used commonly in

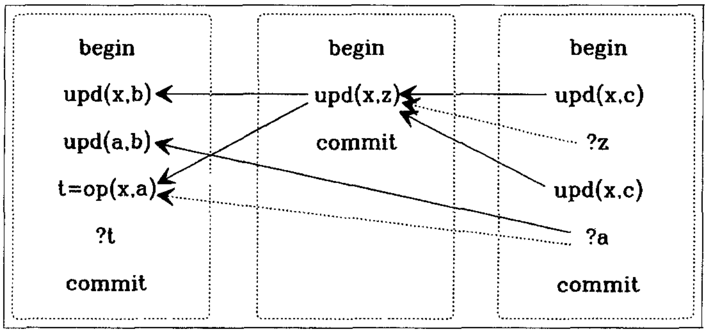


Figure 2: Example transactions and R-graph

concurrency control [4, 3]. The R-graph as defined below contains more information than a WFG, however, since its nodes are not transactions, but actions of the transactions.

Definition 4.3 Given is a set of transactions $\mathcal{T} = \{T_1, \cdots, T_m\}$ being executed by the system at a given time, with $T_i = (a_1^i, \cdots, a_{n_i}^i)$. A *Direct Resource Dependency Graph* or R-graph of a transaction set $\mathcal{T}$ is a directed graph $G = \langle V, E \rangle$. The set of vertices V corresponds with the actions a_j^i of the transactions in $\mathcal{T}$: $V = \{v_i^j \mid 1 \leq i \leq m \wedge 1 \leq j \leq n_i\}$ and vertex a_j^i is labeled with action a_j^i. The set of edges E corresponds with the direct resource dependencies that exist between the actions a_j^i of $\mathcal{T}$: $E = \{\langle v_1, v_2 \rangle \in V \times V \mid drd(action(v_1), action(v_2))\}$. □

The use of an R-graph is independent of the locking scheme used. Figure 2 shows an example of an R-graph with three concurrently executing transactions. The resource dependencies associated with the solid edges are based on a two-phase locking protocol with shared and exclusive locks [4]. If exclusive locks are used only, the dependencies associated with the dotted edges are added to the graph.

4.2 Concurrency control

As shown above, resource dependencies between actions of multiple transactions can be represented conveniently by means of an R-graph. Therefore, this graph can easily be used for concurrency control purposes. Concurrency control is then performed by manipulating the R-graph and keeping a resource administration as follows.

A new vertex is added to the R-graph when a new action in a transaction is to be executed. This can be done at two different moments: when the action is submitted to the system, or when the transaction is ready to actually execute the action. These situations can be described as "greedy locking" respectively "lazy locking". Inserting a new vertex implies inserting all edges originating from this vertex. After a vertex has been added to or removed from the R-graph, the graph is scanned for actions that can be submitted for execution, i.e. actions that are *resource executable*. When an action is submitted for execution, the resource administration may need to be updated (resources may have become unavailable). A vertex associated with an action is removed from the R-graph when the execution of that action has been completed. Removing a vertex implies removing all edges ending in this vertex. Further, the resource administration may need to be updated (resources may have become available). The definition of *resource executable* used here is analogous to the definition of *order executable* as given before.

5 Integrating both worlds

In the previous sections the notions of order and resource dependency and their graph representations were discussed; as mentioned before, the concepts are much alike. Therefore, both types of dependencies can be integrated into one global graph representation describing all dependencies between the actions being handled by the system.

Definition 5.1 Given is a set of transactions $\mathcal{T} = \{T_1, \cdots, T_m\}$ being executed by the system at a given time, with $T_i = (a^i_1, \cdots, a^i_{n_i})$. A *Global Direct Dependency Graph* or G-graph of $\mathcal{T}$ is a directed graph $G = \langle V, E \rangle$. The set of vertices V corresponds with the actions a^i_j of the transactions in $\mathcal{T}$: $V = \{v^j_i \mid 1 \leq i \leq m \wedge 1 \leq j \leq n_i\}$ and vertex v^i_j is labeled with action a^i_j. The set of edges corresponds with the direct dependencies that exist between the actions of $\mathcal{T}$: $E = \{\langle v_1, v_2 \rangle \mid dd(action(v_1), action(v_2))\}$. The *direct dependency* relation between two actions is defined as follows:

$$dd(v_1, v_2) \quad \Leftrightarrow \quad dod(v_1, v_2) \vee (drd(v_1, v_2) \wedge \neg(\exists v_3 \mid od(v_1, v_3) \wedge rd(v_3, v_2)))$$

□

Informally, a G-graph is constructed by merging the R-graph and all the O-graphs of the transactions being executed, and removing all superfluous resource dependencies; a resource dependency is superfluous here, if it indicates that an action a_1 must wait for a resource while an action a_2 that is surely executed before a_1 is waiting for the same resource. Figure 3 shows a G-graph with three transactions, based on the R-graph shown before. In this graph, the edges within the boundaries of the transactions represent direct order dependencies; edges crossing transaction boundaries represent direct resource dependencies.

Global scheduling of the execution of actions in a system can be based on the G-graph. The scheduling is analogous to the scheduling based on an O-graph or R-graph: actions submitted to the system are added to the graph, executable actions are submitted to the action execution layer, and completed actions are removed from the graph. Global scheduling based on the G-graph implements both dynamic transaction optimization and concurrency control within one simple conceptual mechanism.

6 Architectural issues

In this section the architecture of an action scheduler and its integration into a DBMS are discussed at a conceptual level. These ideas are used in Section 7 in a

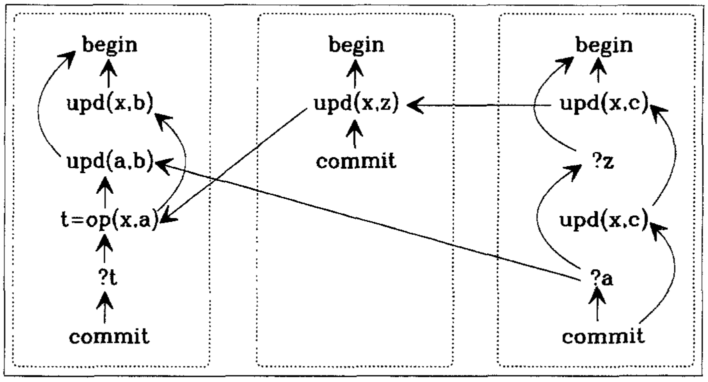


Figure 3: Example transactions and G-graph

real world DBMS.

6.1 Action scheduler architecture

An architecture for the action scheduler is shown in Figure 4. The *Graph Processor* forms the heart of the action scheduler; this module maintains the dependency graph using the algorithms described before. The processor receives new actions to be added to the graph when new taks are submitted to the action scheduler. Actions that are executable are sent for execution to the execution control. When the execution of an action has been completed, the action is removed from the graph; its resources are handed back to the resource control module. The *Action Analyzer* module analyzes incoming actions prior to sending them to the graph processor. The analysis detects the resources necessary for the execution of the actions; resource requests are sent to the resource control module. The *Resource Control* module keeps an administration of the resources needed for transaction execution. If it does not manage all resources in the system, it can take steps to acquire resources from other resource controllers. Available resources are sent to the graph processor. The *Execution Control* module controls the execution of actions that were released by the graph processor. It monitors the execution, such that completion of actions can be notified to the graph processor.

6.2 Integrating the action scheduler into a DBMS

The action scheduler as discussed above can be integrated into a complete DBMS architecture. The most simple approach is to have one central action scheduler in the system. In the case of a distributed system, it can be advantageous to split up the scheduler into a number of schedulers that each perfrom part of the scheduling task. These two cases are discussed below.

An action scheduler managing the entire G-graph of a system can be used as a centralized transaction management layer of the DBMS, controlling both transaction execution and concurrency between transactions. This situation is depicted in Figure 5. The action scheduler forms the interface between the action preprocessing

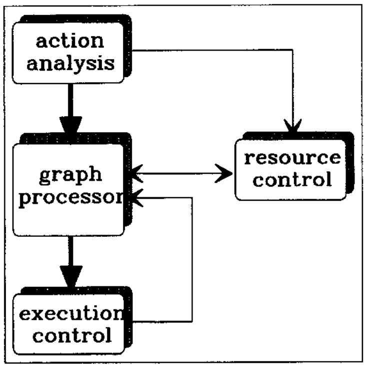


Figure 4: Action Scheduler architecture

layer and the action execution layer of the system. It accepts transaction specifications from the query optimizer, and submits actions to the execution layer of the system. Note, that pipelining is possible in this process: the transactions can be handed in several pieces to the action scheduler, and scheduling can start immediately when a piece is available. In this way, parallelism can be obtained between action preprocessing, scheduling, and execution. Parallelism between the various query processing layers of a DBMS can result in an improved overall performance [11].

In a distributed (parallel) DBMS it can be advantageous to distribute the action scheduling tasks to avoid the scheduler becoming a performance bottleneck. Distributing the scheduling tasks can easily be done by partitioning the dependency graph and assigning a private graph processor to each partition. The graph processors can then be allocated on different processors of the hardware architecture. A natural form of distribution is obtained by partitioning the central G-graph into one central R-graph and an O-graph for each transaction being executed. This leads to a situation with distributed transaction management and centralized concurrency control. The corresponding system architecture is depicted in Figure 6. It is possible to distribute the central R-graph also to obtain distributed concurrency control as well.

7 Action scheduling in PRISMA/DB

The action scheduler architectures as depicted in Figures 5 and 6 can be used in real world database systems. This section discusses the application of the ideas in the context of the PRISMA parallel database management system [10, 14].

7.1 Architecture

PRISMA is a parallel and multi-user database management system; therefore, the distributed architecture as shown in Figure 6 is used. The PRISMA architecture consists of three layers. The *transaction preparation* layer consists of the various user interfaces to the system and the query optimizer. This layer accepts transac-

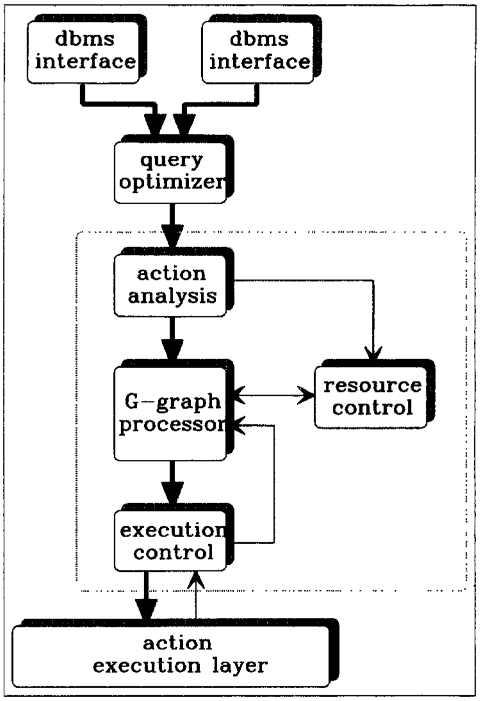


Figure 5: Action scheduling with centralized transaction management

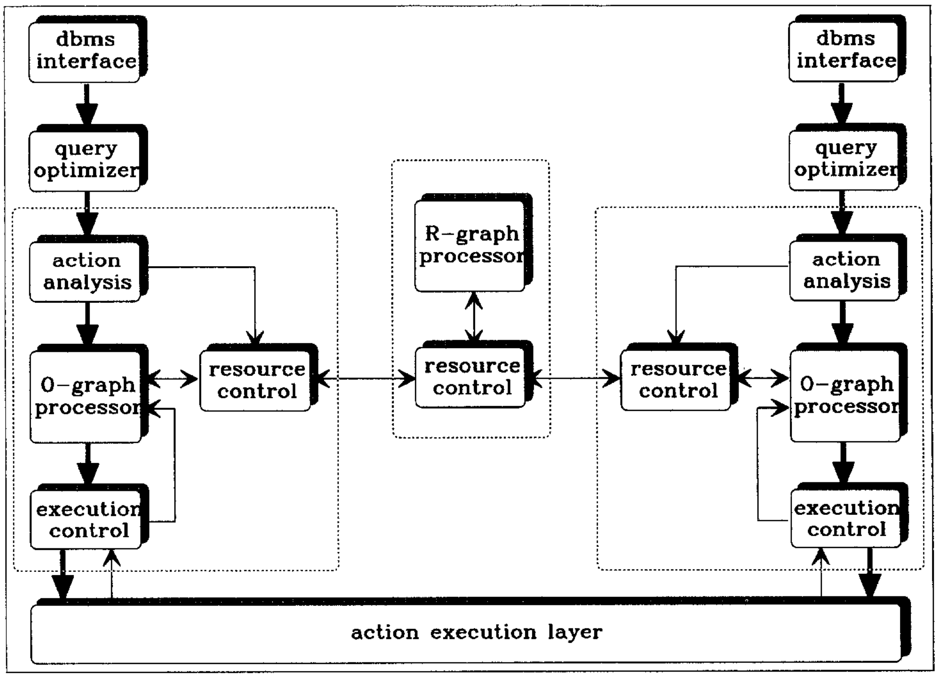


Figure 6: Action scheduling with distributed transaction management

tion specifications submitted to the system and performs some transformations on the specification of the transactions. The *transaction management* layer manages the parallel execution of the transactions. It consists of a centralized concurrency controller and a transaction manager per transaction being executed. The *transaction execution* layer consists of a parallel relational algebra engine that executes the actions submitted by the transaction management layer in a parallel fashion. Transaction are specified in an extension to the relational algebra, that can easily be mapped to the abstract actions presented before.

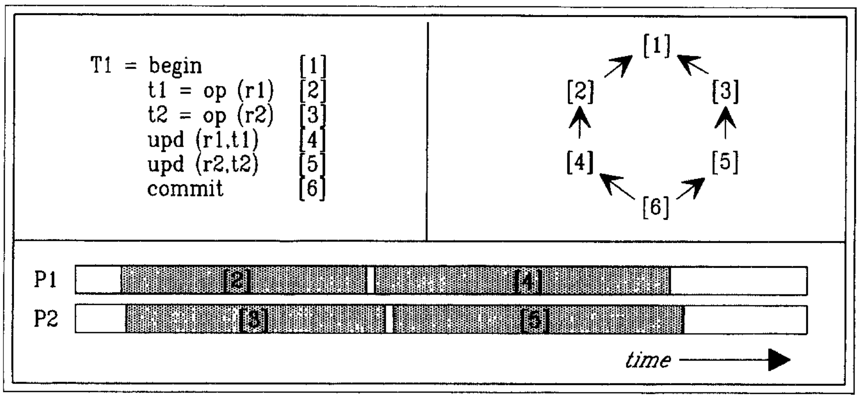
The O-graph processor is located in the transaction manager (TM) of PRISMA. It uses a variation on the graph processing algorithms presented in Section 3.3. Note, that a separate TM process is created for each transaction to be executed by the system. As such, O-graph-based scheduling is decentralized per transaction. The R-graph processor is located in the concurrency control unit (CC) of PRISMA. In cooperation with the TM's, the CC employs a simple two-phase locking protocol [4, 3] with shared and exclusive locks. Locks are always released at transaction commit. This implies that all edges in the R-graph end in a *commit* action. A centralized CC process is used because this simplifies the design of the system, and enables cheap deadlock detection (the entire R-graph is located on one node of the system).

7.2 Measurement results

This section presents the results of measurements to show the effectiveness of transaction optimization in the PRISMA context. The goal of this section is to give the reader a general impression, not to present a complete performance analysis. Two situations are discussed: the situation in which parallel execution of actions is used to reduce the execution time of a transaction, and the situation in which the order of execution of actions is changed because of unavailable resources. Currently, PRISMA does not make extensive use of early abort situations, so this situation cannot be demonstrated here. The measurements presented below were performed on a POOMA shared-nothing multi-processor [13]. Further details can be found in [8].

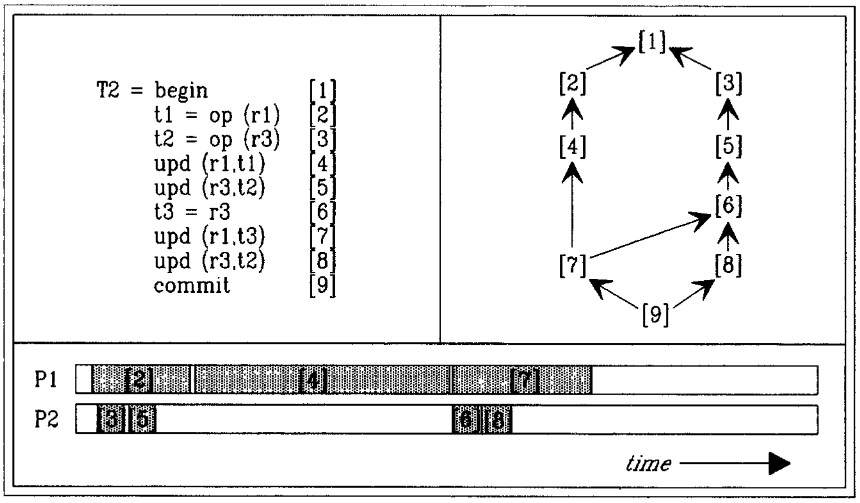
Figure 7 shows the execution of a transaction $T1$. The upper left part of the figure shows the transaction in terms of the action types presented before. The O-graph of the transaction is depicted in the upper right part of the figure. The lower part of the figure shows how the execution of the actions takes place in time on the processors of the system. Each bar represents one processor executing actions of the transaction. The length of the bars represents the total execution time of the transaction, including control overhead at the beginning of the transaction and logging at the end of the transaction. Only the execution of the actions is shown in the bars; the scheduling of the transaction takes place on a different processor and is not shown. Transaction $T1$ performs actions on relations $r1$ and $r2$ allocated on processors $P1$ and $P2$. The actions on $r1$ and $r2$ are mutually independent, so they can be executed in parallel. The time bars show that this is indeed the case. The gain in transaction execution time compared to a sequential execution of all actions is obvious.

Figure 8 shows the execution of a transaction in which order dependencies exist



```
T1 = begin          [1]
     t1 = op (r1)   [2]
     t2 = op (r2)   [3]
     upd (r1,t1)    [4]
     upd (r2,t2)    [5]
     commit         [6]
```

Figure 7: Scheduling of a transaction



```
T2 = begin          [1]
     t1 = op (r1)   [2]
     t2 = op (r3)   [3]
     upd (r1,t1)    [4]
     upd (r3,t2)    [5]
     t3 = r3        [6]
     upd (r1,t3)    [7]
     upd (r3,t2)    [8]
     commit         [9]
```

Figure 8: Scheduling of a transaction

between the actions on the relations involved. Transaction $T2$ performs actions on relations $r1$ and $r3$. Actions [2] through [5] can be scheduled in the same way as in the previous example. As shown in the O-graph, action [7] is order dependent on action [6]. Action [6], however, involves the transfer of data between processors $P1$ and $P2$ (modelled here as a "remote" assignment), and cannot be executed before $P2$ is ready to receive, i.e. has executed action [4]. The fact that PRISMA exploits pipelining parallelism [14, 15] enables a parallel execution of actions [6] and [7]. This is an implementation detail, however, that does not violate the theory of order dependency. Action [8] is order dependent on action [6], and has to wait for the completion of this action. This example shows again, that the execution of actions is scheduled as early as possible.

In Figure 9 the execution of two concurrent transactions $T3$ and $T4$ is depicted. Transaction $T3$ is started slightly earlier than $T4$ in this example. Therefore, $T3$ first obtains an exclusive lock on relation $r1$. Resource $r1$ is unavailable at the start

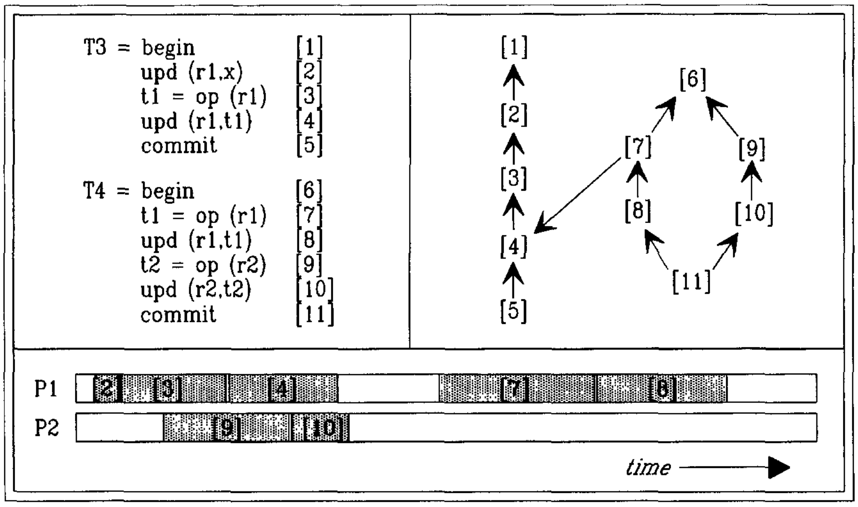


Figure 9: Scheduling of concurrent transactions

of $T4$, so the actions of $T4$ on $r2$ are scheduled earlier than those on $r1$. The actions of $T4$ on $r1$ are executed as soon as $T3$ has released its locks. Note, that $T3$ has to log its updates before it can release its locks on relation $r1$[4]; this accounts for the time gap between the execution of actions [4] and [7].

8 Conclusions

This paper describes a dynamic action scheduling technique that makes use of parallel action execution, resource availability information and early abort situations to improve both the response times of transactions and the throughput of a database system. This results in an improvement of the overall system performance. The scheduling technique will be most beneficial in multi-user systems with complex transactions, i.e. transactions consisting of many actions. These complex transactions may be defined by the user, or generated by the system.

Dynamic action scheduling can easily be described using a graph-based approach, casting the scheduling algorithms into graph processor algorithms, used for a graph-based action scheduler. This scheduler can be integrated into an abstract DBMS architecture. To accomodate decentralized transaction management, the scheduler can be decentralized by partitioning the global graph it operates upon. The feasibility of the approach is demonstrated by the implementation of a decentralized graph processor in the PRISMA parallel database system. Measurements performed on this system show the effectiveness of the scheduling technique, both for reducing transaction response times and for improving system throughput. Further measurements with more complex situations will be conducted in future.

The technique can easily be extended and improved in a number of ways. Firstly, the analysis of the dependencies between actions can be made more "intelligent".

[4]This is due to the fact that transactions in PRISMA do not release any locks before end of transaction, and logging is considered an integral part of a transaction performing updates.

Secondly, the scheduling algorithms can use resource information not only concerning data resources, but also concerning processing resources.

References

[1] A.V. Aho, J.D. Ullman; *Principles of Compiler Design*; Addison-Wesley, 1978.

[2] H. Boral et al.; *Prototyping Bubba, A Highly Parallel Database System*; IEEE Trans. on Knowledge and Data Engineering, Vol. 2, No. 1, 1990.

[3] S. Ceri, G. Pelagatti; *Distributed Databases, Principles and Systems*; McGraw-Hill, 1984.

[4] C.J. Date; *An Introduction to Database Systems, Volume II*; Addison-Wesley, 1983.

[5] M.H. Eich, D.L. Wells; *Database Concurrency Control Using Data Flow Graphs*; ACM Trans. on Database Systems, Vol.13, No.2, 1988.

[6] P.W.P.J. Grefen, P.M.G. Apers; *Parallel Handling of Integrity Constraints on Fragmented Relations*; Proc. Int. Symp. on Databases in Parallel and Distributed Systems; Dublin, Ireland, 1990.

[7] P.W.P.J. Grefen, P.M.G. Apers; *Integrity Constraint Enforcement through Transaction Modification*; Proc. Int. Conf. on Database and Expert Systems Applications; Berlin, Germany, 1991.

[8] P.W.P.J. Grefen; *Dynamic Action Scheduling in a Parallel Database System*; Memorandum INF91-58; University of Twente, The Netherlands, 1991.

[9] B.E. Hart, S. Danforth, P. Valduriez; *Parallelizing a Database Programming Language*; Proc. Int. Symp. on Databases in Parallel and Distributed Systems; Austin, USA, 1988.

[10] M.L. Kersten et al.; *A Distributed Main Memory Database Machine*; Proc. 5th Int. Workshop on Database Machines; Karuizawa, Japan, 1987.

[11] K. Li, J.F. Naughton; *Multiprocessor Main Memory Transaction Processing*; Proc. Int. Symp. on Databases in Parallel and Distributed Systems; Austin, USA, 1988.

[12] K. Satoh, M. Tsuchida, F. Nakamure, K. Oomachi; *Local and Global Query Optimization Mechanisms for Relational Databases*; Proc. Conf. on Very Large Data Bases; Stockholm, Sweden, 1985.

[13] M.C. Vlot; *The POOMA Architecture*; Proc. PRISMA Workshop on Parallel Database Systems; Noordwijk, The Netherlands, 1990.

[14] A.N. Wilschut, P.W.P.J. Grefen, P.M.G. Apers, M.L. Kersten; *Implementing PRISMA/DB in an OOPL*; Proc. Int. Workshop on Database Machines; Deauville, France, 1989.

[15] A.N. Wilschut, P.M.G. Apers; *Dataflow Query Execution in a Parallel Main-Memory Environment*; Proc. Int. Conf. on Parallel and Distributed Information Systems; Miami Beach, USA, 1991.