The HiPAC Project: Combining Active Databases and Timing Constraints*

U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy,
M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin

Computer Corporation of America
Four Cambridge Center, Cambridge MA 02142

M.J. Carey, M. Livny, R. Jauhari

Department of Computer Science
University of Wisconsin, Madison WI 53706

Abstract

The HiPAC (High Performance ACtive database system) project addresses two critical problems in time-constrained data management: the handling of timing constraints in databases, and the avoidance of wasteful polling through the use of situation-action rules that are an integral part of the database and are monitored by the DBMS's condition monitor. A rich knowledge model provides the necessary primitives for definition of timing constraints, situation-action rules, and precipitating events. The execution model allows various coupling modes between transactions, situation evaluations and actions, and provides the framework for correct concurrent execution of transactions and triggered actions. Different approaches to scheduling of time-constrained tasks and transactions are explored and an architecture is being designed with special emphasis on the interaction of the time-constrained, active DBMS and the operating system. Performance models are developed to evaluate the various design alternatives.

1. Introduction

Applications, such as Computer Integrated Manufacture (CIM), power and data distribution network management, air traffic control, battle management, chemical and nuclear process control, and knowledge source control in expert systems require the time-constrained processing of large amounts of data. In contrast to traditional applications in which performance can be measured by the system's throughput and timing constraints are essentially convenience factors (e.g. a banking transaction should not require more than 2 sec), in truly time-constrained systems the correctness of a result depends not only on the correctness of a computation or the proper interleaving of operations, but also on the timeliness of the result.

In a conventional DBMS the database is a passive data repository. To determine whether a change to data in the database has made a condition become true, an external query has to be issued by the application program. If the time window in which a response has to occur is small, then the only solution is frequent polling of the database with the concomitant waste of resources. Real-time systems have resorted to external trapping of signals which allows only basic, mostly single attribute conditions to be evaluated.

This work was supported by the Defense Advanced Research Projects Agency and by the Rome Air Development Center under Contract No. F30602-87-C-0029. The views and conclusions contained in this report are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

Active database managers provide the basic mechanisms to reduce the waste incurred by polling while providing the full capabilities of a DBMS. In an active DBMS conditions are defined on database states and events and the corresponding actions are defined as part of the database. Instead of the applications polling the database, the DBMS's *condition monitor* evaluates the pertinent conditions and invokes the proper action(s). By extending the execution model correct execution of transactions and triggered actions can be guaranteed. In addition, to provide timeliness, techniques for estimating worst case execution times of database operations must be improved and DBMS scheduling algorithms must be extended to satisfy relative task urgencies.

The combination of active database management and timing constraints is the basis for providing contingency plans. Contingency plans are alternate actions that can be invoked whenever the system determines that it cannot complete a task in time, be it a condition evaluation or a triggered action. In such a case, a contingency plan is an alternate action that is more economical but still provides useful results.

The HiPAC project at CCA focuses on three critical aspects of time-constrained database management: active database management, timing constraints, and contingency plans.

To achieve the desired functionality, the HiPAC project is investigating the following major topics:

- The nature of the conditions and tasks handled in those applications requiring active, time-constrained data management.
- The knowledge model for expressing complex situation-action rules and timing constraints.
- An execution model that provides consistency in the face of conditions and triggered actions that can be coupled with varying degrees of tightness.
- The condition monitor and tactics to handle and optimize overlapping, dynamic sets of realistic conditions.
- Scheduling algorithms that allow for timing and resource constraints and database controlled preemption.
- An architecture that defines the interaction among functional components and with the underlying operating system.
- A modelling task that will result in the necessary performance models for active, time-constrained database systems.

To develop a true real-time DBMS (which guarantees that hard deadlines are met), additional problems, such as main memory database management, aggressive resource allocation, and new operating system primitives have to be integrated with an active, time-constrained DBMS.

This paper reports research in progress and does not attempt to provide proven solutions, but discusses briefly the main problems and the approaches we are taking.

2. Application Analysis

The goal of this portion of our research is to identify applications that require active, time-constrained data management and to generalize their features. Rather than solving yet another special case, we are attempting to identify what primitives should be provided in the knowledge model, the execution model, the condition monitor and the scheduler. In doing so we have split the application analysis into two main areas:

- the nature of the tasks;

• the nature of the conditions (or situation part of the situation/action rules).

The remainder of this section describes the criteria that were applied to the applications we analysed: air traffic control, battle management, process control/fault diagnosis, network management, CIM - shop floor control, and knowledge source control in expert systems. We are actively pursuing a closer interaction with potential users of this technology and invite suggestions and submission of further requirements.

2.1 The nature of tasks:

The nature of the tasks will have the biggest impact on the execution model and the scheduling algorithms.

Predictability of arrival rate. Periodic arrival is the basic case addressed by all real-time or near real-time systems. A lower degree of predictability is a statistical arrival of tasks according to a distribution function which is followed within a known degree of confidence. Tasks may have completely unpredictable arrival rates or they may have an unpredictable triggering condition after which tasks will arrive in either periodic or predictable manner.

Nature of timing constraints and value functions. Only an analysis of the applications can determine the criticality of meeting a timing constraint, what the mixture of hard and soft timing constraints is, and what the shape of the value function is. This will ultimately drive the scheduling algorithms and determine what constructs should be available in the knowledge model to describe timing constraints and criticality.

Predictability of duration. The basis for any hard timing constraints is the ability of predicting the worst case duration of a task. This is difficult at best in a database environment because of three main reasons: a) database operations are data driven, b) database operations involve I/O operations, and c) database operations involve sharing of resources. An analysis of the application determines which are the objects associated with a task, what traffic can be expected in the database and what metadata have to be kept for dynamic estimation of duration. Predictability is most important for scheduling and the execution model and will be discussed further in those sections.

Contingency plans. Contingency plans are meaningful only for some tasks. Examples of contingency plans are the use of less resolution in a spatial search or the use of old aggregate data if the aggregate changes only slowly in response to updates to underlying data. Application analysis has to determine what kind of contingency plan should be provided and the scheduler has to provide the mechanisms for their evaluation and timely invocation.

Atomicity of tasks. The issue is whether to model single, atomic tasks or sets of tasks with precedence constraints. Single atomic tasks are easier to schedule, while the conflict-free interleaving of small tasks with precedence constraints is more efficient. From the requirements of some representative applications, we will determine the types of constraints to support in the execution model.

2.2 Nature of conditions and triggered actions

The nature of the conditions will have its largest impact on the knowledge model, which has to provide the necessary primitives to express the situation-action rules and on the condition monitor. The triggered actions and the coupling between situations and actions will have a major impact on the execution model.

Triggering events and notion of time. The triggering event is the event that causes a condition to be evaluated, for example, an update or a clock-condition becoming true. Therefore, an analysis of the applications yields the basic constructs of the knowledge model needed for representation of triggering events and the data that arrive with them, as well as the model of time which is most adequate for representation of timing constraints in a variety of applications. For example, absolute time, relative

time, or time relative to other events. Precipitating events can be periodic or aperiodic, or an aperiodic event can trigger a periodic condition evaluation. The applications also indicate that combinations of non-time-related events with time-related events occur.

Result transfer from conditions to actions — binding. It is important to analyze what portion of the data used in condition evaluation will be used in the action portion of the situation/action rule, how this information should be passed on, and how tightly the action has to be coupled with the situation part and, if applicable, with the transaction that caused the firing of the situation/action rule.

Scope of evaluation domain. To be useful in condition evaluation, data may have to be filtered or converted into discrete categories which are meaningful to the user and therefore employed for condition definition. The scope of the condition evaluation domain refers to what data are involved in the definition and evaluation of the condition.
Typical examples are:

- *selection condition*: single attributes of a single object (T > 300 deg);
- *restriction condition:* multiple attributes of a single object (T > 300 deg AND P > 100 psi);
- *aggregation condition*: single attribute aggregated over multiple instances (total fuel requirement exceeds available fuel);
- *join condition:*
  single common attribute of multiple homogeneous objects — sensor correlation: (if Output(sensor1) = Output(sensor2) then create trace object);

  several attributes of heterogeneous objects (if P(stream1) > Pmax(tank5)).
- *application-specific operators:* Time normalization or interpolation of data used in condition evaluation.

Single point vs. data histories. Single data points appear sufficient for most simple conditions. However, most applications that were analyzed appeared to require data histories to determine trends, such as multiple positions to determine direction and speed of a moving object, or multiple temperature and pressure points to determine trends and possible faults in a reactor. Databases typically consider the present state of the database as the consistent state of the world. In evaluating conditions, this may not be true. For example, interpolation/extrapolation may be required to determine the position of a moving object on which a condition is defined.

Interdependency of conditions. Conditions can be grouped and may be activated (and deactivated) as a set. For example, after take-off, a different set of situation/action rules applies than while taxiing on the runway. The grouping of conditions raises issues of naming groups of situation/action rules, their association with different object classes and the reset semantics for conditions. These are issues that drive the knowledge model.

Attachment of conditions to objects. It is important to be able to associate database objects with situation-action rules. The knowledge model has to provide the mechanisms to attach rules to objects (e.g. by making rule objects first class objects), while the applications have to dictate where the rules should be attached.

2.3 Illustrative examples

To illustrate the analysis we will discuss briefly two examples.

Example 1: "Every time more than 5 platforms converge on a given area, issue an 'abnormal concentration' report".

This condition can be expressed also as the intersection of the direction vector with an area determined by the endpoints of the diagonal, i.e. it requires the use of historical data to derive the direction and spatial data are involved in the condition evaluation. The scope of the condition spans multiple attributes of multiple simple objects and requires join operations. The condition is evaluated aperiodically whenever a new position report arrives. The arrival rate is unpredictable and metadata are required to predict the duration of condition evaluation. The timing constraint is soft and a value function in the form of a step function or a monotonically decreasing function may be appropriate. The triggering event is an update and the condition does not have to be evaluated before the update transaction can be committed.

Example 2: "If distance < 10 mi activate threat analysis with hard deadline < 2 sec."

This is an example of a condition evaluation with a hard timing constraint. This example also exhibits aperiodic condition evaluation and unpredictable arrival rates. The duration of the evaluation is predictable and it could be modelled either as a state or an event constraint.

3. Knowledge Model

The goal of this task is to develop a "knowledge model" that provides primitives for defining situation-action rules, timing constraints, control mechanisms for efficient rule searching, and support for the execution model primitives.

The elements of a useful knowledge model exist today in semantic or object-oriented data models and in AI knowledge representations. For example, data models provide abstractions and constructs for capturing factual data (entities or objects, attributes or properties, and relationships) and operational behavior (constraints, queries, operations, derived data, and simple triggers), and for specifying the atomic units of concurrent execution over shared data (atomic transactions). AI knowledge representations provide analogous abstractions and concepts for expressing facts (semantic nets, predicates, frames or objects) and operational behavior (inference rules, production rules, procedural attachment, and active objects); but typically they lack mechanisms for concurrent access to shared data.

The HiPAC knowledge model will integrate object-oriented modelling concepts and AI knowledge representation techniques. Specifically, we will use the interpretation of the PROBE data model [MANO86, MANO87], for *entity* (used to model a real-world object) and *function* (used to model attributes or properties of entities, operations on entities, and relationships among entities). Entities may be of one or more *types*, which may be arranged in *generalization (IS-A) hierarchies*. Also, the PROBE data model provides generic (i.e., representation- and dimension-independent) concepts for modelling spatial and temporal objects, based on the notion of *point-sets*, and a useful class of recursive queries called *traversal recursion*.

The necessary extensions for HiPAC are:

- Rule objects — which can be used to express constraints, derived concepts, exceptions, policies, complex triggers, and active objects; and can also be used as inference rules or production rules in a backward or forward chaining manner.
- Specific temporal constructs for expressing events (e.g., check the account balance at 5 p.m. every working day), timing constraints (e.g., this task has a hard deadline of 2 seconds), task urgencies (e.g., task 1 should be completed before task 2), transition conditions (e.g., the old balance exceeds the new balance), and historical predicates (e.g., the output of this sensor has increased monotonically over the last half hour).

- Execution model primitives — in addition to the basic atomic transactions, the model allows the specification of a variety of coupling modes. Coupling modes are defined between transactions that cause events of interest to occur and the evaluation of conditions in rules triggered by these events, and correspondingly between the evaluation of a condition and the actions that are triggered if the condition evaluates to true. The model also allows the specification of priorities among tasks that can execute concurrently.

In HiPAC's knowledge model, rules are first-class objects. This means that rules can be typed and can participate in type hierarchies (e.g., the IS-A hierarchy); rules can have attributes (functions) just like any other object; and rules can be related (via functions) to other objects, including other rules (e.g., to form rule-sets).

A rule will typically consist of a type specification; a triggering event; a situation part; an action part; a timing or priority specification; coupling modes between transaction and condition evaluation, and between condition evaluation and action; and rule-attributes.

The triggering event for a rule may be a temporal event (e.g., the clock signal "5 p.m."), the execution of a database operation ("update object O"), or an arbitrary application signal (e.g. diagnostic routine on a hardware component sends failure signal). The situation part of the rule is essentially a query against the database state (possibly including historical information). The action part is any DML program (in our case, written in the PROBE algebra [MANO86, MANO87]), which may include calls to application programs and commands for enabling or disabling some rules. The semantics are that the situation part is evaluated when the triggering event occurs (provided the rule is enabled). If the situation is true (i.e., the query returns a non-empty result), then the action part may be scheduled for execution. The scheduler schedules transactions, situation evaluations, and triggered actions in a way that respects the coupling modes and timing specifications. The type and functions of a rule can be used to set up contexts for reducing the scope of rule search.

The following example illustrates the various components of a rule:

Type:	Intermediate_Range
Parameters:	USE self IN Friendly_Platform, target IN Hostile_Platform
Event:	ON modify Position(self) or Position(target)
Situation:	IF Distance(self, target) < criticaldist
Action:	THEN
	begin
	modify Alert_Code(self) := orange
	disable rules of type Intermediate_Range for self, target
	enable rules of type Close_Range for self, target
	notify Home_Base(self)
	end
Timing:	deadine < 2 secs.
Mode:	immediate
Other attributes:	

We are working on defining the syntax and semantics of the various components of rule objects identified so far. (The syntax used in the above example was intended only to be suggestive.) As the requirements of the applications become clearer and the execution model is refined further, the knowledge model may have to be correspondingly refined.

4. Execution Model

A conventional database system consists primarily of user application programs that are invoked when a user explicitly requests so. Execution of such programs typically results in the processing of a sequence of *transactions*, where each transaction is assumed to be a unit of atomicity and recovery. Concurrency control and recovery techniques are used in conventional DBMS's to preserve the serializability and the permanence requirements of database transactions.

In an active DBMS, however, the system must execute triggered programs, or actions, in addition to user transactions. How should the execution of triggered actions be treated with respect to user transactions? Are there serializability and permanence requirements associated with such actions? The execution model is an attempt to answer such questions.

4.1 The Constrained Serializability Model

Most of the currently proposed strategies for handling trigger execution essentially consider trigger execution to be part of the triggering "transaction." Within this framework, some triggers may be evaluated and invoked *within* a triggering transaction (i.e., before the latter arrives at its end point), and others may be evaluated only at the end of the triggering "transaction." In either case, the triggered action is basically an in-line extension of the triggering "transaction," and the atomicity requirement is applied to the combined execution. We shall call such triggers *immediate triggers*.

While immediate triggers are obviously an important class of triggers, we propose to consider more formally an execution model in which triggered actions are allowed to be broken off into different threads from those of the triggering "transactions." We propose a generalized transaction model for defining correctness of concurrent execution of *user transactions* and *triggers*. We call our model a *constrained serializability model*.

The model considers both the user transactions and the triggered actions as *task units*. At the base level (i.e., the least stringent level), the correctness criterion states that that the concurrent execution of multiple task units must be equivalent to some serial execution of the same set of task units. However, depending on the *coupling mode* of the trigger, which may be implicitly or explicitly specified, the task unit associated with the trigger may have to satisfy additional ordering requirements with other task units. In addition, condition evaluation itself must be scheduled properly. At the most stringent level, a trigger may be required to execute within the same atomic unit as the triggering "transaction," which would mimic the behavior of an immediate trigger.

4.2 Correctness Criterion and Coupling Modes

We model a trigger as a situation-action pair, denoted $<S,A>$. When S becomes true, the action A is to be executed by the system, where S is a predicate over the state of the database (which may include historical objects and other time- or event-oriented objects, as discussed in the section on Knowledge Model), and A is a general purpose procedure which may or may not update the database.

The execution of a program that checks whether S is satisfied is called a *checker* for S. The system thus executes three types of programs: Actions, Transactions, and Situation Checkers. We will call a transaction or the action of a trigger a *task unit*. A task unit or a situation checker is called an *execution unit*.

A *schedule* H is a possibly interleaved sequence of *steps* of a collection of execution units such that
(a) if there exists some task unit T in H such that a trigger TR is potentially triggerable by T, then a *checker* C for the situation S of TR on behalf of T must also exist in H, and C must come after T in H; and
(b) if C returns true, then the action A of TR on behalf of T must also exist in H, and A must come after C in H.

A correct schedule is one in which the steps of the execution units in the schedule conform to certain requirements. The weakest correctness criterion is serializability, where a schedule *H* is correct if execution units contained in *H* are serializable. We call such a schedule *EU-serializable*.

Further constraints can be added to this weakest criterion. Presently we have identified constraints based on the "coupling" of "related" execution units. We impose coupling constraints between T, a triggering task unit, and C, the checker on behalf of T for a trigger potentially triggerable by T; and between C, the checker unit for a trigger, and A, the corresponding action execution unit. In general, however, other types of constraints can also be added. The general criterion of correctness in a trigger system is therefore one of *contrained serializability*. In the remainder of the section we illustrate some of the "coupling" constraints we have developed.

We identify three different coupling relationships between T and C: Null, Weak and Strong; and two between C and A: Null and Strong. This framework provides a total of 6 different types of triggers.

The "Null" relationship simply requires EU-serializability. For example, Null coupling may be specified both on the TC side and the CA side for the following trigger:

> if #Passengers(flight) > Capacity(flight) then Redirect overbooked Passengers to other flights; notify Passengers

This means that the condition need not be checked atomically with every transaction that updates the flight records, and that the action need not be performed as soon as the overbooked situation is detected. This allows for increased concurrency.

The "Strong" relationship requires further that the two execution units involved be adjacent in the equivalent serial schedule. For example, Strong coupling may be specified on the CA side of the following trigger, but Null coupling on the TC side:

> if Quantity_On_Hand(item) < Threshold(item) and not On_Order(item)
> then Prep_to_Order(item); On_Order(item) := true; Submit_Order(item)

This allows concurrency among multiple updates to an item, but once an understocked situation is detected, the ordering action must be performed immediately.

The "Weak" relationship is weaker than "Strong", as explained further below. In essence, weak coupling treats a situation checker as a read-only transaction which may "see" a different order of the triggering task units from that in the equivalent serial schedule. We will use an example to motivate weak coupling.

Example: Consider the following trigger which takes the Aggregate_Overdrawn action whenever the sum of many accounts within the same aggregate units goes below a threshold:

> TR : If aggregate (A1,...,An) < threshold then Aggregate_Overdrawn.

Let T1,... Tn be triggering transaction units that update A1,...,An respectively. Note that T1, ...Tn themselves to not conflict with each other.

If strong coupling is requested between every Ti and the corresponding situation checker Ci (on behalf of Ti) for TR, then none of T1, ..., Tn can run concurrently (even though they do not conflict!) because strong coupling means that Ti and Ci must be executed atomically, and must be serialized with Tj and Cj, for all i, j in 1,...,n. That is, any interleaving of the form Ti Tj Ci Cj is illegal, since Ci and Tj conflict, and Cj and Ti conflict.

On the other hand, weak coupling allows interleaving of the form Ti Tj Ci Cj to occur in the schedule. Letting n = 3 in our example, suppose the system chooses the serialization order <T1, T2, T3> of the account update transactions, and the aggregate account is overdrawn at the end of T2 but no longer overdrawn at the end of T3. Then, if situation checkers C1, C2 and C3 are allowed to run

after T3, then the overdraft occurring at the end of T2 would not cause any action to be triggered. As far as the situation checker C2 is concerned, it "perceives" the system to have processed the three update transactions in the order <T1, T3, T2>, and therefore at the end of T2, C2 would see a state of the database where there is no need to fire the trigger action. As far as checker C3 is concerned, it "perceives" the system to have processed the three transactions in the order <T1, T2, T3>, and therefore at the end of T3, C3, too, sees a state that does not require the trigger action to fire. In effect, weak coupling allows "one cycle" of updates to the accounts before running the checker, but prevents any other conflicting transactions to run in between. In practice, of course, the checkers C1, C2, C3 (which are all checking the same condition over the same state of the database) could then be collapsed into one checker.

It is interesting to note that, while developed and motivated entirely separately, this notion of weak coupling is analogous to the notion of of "weak consistency" described in [CHAN85]. Weak consistency allows one site in the network to run a read-only transaction R1 with the view that T1 had run before T2, and another read-only transaction R2 on another site to run with the view that T2 had run before T1, where T1 and T2 do not conflict. In fact, what is achieved in the weak consistency model in their paper is that one site may adopt T1 R1 T2 as the equivalent SR schedule, while another site may adopt T2 R2 T1 as the equivalent SR schedule, where R1 and R2 are two read-only transactions.

The advantage of the weak coupling is obvious. For triggers with a complex condition read set, which intersects with a relatively large number of potential transactions (or task units) in the system, strong coupling between the triggering task units and checker units would destroy parallelism, even though the transactions conflict only through the triggers and are otherwise unrelated. The utility of the increased level of parallelism would depend on the applications.

Work in progress includes refining the constrained serializability model, extending the model to a nested transaction model for handling concurrently fireable triggers and immediate triggers, and developing concurrency control algorithms to support this model.

5. Condition Monitor

The main objective of this portion of the HiPAC project is to design a condition monitoring subsystem that will evaluate application defined situations in a timely manner. Since our primary concern is the efficient monitoring of complex conditions involving multiple relations, aggregates and views, we have identified a range of techniques that need to be addressed in order to optimize the evaluation of complex conditions. A variety of techniques (such as eager and lazy evaluation) are being investigated in systems that support condition monitoring (such as LOOPS/KEE, POSTGRES, SYBASE). These techniques are useful for evaluating simple conditions that access small amounts of data. When the conditions span several relations, and access and compute large amounts of intermediate data, other optimization techniques become relevant in addition to the ones that have been reported. In this section we report our initial work which has identified a range of techniques for monitoring conditions. We propose three techniques, substantiate their relevance and indicate their applicability for condition monitoring.

We take the view that conditions should specify *what* to monitor rather than *how* to monitor it. The task of optimizing the condition evaluation is best done by the system. Below, we briefly survey proposed approaches to condition specification and monitoring to provide a context for discussing our approach to condition monitoring.

POSTGRES [STON86, STON87] provides alerters and triggers with which condition monitoring as well as forward and backward inference mechanisms can be realized. Specifically, lazy and eager evaluation techniques are used for optimizing the evaluation of conditions specified by the user. Triggers and alerters are defined as queries with special key words and are associated with the

appropriate tuple or field instances by the system. In LOOPS [BOBR83] (as well as KEE [INTE85]), the association of an active value (a special data structure) to an instance variable achieves the effect of associating a trigger/alerter - for read access, for write access, or for both. Trigger/alerter is specified by the user as a LISP function and is evaluated by the system without further analysis. SYBASE [DARN87] supports situation action rules on database operations (such as insert, delete etc.) with restricted conditions and actions. For example, conditions can only reference tuples/attributes of a single relation.

In the following discussion a primitive event is an action that can be recognized by the DBMS or the underlying operating system (inserts, deletes, updates, and clock events are examples of primitive events). A situation-action rule consists of a condition part and an action part. A condition is a query which may have free variables. We do not propose any restrictions on the action portion of a situation-action rule. A primitive event (or events) is associated with a set of situation-action rules indicating when the condition should be evaluated. The primitive events that need to be monitored are passed on to the relevant portions of the DBMS and to the operating system. The end result of condition monitoring is a sequence of actions that need to be executed in a specified manner.

The evaluation of a set of conditions is analogous to the evaluation of a set of queries. However, there are some differences in the details of condition evaluation, for example, a single evaluation vs. evaluation whenever the event occurs. This similarity with query evaluation motivated the use of a graph formalism whereas the differences prompted us to explore techniques that are specific to condition evaluation. Hence, central to the concept of condition monitoring is the graph abstraction which is generated from the specifications of conditions and primitive events. The graph abstraction is viewed as an extension of the operator graph to include information that is required for processing conditions. The graph abstraction proposed here is comparable to the RETE network structure used in OPS5 [FORG82]. However, the graph abstraction differs from the RETE network in several aspects. The graph abstraction is not a decision network at the tuple level. It has nodes corresponding to database operators (such as select, join etc.) and data. The working memory is different from the database and is used for materializing data values required for optimization. Finally, instead of the recognize-act cycle, we intend to concurrently evaluate several conditions using several (perhaps overlapping) subgraphs. We consider three techniques to be especially useful for evaluating a set of conditions efficiently. They are:

- Identification and representation of common sub-conditions.
- Materialization and maintenance of intermediate results (instead of computing them every time).
- Techniques for evaluating conditions incrementally (instead of evaluating them always).

These techniques are not new. However, our contribution lies in recognizing the applicability of these techniques for condition monitoring and proposing the mechanisms for their exploitation.

Multiple Condition Optimization: The similarities between multiple query evaluation and multiple condition evaluation strongly indicate that one could benefit from the techniques developed for the other. Earlier approaches to multiple query evaluation [SELL86, CHAK86, GRAN81] have used either top down or bottom-up techniques for optimizing a set of queries. There is a need for unifying the approaches proposed earlier and to develop new techniques that are aimed towards extensibility and to handle various levels of abstraction in a uniform manner. We are investigating techniques for multiple condition optimization using levels of abstraction on the extended operator graph representation. As none of the existing DBMSs optimizes multiple queries currently, our approach not only provides a mechanism for handling multiple conditions efficiently but can be used for optimizing queries also.

Derived Data Management: Whenever a condition (or even a query) is evaluated more than once, it is beneficial to materialize partial results (results corresponding to sub-expressions, for example) and use them in the evaluation of the condition (or the query). This technique is not only useful for evaluating conditions efficiently but may play a key role when there are urgency constraints

associated with the evaluation of a condition. In order to use this technique effectively, two problems need to be addressed: a) decide what partial results need to be materialized, and b) what should be done when these partial results are no longer valid (on account of changes to underlying values used in the creation of the partial result). There has been very little work on the incremental maintenance of derived data [BLAK86, PAIG81] which is central to the problem of using derived data effectively. We are investigating techniques for determining what partial results need to be maintained and how to maintain them incrementally. The need for derived data is more pronounced when multiple conditions share common derived data and hence becomes useful in evaluating several conditions efficiently.

Incremental Condition Evaluation: Normally, a condition is evaluated when the event specified for evaluating the condition occurs. However, under certain conditions, it may be possible to infer whether the condition is true with respect to an event without evaluating it. For example, if the condition is to check whether the value quantity-on-hand is > 500 and if the condition is true to start with and the value of quantity-on-hand is increased, then there is no need for re-evaluating the condition at all. Though the above example is simplistic, whenever large amounts of data have to be accessed for evaluating a condition, the saving in computation can be substantial. In the above example, incremental (or selective) evaluation of the condition is derived from the properties of the operators in the condition and the properties of the operators that cause the event to occur. Some work has been done in this direction by [BERN80]. We plan to integrate these techniques into our approach and extend them to a wider class of conditions.

5.1 An Illustrative Example

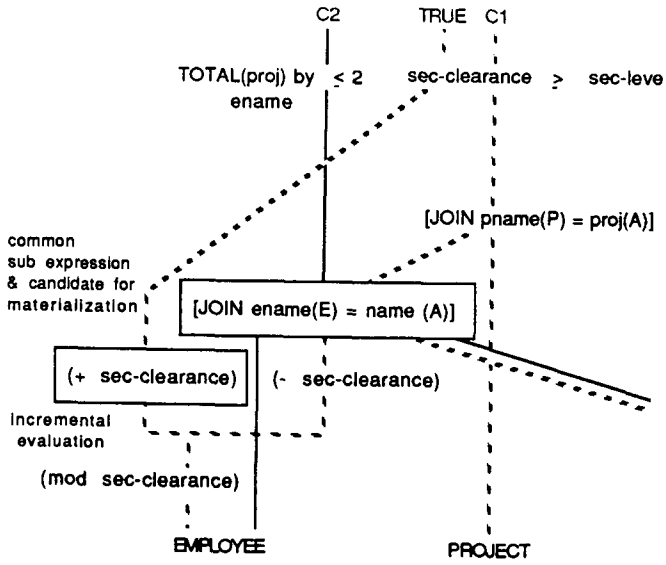

Figure 1: Condition evaluation

Figure 1 indicates the application of the above techniques on two conditions specified over a database. This example is presented at a conceptual level indicating the usefulness of the techniques and need to be refined further using the techniques proposed in this paper. The database and the conditions are as follows:

Objects:

EMPLOYEE(ename, sec-clearance, salary, etitle)
PROJECT(pname, sec-level, budget)
ASSIGNMENTS(name, proj, capacity)

Conditions:

C1: An employee can work only on projects at levels for which s/he has clearance.

C2: An employee can work only on two projects at a time

Referring to Figure 1, a common join is detected by the multiple condition optimizer. It is also a potential candidate for materializing values that are relevant to the conditions C1 and C2. If it is chosen for materialization, its maintenance have to be incorporated into the graph. It is evident that condition C1 gives rise to incremental evaluation of the condition exploiting the properties of '>=' operation. That is, any update to the EMPLOYEE relation that increases the sec-clearance value cannot make the condition C1 false.

5.2 Further Research Issues

We concentrated in this section on the techniques that are useful for efficient evaluation of complex conditions. Currently, we are investigating each of the techniques proposed in this paper to develop the necessary algorithms for their use. We are also refining the graph abstraction in conjunction with the development of algorithms. For condition monitoring, incremental addition and deletion of situation-action rules seems to be a necessity. As a result, techniques for modifying the graph incrementally to accommodate dynamic addition and deletion of situation-action rules need to be investigated further.

Though we have defined only primitive events in this paper, it may be useful to have an event calculus for specifying complex events. Another useful capability would be the ability to define conditions on derived objects (termed virtual conditions).

6. Scheduling

The central part of any time-constrained system is the scheduler. The goal in the HiPAC project is to develop scheduling algorithms that satisfy timing and priority requirements at the same time they satisfy concurrency constraints. To accomplish this goal, the interaction between the HiPAC scheduler and the operating system scheduler has to be defined carefully, since many of the interaction problems between a DBMS and the OS are magnified.

Before addressing the scheduling issues, a clear understanding of what is meant by *real-time* in the context of database systems is needed. The common interpretation of *real-time* is that hard timing constraints can be enforced and that tasks can be *guaranteed* to execute within the time specified by the hard timing constraint. Three basic problems arise in a database environment:

1. Concurrency control mechanisms that depend on blocking introduce unpredictable delays. Therefore, if predictability is crucial, a predeclaration scheme based on transaction preanalysis or a relaxed notion of consistency which allows for non-serializability has to be used. "Cautious schedulers" [KATO85] fall into this category. In essence, they identify a feasible schedule that will never lead to abort, and schedule actions of a transaction according to that schedule. The difference between a cautious scheduler and the scheduler for a real-time system is that the scheduling problem for the latter is augmented by additional constraints: precedence constraints due to triggers, and timing constraints imposed by deadlines.
2. Database operations are intrinsically data dependent and a transaction may change its execution time from one execution to the next. Execution time may change because of more instances involved in an operation, additional levels in an index, or new insertions that require the following of additional pointers. Predictability of database operations can be increased by using access methods and storage strategies that are predictable (at least within bounds) and do not degrade in unexpected ways. Finally, additional metadata can be kept, such as number

of tuples and levels of an index to make dynamic estimates of the duration of an operation, similar to the information used during query optimization. Although this may not yield 100% accuracy and complete predictability, it is a reasonable approximation.

3. Database operations involve massive I/O. Only for the simplest case of strictly periodic transactions can the state of the buffer or the availability of the channel be anticipated. Any mix that involves aperiodic transactions makes it impossible to predict the state of the buffer or the contention for a physical device. A pessimistic (worst case) approach which assumes a page fault per access is unacceptable since many transactions which would actually complete would not be scheduled. This problem will disappear in main-memory databases. In the meantime, segmented buffers for periodic and aperiodic tasks and a probabilistic approach to modelling page faults with monitoring of a transaction's progress at predetermined check-points appears to be the best possible solution.

For of these reasons it is difficult to talk about *real-time database systems*. However, through careful design and some trade-offs time-constrained database processing is possible. This is the notion underlying our discussion of scheduling.

Scheduling issues can be addressed in terms of a task model, a resource model, a load model, a processing model, an interrupt model, a delay cost model, and a discussion of the algorithm classes. We shall discuss these issues and state what approach appears most relevant for time-constrained database systems.

Task model: The task model specifies whether the scheduler should look at single atomic tasks or composite tasks combined through precedence relationships. For HiPAC we selected composite tasks built from atomic tasks and precedence relationships. Tasks should be dynamically schedulable and the scheduler has to enforce both inter- and intra-transaction precedence constraints. Finally, we assume that contingency plans can be defined for some tasks and can be invoked when the scheduler realizes that a time constraint cannot be met. Trigger chain reactions introduce a source of non-determinism since the action of one situation-action rule could be the evaluation of another rule. For any meaningful estimate the possible propagation of situation-action rules must be limited.

Resource model: The resource model assumes the existence of shared and exclusive resources. Examples of shared resources are data, while processors and channels are regarded as exclusive resources. The granularity of data resources has to be determined carefully, since scheduling algorithms that consider resources are usually of O(n2) where n is the number of resources. Clearly, a granularity at the record or tuple level is unrealistic. The resource model has to include mechanisms for predicting resource requirements in the presence of triggered (chains of) actions and the assignment of resources under various priority schemes.

Interrupt model: The interrupt model determines whether preemptive or non-preemptive execution is used and specifies the preemption modes and mechanisms that support preemption. HiPAC assumes preemptable tasks with preemption predicated on the resource sharing mode. Preemption is limited in a time-constrained database environment since preemption no longer means just the suspension of a task while another task executes. Since database tasks operate on shared resources, preemption may require roll-back to guarantee database consistency, possibly causing longer delays than if the transaction that is executing is allowed to finish. Preemption has to be, therefore, DBMS-controlled.

Load model: The load model specifies task arrival rates and duration. We are not limiting arrival patterns at this time, thus allowing both periodic tasks and random, aperiodic arrivals. Some of the applications we are considering dictate the need for handling long overload conditions, a situation in which the use of contingency plans becomes particularly attractive.

Delay cost model: The basic delay cost models distinguish between hard and soft timing constraints. We discussed the problems at the beginning of this section. We are leaning towards a value function approach in which hard time constraints can be modelled as step functions. We are studying mechanisms to express criticality and metrics that allow us to determine the goodness of a value function. Predictability should be strived for whenever possible (as indicated above), but probabilistic durations appear more useful than worst case time estimates whenever worst case estimates are so pessimistic that they deviate from actual behavior in a manner that cripples the scheduler's ability to schedule tasks, particularly under heavy load conditions.

Processing model: The processing model describes the hardware assumptions. Most real-time scheduling algorithms assume the availability of a dedicated scheduling processor to avoid the additional complexity of having to account for the variable time used by the scheduler as part of the total available time. This is a reasonable simplification which has to be simulated in the absence of an actual processor by assigning a fixed time slice to the scheduler.

Scheduling algorithms: Time-constrained database systems require dynamic on-line scheduling of preemptable and non-preemptable tasks with time-, resource-, and precedence-constraints. The scheduling algorithms have to recognize priorities (or the criticality of a task) and have to allow early evaluation of a task's schedulability to invoke contingency plans. We are currently studying various approaches developed for real-time Operating Systems, among them the combination of heuristics proposed as part of the SPRING project at the University of Massachusetts, Amherst [STAN87], [ZHAO87a], [ZHAO87b], [ZHAO87c], and the best-effort scheduler, a value-function approach proposed as part of the Alpha kernel developed at Carnegie Mellon University [LOCK86]. Both approaches need substantial modifications to be useful in a time-constrained database environment. For example, they will have to be modified to handle concurrency constraints and contingency plans. We are also investigating the complementary approach: starting with a conventional DBMS concurrency control algorithm and modifying it to satisfy the precedence constraints of the execution model and simple types of timing constraints (e.g., task priorities).

7. Architecture

Another goal of the HiPAC project is to develop an architecture for an active DBMS with time constraints. New DBMS subsystems must be introduced to support rules and time constraints. Conventional DBMS subsystems must be modified to take time constraints into account and interfaces between the new and existing subsystems must be defined. Furthermore, the DBMS will place additional requirements on the operating system. The long range architecture for HiPAC is envisioned to be distributed with a backend and several interconnected workstations. However, before we can think of the distributed architecture, we have to solve the problems discussed in previous sections for the single node case.

A functional decomposition of HiPAC is shown in Figure 2. The transaction management component of the DBMS has been extended to support the knowledge model, execution model, and scheduling for time constraints. A new component has been added to support condition monitoring. The object manager and physical data manager are as in an object-oriented DBMS; they have no knowledge of rules or time constraints.

The knowledge model is implemented as part of the transaction manager. When a rule is created, the metadata for a rule is stored on disk by the object manager in the form of a rule object. However the object manager does not distinguish a rule object from any other user-defined object. The transaction manager notes the triggering event for the rule. Whenever an operation is performed on an object, the transaction manager determines whether or not any rules have been triggered. If so, it schedules situation evaluation for those rules. The result of each situation evaluation is returned to the transaction manager. The transaction manger schedules the triggered action where appropriate

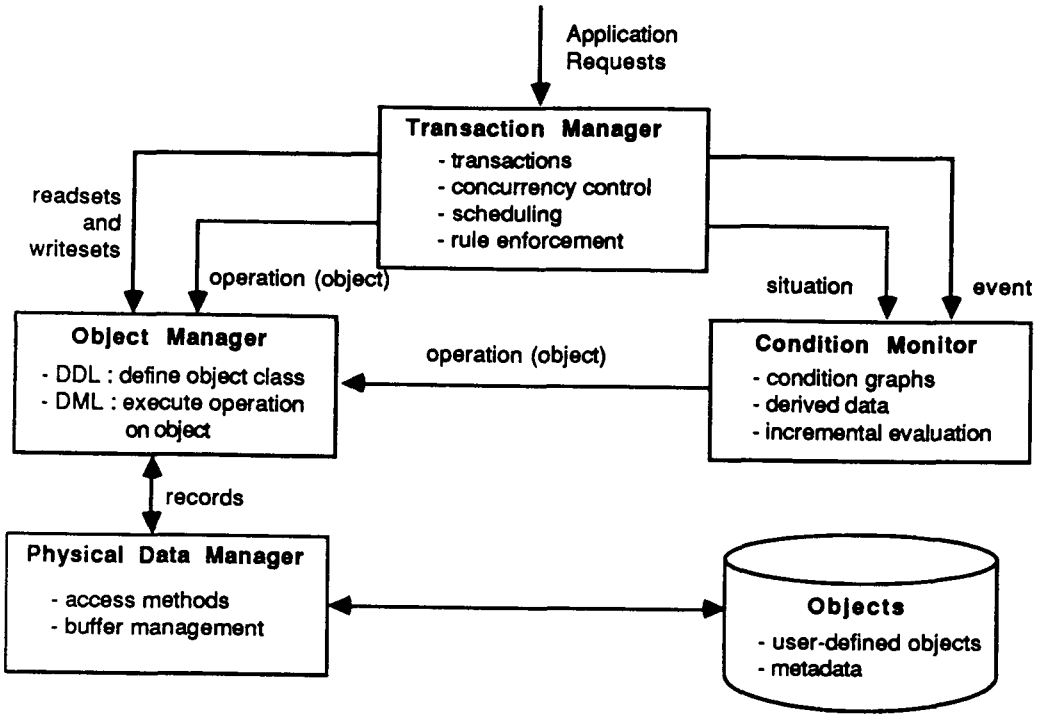

Figure 2: Schematic of HiPAC Architecture

and the action is performed by the object manager.

Condition monitoring is performed by a new DBMS component. When a rule is created, the situation portion of the rule is passed to the condition monitor. At this time the condition monitor decides how to monitor the situation. When the triggering event for a rule is detected by the transaction manager, the condition monitor is called on to evaluate the situation portion of the rule. Materialized derived data may also be updated during condition evaluation. The condition monitor returns the result of the evaluation, along with any data that must be passed to the triggered action.

The execution model is implemented in the transaction manager. If the coupling mode for a rule specifies that the situation evaluation and/or triggered action should be performed in a separate transaction from the triggering event, then the transaction manager is responsible for starting and terminating the transaction. It must also generate a correct schedule for application transactions, condition evaluation, and triggered actions.

The schedule of DBMS operations is not determined entirely by the execution model. The transaction manager must also schedule for time constraints, i.e. the correct schedule chosen must meet time constraints. This implies that the concurrency control mechanisms must take time constraints into account when deciding how to allocate resources. The additional work required to perform condition evaluation and triggered actions may cause overload. When an overload situation arises, the transaction must detect it as soon as possible and decide where to substitute contingency plans in the schedule.

Note that the database transaction is part of a unit of work that has a time constraint and is scheduled by the operating system. HiPAC must interact with the operating system to meet the time constraints. These interactions are as follows:

- HiPAC produces cost estimates for database operations that are factored into the cost estimate for the application to which a time constraint applies.
- The operating system must give HiPAC scheduling information for the applications performing the database transactions (e.g. deadlines). This information will limit the scheduling choices made by HiPAC.
- HiPAC gives the operating system scheduler serialization information. This information will limit the scheduling choices made by the operating system.

For example, suppose the deadline for application A1 comes before the deadline for application A2, and applications A1 and A2 perform database transactions T1 and T2 respectively. Then HiPAC will use this information to eliminate a serialization that puts T2 before T1.

Clearly HiPAC and the applications that use it should run under a real-time operating system. However, available real-time operating systems will not support the desired level of interaction between the HiPAC scheduler and the operating system scheduler. Therefore we are developing two architectures for HiPAC. One is an idealized architecture for an operating system that supports time constraints and interaction between the operating system scheduler and the HiPAC scheduler. Here we are using concepts from research projects in real-time operating systems, such as Alpha[NORT87] and Spring[STAN87]. The other architecture is for the HiPAC breadboard, which will run under a currently available operating system.

8. HiPAC Performance Issues

It should be clear from the preceding sections that the design and selection of alternative HiPAC algorithms and approaches *must* be accompanied by a careful performance evaluation effort. Such an effort is currently underway at the University of Wisconsin, Madison. The UW-Madison portion of the HiPAC project has several goals: to study architectural alternatives for the system and their impact on performance; to evaluate algorithms proposed for various components of HiPAC; to participate in the development of algorithms that are particularly performance-critical; and to provide a performance testbed for studying the extent to which the final HiPAC design is able to meet/exceed the time-constrained processing requirements of its target applications. This section outlines some of the performance-related work that is now in progress.

8.1 Initial Architectural Issues

The long-term target hardware configuration for HiPAC consists of a collection of workstations connected to a backend host machine via a communications network. Users initiate processing activity from the workstations, with HiPAC running on the backend system. In addition, some portion of the HiPAC software may run on the workstations so that processing can be distributed between the workstations and the HiPAC host. Interesting issues that arise in such an architecture include: How much of the data should be stored (or replicated) at the workstations? To what extent can trigger-related processing be offloaded onto the workstations? What is the performance impact of such offloading?

In order to get an initial feeling for some of these issues, we developed a simple performance model of the distributed/backend HiPAC configuration and conducted a study of the performance impact of offloading some of the trigger-related processing onto the workstations. Our initial model consisted of a system with N workstation nodes, one HiPAC host, and a communications network.

The workstations and the host were each modeled as a queuing network with a CPU and one or more disks. System-related parameters included the number of workstations, the CPU and disk capacities of the host and the workstations, and the CPU and network costs associated with network communications. The HiPAC workload was modeled as a stream of *jobs* arriving at the workstations, with each job being a sequence of one or more *tasks* (consisting of CPU and disk processing). Each job consisted of an initial (user-requested) task plus a set of additional tasks triggered by the initial task (or by subsequently triggered tasks), thus capturing the active nature of the database. The model's workload-related parameters captured the arrival rate of jobs and the processing characteristics of their initial tasks. To capture the active nature of the database, the probability of the occurrence of a trigger due to task execution was another model parameter. Finally, in order to study offloading issues, the model parameters included the probability of a task being executed locally (e.g., at its originating workstation) as opposed to being shipped to the HiPAC host for processing. This model was implemented using the DENET discrete-event simulation language [LIVN87].

We ran a series of experiments to isolate the effects of the following HiPAC configuration parameters on system performance for different probabilities of workstation versus host task execution:

(a) the number of workstations in the system;
(b) the CPU cost for sending messages over the network;
(c) the relative processing power of the host and the workstations.

The primary performance metrics for the study were job response time and throughput. In order to incorporate a time-constrained processing flavor into the model, we also maintained percentile information on job response times. Space prevents us from presenting details of the model or the results, but we summarize the main initial conclusions here.

As a general rule, we found that executing tasks at their originating workstations, as opposed to always shipping them to the HiPAC host, can lead to significant performance gains. For example, in a configuration with five workstations and one HiPAC host, throughput increased by a factor of two to three when the right fraction of the processing was offloaded onto the workstations. Further, for a given workload and system configuration, there was an optimal fraction of tasks to offload from the host onto the workstations. This optimal fraction was configuration-dependent, increasing as the number of workstations or the communications cost was increased, and decreasing as the relative processing power of the HiPAC host was increased. These results suggest several design guidelines for the HiPAC architecture. First, offloading some of the host processing onto the workstations will be important, and a dynamic scheme for offloading tasks to distribute the system's load should be investigated. Second, in order to sustain a reasonable fraction of the workload at the HiPAC host in a large configuration, the host's processing power must be significantly greater than that of the workstations. Finally, efficient message-passing techniques will be needed in order to fully take advantage of the distributed nature of the system, as high message-related costs were found to have a significant negative performance impact on the system.

8.2 Transactions, Triggers, and Performance

The current phase of our HiPAC performance evaluation work is concerned with studying the interaction between the different types of triggers provided by the HiPAC execution model and the transaction management aspects of database management. We are currently developing a much more detailed HiPAC performance model for this purpose.

Our current model captures an interesting subset of the transaction/trigger interactions possible under the HiPAC execution model. We currently support three classes of database processing tasks: triggering actions, which are user-initiated actions (or subsequently triggered actions) that can trigger other actions; trigger checkers, which are initiated to check whether or not a given condition has indeed been triggered; and resulting actions, which are actions initiated when a checker discovers that an action indeed should be taken. Each of these task types is modeled as a sequence of read and/or

write operations on database objects. The passive portion of the database is just a set of objects; the active portion of the database is a set of predicates over database objects (which we model in a probabilistic manner at the moment) and their associated actions.

In the execution model subset that we presently support, triggering actions and their associated checkers and resulting actions can be grouped into transactions in four ways. Consider an action which causes a trigger to fire — for the trigger to be processed, its checker must be run, and then the resulting action must taken (assuming that its predicate is found to be satisfied). With Strong TC and CA Coupling, the checker and resulting action execute together with the triggering action as a single transaction. With Stong TC-Coupling, the checker is coupled together with the triggering action as part of the same transaction, but the resulting action runs as a separate transaction. With Strong CA-Coupling, the triggering action is in one transaction, with the checker and the resulting action being coupled together in a single separate transaction. Finally, with Null-Coupling between T and C, and between C and A, each task executes as a separate transaction. The model currently does not consider Weak Coupling and has to be extended for this coupling mode.

Our current research work is directed towards studying the performance implications of this execution model subset. We have developed a fairly detailed model of an active DBMS, and we have just completed a DeNet implementation of the model. The model and its implementation consist of five separate module types, dealing respectively with workload generation, transaction execution, concurrency control, resource management, and trigger management. Each of these components is represented by a DeNet discrete-event module (or DEVM). Among these modules, the Trigger Manager is the module responsible for capturing the active nature of the system. The Trigger Manager encapsulates a model of the active database or "trigger base", and it is responsible for scheduling the activities that result from triggers being fired. In our current implementation, we model the active portion of the database probabilistically. Model parameters are used to control the probability of an object-level operation causing a trigger to fire, the distribution of trigger types (from among the four outlined above), and the size and contents of the read and write sets of checkers and their resulting actions.

Experiments based on this performance model are currently getting underway. After investigating the impact of the various trigger types on HiPAC performance, we intend to use the model to investigate time-constrained processing issues as well. In particular, we plan to extend our trigger model to include priorities, and we will modify the concurrency control and resource management modules of our model to employ priority in making scheduling-related decisions. We will then be able to study the effects of various concurrency control algorithms and priority-based resource scheduling schemes on system performance. We will also eventually extend the model for use in studying the host/workstation HiPAC architecture at this level of detail.

9. Conclusions

The current state of the art makes it difficult to talk about real-time database systems in a strict sense. Therefore, the HiPAC project attempts to find solutions in two critical areas, the handling of timing constraints and active database management. It also introduces the notion of contingency plans. Once these issues in time-constrained, active database management have been solved, they can be integrated with main memory database management, new resource allocation strategies, multiprocessor architectures, and a new generation of real-time operating systems into true real-time database management systems.

References

[BERN80] Bernstein, P. A., Blaustein, B. T., and Clarke, E. M., "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data", Proceedings of International Conference on VLDB, 1980, pp. 126-136.

[BLAK86] Blakeley, J. A., Larson, P., and Tompa, F. M., "Efficiently Updating materialized Views", Proceedings of SIGMOD, 1986, pp. 61-71.

[BOBR83] Bobrow, D. G., Stefik, M., "The Loops Manual", Intelligent Systems laboratory, Xerox Corporation, 1983.

[CHAK86] Chakravarthy, U. S., and Minker, J., "Multiple Query Processing in Deductive Databases Using Query Graphs", Proceedings of International Conference of VLDB, 1986, pp. 384-391.

[CHAN85] Chan, A. and Gray, R. "Implementing Distributed Read-only Transaction," IEEE Trans. on Softw Engr, SE-11, 2, February 1985, pp 205-212

[DARN87] Darnovsky, M., and Bowman, J., "TRANSACT-SQL USER'S GUIDE", Document 3231-2.1, Sybase Inc., 1987.

[FORG82] Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem", Artificial Intelligence, Vol. 19, 1982, pp. 17-37.

[GRAN81] Grant, J., and Minker, J., "Optimization in Deductive and Conventional Relational database Systems", In Advances in Data Base Theory, Vol. I, (eds. H. Gallaire, J. Minker and J. M. Nicolas), Plenum Press New York.

[INTE85] IntelliCorp, "KEE Software Development System User's Manual", IntelliCorp, Mountain View, 1985.

[KOEN81] Koenig, S., and Paige, R., "A Transformational Framework for the Automatic Control of Derived Data", Proceedings of the International Conference on VLDB, 1981, pp. 306-318.

[KATO85] Katoh, N., Ibaraki, T. and Kameda, T. "Cautious transaction schedulers with admission control," ACM Trans. on Database Syst. 10, 2 (June 1985) pp 205-229

[LIVN87] Livny, M., *DeNet User Guide*, Computer Sciences Department, University of Wisconsin, 1987.

[LOCK86] Locke, C.D.: "Best Effort Decision Making for Real-Time Scheduling", PhD Thesis, Department of Computer Science, Carnegie Mellon University, May 1986.

[MANO86] Manola, F.A., Dayal, U.: "PDM: An Object-Oriented Data Model", 1st International Workshop on Object-Oriented Database Systems, Monterey CA, Sept. 1986.

[MANO87] Manola, F.A.: "PDM: An Object-Oriented Data Model for PROBE", CCA Tech. Rep. September 1987.

[NORT87] Northcut, J.D.: "Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel", Academic Press, 1987.

[STAN87] Stankovic, J., Ramamritham, K.: "The design of the Spring Kernel", Proc. 1987 Real Time Systems Symposium, 1987.

[STON86] Stonebraker, M., Rowe, L., "The Design of POSTGRES", Proceedings of ACM-SIGMOD, 1986, pp. 340-355.

[STON87] Stonebraker, M., Hanson, M., and Potamianos, S., "A Rule manager for Relational database Systems", Technical Report, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1987.

[SELL86] Sellis, T., "Global Query Optimization", Proceedings of SIGMOD, 1986, pp. 191-205.

[ZHAO87a] Zhao, W., Ramamritham, K., Stankovic, J.; "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", IEEE Transactions on Software Engineering, May 1987.

[ZHAO87b] Zhao, W., Ramamritham, K.; "Simple and Integrated Heuristic Algorithms for Scheduling Taks with Time and Resource Constraints", Journal of Systems and Software, August 1987.

[ZHAO87c] Zhao, W., Ramamritham, K., Stankovic, J.; "Preemptive Scheduling Under Time and Resource Constraints", IEEE Transactions on Computers, August 1987.