On Transaction Boundaries in Active Databases: A Performance Perspective

by
Michael J. Carey
Rajiv Jauhari
Miron Livny

Computer Sciences Technical Report #796
November 1988

On Transaction Boundaries in Active Databases: A Performance Perspective

Michael J. Carey
Rajiv Jauhari
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

The workload of an active DBMS consists of two types of activities: externally generated tasks submitted by users, and rule management tasks caused by the triggering of rules stored in the knowledge component of the system. Most design proposals for active DBMSs assume that an external task should be combined with all the resulting rule management tasks into a single transaction. There is no compelling reason for this assumption, however; the semantics of rules can be used to divide the workload into transactions in a number of different ways. In this paper, we describe a performance model of an active database, and present the results of simulation experiments that study system performance as a function of transaction boundary semantics for varying levels of data contention, rule complexity, and data sharing between externally submitted tasks and rule management tasks. Our results demonstrate that the way in which transaction boundaries are imposed can have a major impact on the performance of an active DBMS, and that this aspect of rule semantics must therefore be carefully considered at the time that rules are specified.

This work was performed under subcontract to Xerox Advanced Information Technology, which was supported by the Defense Advanced Research Projects Agency and by the Rome Air Development Center under Contract No. F30602-87-C-0029. Additional support was provided by the National Science Foundation under grant IRI-8657323 and by the Digital Equipment Corporation through its Initiatives for Excellence program. 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.

1. INTRODUCTION

It is generally recognized that in order to meet the growing demands placed on information processing systems, rules and facts need to be integrated into a "knowledge base management" framework. A computer-controlled factory, for example, can be expected to take advantage of knowledge base management in a number of areas. The inventory of the factory will be controlled automatically by rules that trigger "ordering" actions when the facts of the system indicate a low "quantity-on-hand" of some part. The beginning of each step of the manufacturing process will be triggered by a rule that recognizes the end of the preceding step. Facts regarding the structural properties of the products will be measured and recorded as the products pass through the manufacturing line, and defective items will be identified with the help of rules that trigger actions to discard such items. The "technical secrets" of the manufacturing process may be kept in a secure file to which access is restricted via rules, with the additional safeguard that the names of all those who access the file are automatically logged.

Rules and facts can be integrated in a number of ways. One way to structure a knowledge base management system is to couple a rule manager and a DBMS loosely via a polling mechanism. The rule manager, which appears as just another user to the DBMS, periodically runs queries to monitor the facts stored in the DBMS. Depending on its rules, update transactions may also submitted by the rule manager. It has been argued in [Ston88], however, that such a loosely-coupled system is likely to perform poorly (or even incorrectly) unless the application has specific characteristics — for example, that the rules need to access only a small part of the data, and that the data remains unchanged while it is being processed by the rules. Most large-scale applications of knowledge-based data management (for example, factory control, air traffic control, and stock trading) seem not to have these characteristics, and thus require a tighter coupling between the rules and the facts. Such applications have motivated the development of a number of *active* DBMSs (e.g., [Ston86c, Sell88, Kotz88, Rasc88, Daya88c]), in which rules and facts are tightly-coupled. In these systems, data and rules are both managed by the DBMS using an integrated language and data model.

The workload of an active DBMSs is a mix of *External Tasks (ETs)* and *Rule Management Tasks (RMTs)*. ETs are queries and updates that arrive at the active DBMS from a user or an application program, whereas tasks that result from the presence of rules (both condition-checking tasks and responses to satisfied conditions) constitute the RMTs of the active DBMS. The manner in which these tasks are combined to form transactions may have an impact on the performance of the active DBMS. In most of the proposals for active DBMSs that have appeared in the literature (e.g., [Ston86c, Rasc88]), the user has little flexibility in determining transaction boundaries. For example, a rule in POSTGRES [Ston86c] might state that the values of two data items d_1 and d_2 are required to be equal. In this case, an update to d_1 will cause d_2 to be automatically updated as well; the automatic d_2 update will take place either when d_1 is updated ("eager evaluation") or else when d_2 is subsequently accessed ("lazy evaluation"). Thus, in POSTGRES, the RMT will be coupled either to the updating ET or to the accessing ET, depending on the evaluation scheme used. As will be demonstrated by an upcoming example, the semantics of a rule do not always require such strict coupling of the external task and the internally generated tasks.

In this paper, we explore the performance implications of exploiting rule semantics to determine transaction boundaries in an active DBMS. In particular, we study the impact of rule semantics on the performance of the system as the following characteristics of the workload are varied: the level of data contention, the complexity of the ETs and the RMTs, and the degree of data sharing between an ET and the RMTs caused by it. The execution and knowledge models developed as part the High-Performance Active (HiPAC) Database project [Daya88a] were used to guide the design of our model of an active DBMS. The goal of the HiPAC project, which is currently under way at Xerox Advanced Information Technology and the University of Wisconsin-Madison, is to design an active, time-constrained, and object-oriented database system. The execution model of HiPAC provides a number of different coupling modes that allow the user to specify the way in which ETs and RMTs should be combined into transactions. The coupling mode is considered part of a rule, and specified at the time the rule is created.

The remainder of the paper is organized in the following manner. Section 2 surveys related work in the area of active database systems, including a brief overview of the HiPAC project. We discuss our model of transaction processing in an active DBMS in Section 3. Our probabilistic model of an active DBMS is described in Section 4. In Section 5, our experiments and their results are discussed. Section 6 contains our conclusions and a brief discussion of our continuing work in related areas. Finally, a detailed example showing how ETs and RMTs can be combined to form transactions based on rule semantics is presented in the Appendix.

2. RELATED WORK

Little work has been done in the area of evaluating the performance of knowledge-based data management systems. However, the need for integrating rules and facts in a DBMS context has been recognized for a long time, as the lack of a rule facility can place a significant burden on the DBMS application programmer. For example, in order to support general integrity constraints the absence of such a facility, every transaction that updates the database for a given application must be augmented with code to check the constraints and to take an appropriate action if a constraint is violated. Thus, the condition checking activity and the action become integral parts of users' transactions. In recent years, various approaches have been suggested for adding active capabilities to database systems in order to integrate rules and facts and thus simplify the application programming task. We briefly review some of the major approaches in the rest of this section.

The addition of active capabilities to database systems was first considered in order to support specific DBMS functions such as view maintenance and integrity constraint enforcement. It was proposed in [Eswa75, Eswa76] that "triggers" be added System R in order to enforce integrity constraints or "assertions." Triggering mechanisms of different types have also been suggested to support the maintenance of materialized views, snapshots, and derived attribute values [Bune79, Koen81, Rous82, Morg83, Blak86, Huds86, Lind86, Hans87]. More recently, it has been proposed that generalized active data management capabilities be added to database systems in order to provide a unified mechanism to support a variety of applications, such as those requiring rule-based inferencing [Ston86a, Kotz88, Rasc88, Sell88, Tzvi88]. The POSTGRES project [Ston86b, Ston86c] proposes a general mechanism to support alerters, triggers and rules (among other features) while making as few changes to the relational model as possible.

The HiPAC project combines active, object-oriented databases with time constraints [Daya88a, Daya88b]. In the HiPAC knowledge model [Daya88c], rules are treated as an object class, like all other forms of data. Each rule in HiPAC is structured according the *event-condition-action (ECA)* paradigm. The *event* associated with a rule is the cause of the rule being activated, such as an update to the "quantity-on-hand" object set in the factory control example discussed earlier. The *condition* is a set of queries which must be evaluated in order to check whether a pre-specified set of predicates has been satisfied. If cars were assembled in the factory in our example, a condition could be a query representing the question "is the number of radios in stock low enough that we need to order more?" Finally, the *action* represents the operations to be executed if the condition is met; for example, an order for a thousand radios could be sent automatically to a radio supplier. ECA rules can be used as a convenient mechanism to implement such active functions as alerters, rule-based inference, and constraint management. As mentioned earlier, the way that the event, condition and action of a rule are to be combined into transactions is specified as part of the definition of a rule in HiPAC. Also, each HiPAC rule can have an associated *time constraint* (e.g., a deadline) that captures the importance of completing an action within a specified time after its causing event. Finally, a rule may specify a *contingency plan* in case its time constraint cannot be met; this is a set of simpler alternative actions to be executed instead of the original condition and action.

HiPAC's transaction execution model, described in [Hsu88], is based on a nested transaction model. An external query or update, which may cause one or more events, always forms the root of a transaction tree. Queries and updates representing the corresponding conditions and actions may either be nested subtransactions (i.e., descendant nodes in the same tree) or they may form parts of other transaction trees. Different methods are proposed for mapping events, conditions, and actions onto transactions and subtransactions. A subset of these methods is included in the execution model used in this paper, as described in the next Section.

3. A TRANSACTION MODEL FOR ACTIVE DATABASES

As discussed earlier, an active database system contains a set of rules in addition to the data or facts found in all database systems. These rules can create Rule Management Tasks (RMTs) in response to External Tasks (ETs) that represent external stimuli. The semantics of the rules determine the way in which RMTs are combined with ETs to form transactions. In this section, we describe our model of transaction execution in an active DBMS. We begin by introducing the notion of a *job* as a unit of database activity. We then address the issue of coupling the various tasks of a job into one or more transactions, after which we discuss transaction restart semantics in active databases.

3.1. JOBS VS. TRANSACTIONS

A *job* is defined as the sum total of all the database activity generated by an incoming External Task. Of course, the minimum amount of work represented by a job consists of the initial External Task itself. In terms of the ECA paradigm, External Tasks can cause Events. However, an Event in the ECA model is defined as part of a rule; there may be user queries and updates whose activity does not correspond to the Event of any rule defined in the system. When an External Task causes an Event, the job will also include a *Checker* task. Checkers are system-generated queries that *check* whether the conditions of a rule are satisfied. The Checker will generate an *Action* task to perform the operation dictated by the rule if the conditions are met. Since an Action may query or update the database, it may cause Events that in turn lead to further Actions.

A single External Task could cause an arbitrarily long chain of tasks, all of which would be part of the same job. The relationship between jobs and transactions is that, while a job may consist of one or more transactions, a transaction can be part of at most one job. The simplest mapping of jobs onto transactions is one-to-one, with one transaction per job. All the tasks of a job would then be part of the same atomic unit of database activity. If there are significant fixed costs associated with transaction startup and transaction commit, limiting the number of transactions in this manner could lead to improved performance. However, as mentioned in Section 1, it is not always necessary (or desirable) for all the activity which comprises a job to be coupled together in the form of a single transaction.

3.2. COUPLING TASKS TO FORM TRANSACTIONS

The way in which the RMTs of a job are coupled with the ET and with each other into transactions depends on the semantics of the triggered rule. The imposition of a transaction structure on a job suggests the need for a nested transaction model, such as the model proposed in [Hsu88]. In this model, there are two types of choices to be made when scheduling an RMT. The first choice involves *transaction boundaries*: whether the semantics of the rule require the RMT to be part of the same transaction as the task which generated it, or instead allow it to be executed in a separate transaction. The second choice, also based on the rule's semantics, determines the *time at which the RMT should begin execution*: for example, should a Checker begin execution as soon as the triggering event takes place, or should it begin execution only when the transaction containing the causing task completes? In the work presented here, we study the performance implications of the transaction boundary decision. Our execution model thus can be viewed as a subset of the model described in [Hsu88]. In particular, in our model, any internally generated activity caused by task T of a job J is started only after the causing task is "completed". A task T is said to have completed at the time when, had T been a transaction by itself, it would have been in a position to commit.

With this assumption in mind, our model currently supports four different ways in which the tasks of a job can be coupled to form transactions:

(1) *Strict Coupling (S-Coupling)*: The ET of the job and the RMTs of each rule activated are coupled together in a single transaction; in effect, all the tasks of a job are part of a single transaction.

(2) *External Task-Checker Coupling (EC-Coupling)*: The External Task of the job is coupled with the Checkers generated by it into a single transaction, but each Action generated by a Checker begins a new transaction; recursively generated Checkers are coupled with the corresponding causing Action.

(3) *Checker-Action Coupling (CA-Coupling)*: The External Task of the job is a stand-alone transaction, but every Checker is coupled with the corresponding Action into a single transaction. Each Checker begins a new transaction.

(4) *No Coupling (N-Coupling)*: Each task of the job exists as a stand-alone transaction.

The following example illustrates the differences between these four coupling modes. Job J begins when External Task E arrives at the system. E triggers a rule, R_1, which results in Checker C_1 being executed. The condition checked by C_1 is found to be satisfied, and an Action A_1 results. In turn, A_1 activates rule R_2 and causes another Checker, C_2, to be run. C_2 leads to yet another Action, A_2. Finally, A_2 does not trigger any further rules, so job J consists of a total of five tasks: one ET and four RMTs. Table 1 shows how the four coupling modes would combine the tasks of job J into transactions; to keep the example simple, we have assumed that R_1 and R_2 have the same transaction coupling semantics. If S-coupling were employed, Job J would consist of exactly one transaction. If either EC-coupling or CA-coupling were used, there would be three transactions, although the transaction boundaries would differ as shown in Table 1. If N-coupling were used, each of job J's tasks would be its own transaction. In the Appendix we present an example that illustrates the use of these coupling modes and discusses some of the associated tradeoffs.

Coupling Mode	**Transaction Boundaries** ("[...]" is a transaction)	**Number of Transactions**
S-coupling	$[E \ C_1 \ A_1 \ C_2 \ A_2]$	1
EC-coupling	$[E \ C_1] \ [A_1 \ C_2] \ [A_2]$	3
CA-coupling	$[E] \ [C_1 \ A_1] \ [C_2 \ A_2]$	3
N-coupling	$[E] \ [C_1] \ [A_1] \ [C_2] \ [A_2]$	5

Table 1: Transaction Boundaries for Job J.

The coupling of tasks into transactions requires a modification of conventional transaction restart semantics. In our model, Checkers and Actions occur only in response to changes in the state of the data. If a transaction begins with an External Task, then the External Task must clearly be reexecuted when the transaction is restarted. If a transaction begins with a Rule Management Task, this means that the DBMS has recognized a change of state that requires the execution of a Checker or an Action. Thus, the system must guarantee that the initial RMT is repeated if the transaction is restarted. In our execution model, then, whenever a transaction aborts, it is restarted *at the first task* (External Task, Checker or Action) that formed part of the aborted transaction. The subsequent tasks may or may not be repeated, as the state of the database may have been changed by updates of other transactions that ran since the aborted transaction began.

4. MODELING AN ACTIVE DBMS

Our performance model of an active DBMS is an extension of the model of a database site used in the distributed database study described in [Care88]. It has been implemented using the DeNet simulation language [Livn88], so it can easily be extended later to include multiple sites in order to study a distributed active DBMS.

In our model, an active DBMS has five components: a *Source*, which generates the External Tasks; a *Transaction Manager*, which models the execution behavior of transactions; a *Concurrency Control Manager*, which implements the details of a particular concurrency control algorithm; a *Resource Manager*, which models the CPU, I/O, and the main memory resources of the system; and finally, a *Rule Manager*, which models the active component of the database. Figure 1 presents a detailed view of these components and their key interactions. The model described in [Care88] already contained the first four of these components. However, these components have been modified in our model to capture the active nature of the workload. Also, certain additional features (such as a buffer manager) have been included in

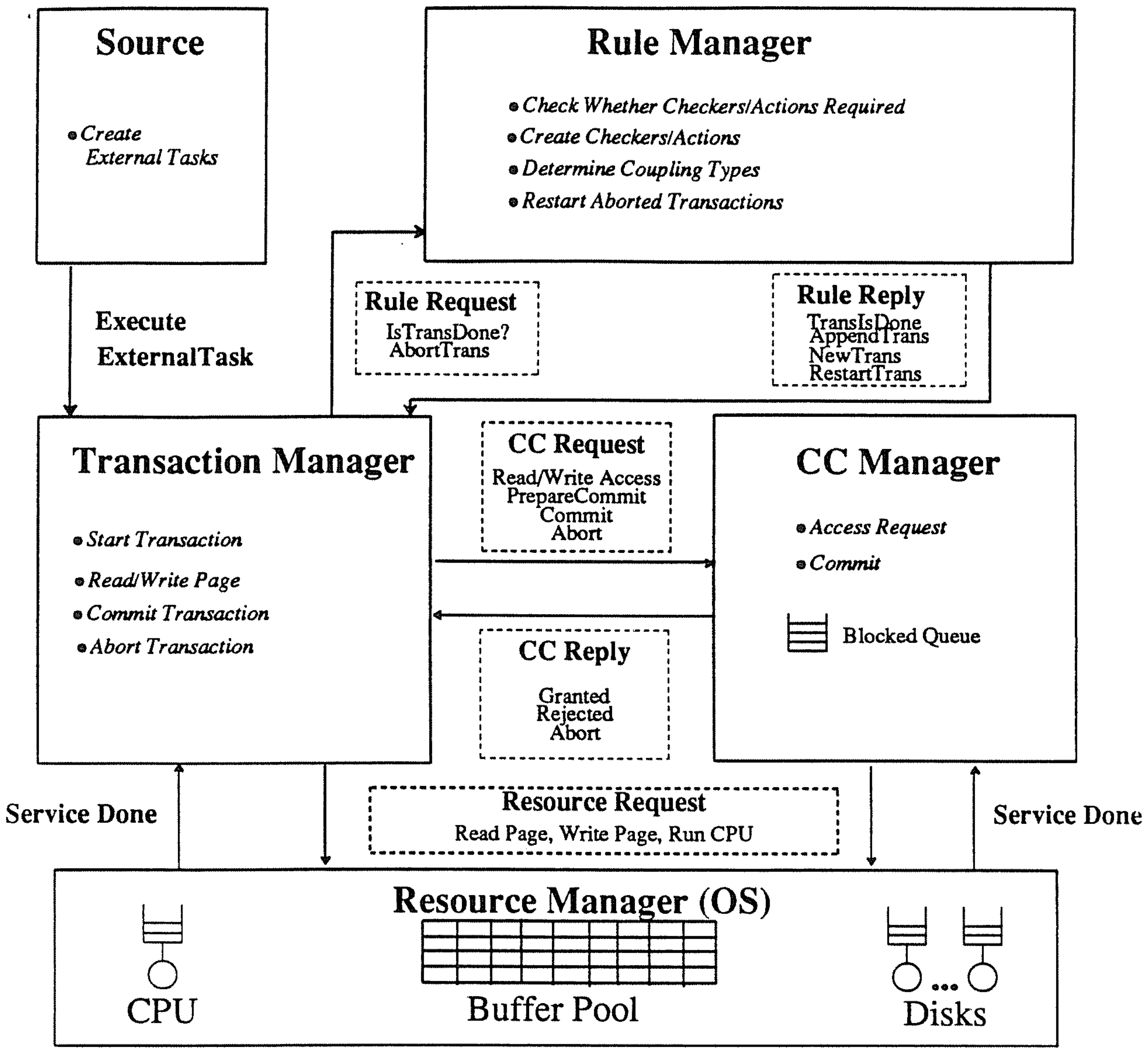

Figure 1: Components of the Active DBMS Model.

our model. Of course, the major extension to the original model is the inclusion of the Rule Manager component. In this section, we first describe the passive database model. The Rule Manager module is then described in detail. Finally, the remaining modules and their parameters are briefly described.

4.1. MODELING THE PASSIVE DATABASE

The passive portion of the database is modeled simply as a collection of *files*. In turn, each file is modeled as a collection of pages. The parameters of the database model include *NumFiles*, which is the number of files in the database, and $FileSize_i$, the number of pages in each file ($1 \le i \le NumFiles$). Each file can be viewed as representing a set of database objects; we model files at the page level because we assume page-oriented concurrency control for simplicity.

4.2. THE RULE MANAGER MODULE

The Rule Manager models the active portion of the database and the active functions of the DBMS. In terms of the execution model, the Rule Manager must capture the rules in the active portion of the database, and it has to interact with the Transaction Manager in order to coordinate the execution of External Tasks, Checkers, and Actions. Table 2 lists the input parameters for the Rule Manager. A detailed description of the Rule Manager module follows.

4.2.1. Modeling the Active Database

Rather than storing rules explicitly, we model Events, Conditions and Actions probabilistically. External Tasks and Actions can cause events, while Checkers represent condition evaluations. The probability that an External Task represents an Event (i.e., that it leads to a Checker) is *ExtCheckProb*. Similarly, the probability that the condition of a rule is satisfied (i.e., that an Action needs to be scheduled) is *ActionProb*. Finally, the probability of an Action leading to another Checker is *ActCheckProb*.

In the current implementation, a job can have only a single sequential thread of tasks (i.e., a job can activate only one rule at a time), and the tasks occur in the following order:

$$ExternalTask \rightarrow Checker_1 \rightarrow Action_1 \rightarrow Checker_2 \rightarrow Action_2 \rightarrow etc.$$

Jobs may differ from one another in the actual number of constituent tasks, depending on the probabilities *ExtCheckProb*, *ActionProb*, and *ActCheckProb*. Once a decision to schedule a Checker or an Action is made, the Rule Manager must determine what *type of coupling* is to be invoked for the newly generated task. This is also done probabilistically in our model; the relevant input parameter is an array called *CouplingTypeProb*. This array has four entries, each representing the probability of using one of the four

Parameter	Meaning
	RMT Probability Parameters
ExtCheckProb	Probability of a Checker following an External Task
ActionProb	Probability of an Action following a Checker
ActCheckProb	Probability of a Checker following an Action
	Coupling Parameter
$CouplingTypeProb_i, (1 \le i \le 4)$	Probabilities of using each of the four coupling types
	Active Workload Parameters
RNumClasses	Number of task classes
RClassProb	Probability of a class
RFileCount	Number of files accessed
$RFileProb_i$	Access probability for file *i*
$RNumPages_i$	Average number of file *i* pages read
$RWriteProb_i$	Write probability for file *i* pages
RPageCPU	CPU time for processing a page of data
RWriteCPUFraction	Fraction of read CPU time required to process a write
	Data Sharing Parameters
ShareMode	Data sharing mode
ShareProb	Probability that a page is accessed by adjacent tasks of a job
	Restart Parameter
RepeatOnRestart	Probability that *all* tasks of an aborted transaction are repeated on restart

Table 2: Input Parameters for the Rule Manager.

types of coupling for combining the adjacent tasks of a job into one or more transactions.

4.2.2. Modeling the Active Workload

One of the functions of the Rule Manager is to generate the reads and writes that make up the RMTs. The active workload parameters of the Rule Manager are prefixed with "R" to distinguish them from similar parameters at the Source. RMTs are characterized into classes in terms of the files that they access and the number of pages that they read and update in each file. Since the only potential cause of an RMT is a change in the state of the database, Checkers are restricted to read-only operations. Actions are modeled as collections of reads and writes (as are External Tasks at the Source). Checkers and Actions may belong to different classes; *RNumClasses* is a two-element array representing the number of classes for both Checkers and Actions. *RClassProb* specifies the probability that an RMT will be of a given class. The remaining per-class parameters characterize each class of RMTs. *RFileCount* is the number of files accessed, and *RFileProb* gives the probability distribution (or relative file weights) for choosing the actual files that the RMT will access. *RNumPages* and *RWriteProb* respectively specify the number of pages accessed by an

RMT and the probability that an accessed page will be updated. The next parameter, *RPageCPU*, specifies the average amount of CPU time required for RMTs of the class to process (e.g., search) a page of data. The final parameter, *RWriteCPUFraction*, represents the CPU time required for writing a page as a fraction of the CPU time required to read the page. The actual number of pages accessed by an RMT ranges uniformly between half and twice the class average, and the page CPU time is exponentially distributed.

From the nature of ECA rules, it is likely that the data accessed by different tasks of a job will overlap to some extent. The data read by a Checker will probably include part of the data updated by the preceding External Task or Action, and the data processed by an Action may well overlap with the Checker's read set as well. The input parameters *ShareMode* and *ShareProb* are provided in order to model such data sharing between the successive tasks of a job. *ShareMode* is used to determine the type of data (data read, or data updated) that is shared between two successive job tasks. Four sharing modes are possible: *Read–Read*, *Read–Write*, *Write–Read*, and *Write–Write*. In *Read–Read* sharing, each page read by task T_i of job J has a probability *ShareProb* of being read by its successor task T_{i+1}; the other modes are defined similarly. Of course, the amount of data shared between any two successive tasks is limited by the number of pages accessed by the smaller of the two tasks.

4.2.3. Modeling Open-Ended Transaction Execution

The Rule Manager is informed by the Transaction Manager of the data accessed by each External Task. It uses this information, plus its knowledge of the internally generated workload, to coordinate active transaction execution with the Transaction Manager.

Let us consider the case of a nearly-completed External Task as an example. When an External Task E has completed its reads and updates successfully, and has reached the point where it can commit, the Rule Manager is invoked to check whether E's transaction can be committed. Depending on the input parameters, the Rule Manager responds in one of the following ways:

(1) If the Rule Manager determines that an additional task is to be coupled with E's transaction, it initializes a new task which is appended to E's transaction. In our example, the new task would be a Checker triggered by E.

(2) If the Rule Manager determines that an additional task must be scheduled as part of a *different* transaction, it initializes a new task and directs the Transaction Manager to execute the new task as an independent transaction. In this case, E's transaction is allowed to commit, while E's job remains active. Again, the new task would be a Checker in our example.

(3) If the Rule Manager decides that E's job has completed its execution, it informs the Transaction Manager that E's transaction may be committed, as no new work has been generated due to rules being activated.

The Rule Manager and the Transaction Manager interact in a similar fashion when either a Checker or an Action reaches the point where it can commit.

4.2.4. Modeling Restarts

When a transaction is aborted, the Transaction Manager is responsible for informing the Rule Manager of the abort. When it receives the information about an abort, the Rule Manager directs the Transaction Manager to restart the first task which was part of the aborted transaction. In our performance model, since decisions are made probabilistically, this restart mechanism can unfortunately lead to a bias against long jobs. Consider a system where there is high contention for data and all jobs are strictly coupled. Suppose a job J begins with a External Task task E, goes through a Checker task C, and is in Action task A when it is aborted. It will then be restarted beginning at task E. Let us assume that it completes task E again and is then aborted in task C. It will then once again begin with task E. If the decisions to schedule C and A are made entirely on the basis of probabilities, it is more likely that Job J will ultimately complete execution if it consists of a single task rather than two or three S-coupled tasks. Thus, there will be a tendency to favor shorter jobs over longer ones, i.e., restarts will keep occurring until the job eventually happens to become a short job. In order to avoid this bias, a parameter *RepeatOnRestart* is included for the Rule Manager (see Table 2). If *RepeatOnRestart* is set to 1.0, then the entire sequence of tasks of an aborted transaction will be repeated on restart. *RepeatOnRestart* settings are allowed to range from 0 to 1, thus representing varying likelihoods that a restarted transaction will go through the exact same sequence of tasks that the aborted transaction did.

4.3. THE SOURCE MODULE

The applications of active DBMSs suggest that an open queueing model be used: e.g., in a large-scale control application, ETs may arrive at the DBMS in the form of data gathered by a large number of sensors. The Source module is the component responsible for modeling the external workload for the DBMS (i.e., it generates the External Task of each job). Each External Task is sent to the Transaction Manager in the form of an initial transaction.

Table 3 summarizes the key parameters of the external workload model. Jobs arrive at the system in the form of a Poisson process with arrival rate *ArrRate*.[1] The workload model used by the Source characterizes External Tasks into classes, just as RMTs are classified at the Rule Manager. *NumClasses* specifies the number of different classes of External Tasks. The remaining parameters are similar to those used to model the RMT workload, as described in Section 4.2.2.

4.4. THE TRANSACTION MANAGER MODULE

The Transaction Manager is responsible for accepting transactions from the Source as well as the Rule Manager and modeling their execution. A read access by a transaction begins with a concurrency control request to get access permission. Once the access is granted, a read request for the page is sent to the Resource Manager. If the data page is found in the main memory buffers, no I/O is required;

Parameter	Meaning
Overall Arrival Pattern Parameters	
ArrRate	Mean exponential arrival rate of External Tasks
NumClasses	Number of classes of External Tasks
Per-Class Parameters	
ClassProb	Probability of this class
FileCount	Number of files accessed
$FileProb_i$	Access probability for file i
$NumPages_i$	Average number of file i pages read
$WriteProb_i$	Write probability for file i pages
PageReadCPU	CPU time for processing a page of data
WriteCPUFraction	Fraction of read CPU time required to process a write

Table 3: External Workload Model Parameters.

[1]Our model is actually somewhat more general than indicated here; it supports multiple arrival processes, and these can each be either exponential or deterministic.

otherwise, a disk I/O to read the page (possibly accompanied by the flushing of a dirty buffer page) is scheduled. Once the data page is available in the buffer pool, a period of CPU usage follows for processing the page. Write requests are handled similarly.

The input parameters of the Transaction Manager module are listed in Table 4. In addition to modeling the execution of transactions, the Transaction Manager controls the load of the system by limiting the number of concurrently active jobs via the parameter *MaxActiveJobs*. If a new job is received by the Transaction Manager when there are already *MaxActiveJobs* jobs in the system, the new job is blocked until one of the active jobs completes execution. Load control at the job level rather than at transaction level is appropriate for an active system, as the amount of "load" represented by a transaction depends upon the coupling modes employed to couple the job's tasks. A delay modeled by *LoggingDelay* represents the time required to flush log records at the end of each transaction. Finally, the Transaction Manager also handles transaction commits and aborts.

4.5. THE RESOURCE MANAGER MODULE

The Resource Manager controls the physical resources of the DBMS, including the CPU, the disks and the buffer pool in main memory. It provides services to both the Transaction Manager and the Concurrency Control Manager. The parameters of the Resource Manager are summarized in Table 4. The DBMS has *NumDisks* disks plus one CPU. The CPU service discipline is processor sharing. Each of the disks has its own queue, which it serves in a FIFO manner; the Resource Manager assigns a disk to serve a new request randomly, with all disks being equally probable, so the I/O model assumes that the files stored in the DBMS are evenly balanced across the disks. Disk access times for the disks are uniform over the range [*MinDiskTime*, *MaxDiskTime*].

The buffer pool model consists of a set of *BufferPoolSize* page frames. Each page frame has a "dirty" flag associated with it to indicate whether its data has been updated since it was read into main memory. If a request to read page P is received from the transaction manager, the buffer manager checks whether page P is in the buffer pool. If page P is present, no disk access is required. If page P is not found, the buffer manager searches for the least-recently-used, non-dirty page which can be replaced by P. If such a clean page C is available, a disk I/O is scheduled to read page P into the buffer frame occupied

Parameter	Meaning
	Transaction Manager Parameters
MaxActiveJobs	Maximum number of concurrently running jobs
LoggingDelay	Per-transaction delay caused by flushing a commit record
	Resource Manager Parameters
NumDisks	Number of disks
MinDiskTime	Minimum disk access time
MaxDiskTime	Maximum disk access time
BufferPoolSize	Number of page frames in the buffer pool
	Concurrency Control Manager Parameters
CCReqCPU	CPU time required to process a concurrency control request

Table 4: Remaining Parameters of the Model.

by C. If no clean page is found, a disk write is scheduled to write back the least-recently-used, dirty page in addition to the read that is scheduled for page P. In either case, after P is read into the buffer pool, the read request is processed by the CPU.

4.6. THE CONCURRENCY CONTROL MANAGER MODULE

The Concurrency Control Manager captures the semantics of the concurrency control algorithm of choice. As was illustrated in Figure 1, the Concurrency Control Manager is responsible for handling concurrency control requests made by the Transaction Manager. These include read and write access requests, requests to get permission to commit a transaction, and several types of concurrency control management requests to initialize and terminate transactions. The Concurrency Control Manager has a variable number of parameters. One parameter, *CCReqCPU*, specifies the amount of CPU time required to process a read or write access request; this parameter is present for all algorithms. Additional parameters can be introduced on a per-algorithm basis as needed.

For the experiments described in this paper, a variation of the Wound-Wait (WW) algorithm described in [Rose78] is used for concurrency control. We chose this algorithm because it is relatively easy to model, and its timestamp-based conflict resolution strategy makes it possible to introduce an interesting modification to the algorithm, as discussed below. Also, the WW algorithm can be modified later to allow priorities to be associated with transactions in order to model time constraints, as proposed in [Abbo88, Daya88a, Stan88].

In the WW algorithm, locks are acquired and released in a two-phase fashion [Gray79]. Deadlock is prevented via the use of timestamps rather than by maintaining waits-for information and then checking for deadlocks. In the original algorithm, each transaction is numbered according to its initial startup time, and younger transactions are prevented from making older ones wait. If an older transaction requests a lock held by a younger transaction, which would lead to the older transaction waiting, the younger transaction is "wounded", i.e., it is restarted. Younger transactions are permitted to wait for older transactions, however. Deadlocks are impossible because any cycle of waiting transactions would have to include at least one instance where an older transaction is waiting for a younger one that is blocked as well, and this is not allowed.

In order to model the relationship between the transactions that make up a job, the following variation was introduced in the WW timestamp allocation scheme. Let Job_i consist of transactions $T_{i1}, ..., T_{im}$, and let Job_j consist of transactions $T_{j1}, ..., T_{jn}$. If Job_i starts before Job_j, then all of Job_i's transactions T_{ik}, k = 1,..,m, will have *earlier timestamps* than any of the transactions of Job_j. Intuitively, the purpose of this modification is to favor tasks of jobs that arrive early over those of later arrivals, thus introducing an element of *job-based priority* to the concurrency control algorithm. As a result of this modification, tasks of older jobs are more likely to obtain locks and less likely to be aborted than tasks of younger jobs. (Note that this timestamp allocation scheme is independent of both the type of the coupling between the tasks and the time ordering of the individual tasks of different jobs.)

5. EXPERIMENTS AND RESULTS

In this section, we present performance results for the four types of RMT coupling semantics described in Section 3 under various assumptions about the database size, the number of data pages accessed by each task of a job, the number of tasks per job, and the extent of data sharing between the tasks of a job. Our performance results are preceded by a discussion of the performance metrics of interest and the parameter settings used to obtain the results presented here.

5.1. SYSTEM PERFORMANCE METRICS

In order to focus on the impact of using different semantics for transaction coupling on system performance, we examine the expected job response time as a function of the job arrival rate [2] for each of the four types of coupling modes. Job response time is computed by subtracting the startup time of the ET of the job from the time at which its last RMT finally completes successfully. In order to obtain a large sample of job executions, each experiment was run until a total of approximately 4500 jobs completed. Other statistical information was also gathered in the course of each simulation, including the maximum arrival rates at which the system remained stable, the number of aborts and the particular tasks in which they occur, the utilization of the CPUs and disks, the number of buffer pool hits and misses, and the number of jobs running concurrently.

5.2. PARAMETER SETTINGS

Each experiment consists of a "base" simulation and a set of simulations in which certain parameters are varied. Table 5 gives the values of the key simulation parameters in the base experiments. The

Parameter	**Base Setting**
NumFiles	1
FileSize	500 pages
ArrRate	varied from 0.2 jobs/sec to 8 jobs/sec
NumClasses	1
NumPages	5
WriteProb	1/3
PageCPU	8 ms (for reads); 2 ms (for writes)
MaxActiveJobs	15
LoggingDelay	10 ms
NumDisks	2
MinDiskTime	10 ms
MaxDiskTime	30 ms
BufferPoolSize	50 page frames
CCReqCPU	0 ms
ExtCheckProb	2/3
ActionProb	1/2
ActCheckProb	2/3
RNumPages	5
RWriteProb	0 (for Checkers); 1/3 (for Actions)
RPageCPU	8 ms (for reads); 2 ms (for writes)
ShareMode	No Sharing
ShareProb	0.0
RepeatOnRestart	1.0

Table 5: Base Parameter Settings.

[2]Since an open queueing model is used to represent the system, throughput alone is not a sufficient performance metric.

database is modeled simply as a single file, consisting of 500 data pages. Jobs arrive at the system in a single exponential stream; the job arrival rate is varied in each experiment. Each task accesses 5 pages of data on the average. External Tasks and Actions update each page that they access with a probability of 1/3, while Checkers are read-only tasks. It takes an average of 8 milliseconds of CPU time to process each data page read, and an average of 2 additional milliseconds to process an update. The maximum number of jobs that may run concurrently is set to 15. There are two disks available for I/O, and each disk has an average access time of 20 milliseconds. The buffer pool can hold up to 50 pages in memory at a time. The concurrency control CPU overhead is assumed to be negligible compared to the CPU time required for page processing. Since the overhead associated with transaction startup (which corresponds to the startup time for a lightweight process) is typically on the order of a fraction of a millisecond [Lisk87], this cost is not included in the model. However, a commit record logging delay of 10 milliseconds is added to each transaction at commit time. The probability that an External Task (or an Action) is followed by a Checker is set at 2/3, and the probability that a Checker is followed by an Action is set at 1/2. Thus, approximately one third of all jobs consist of just one task, another third of two tasks, and the remainder of more than two tasks; the mean number of tasks per job is 2.33. Data is not shared between the adjacent tasks of a job. Finally, the restart parameter is set so that an aborted transaction always repeats its original sequence of tasks.

In order to isolate the effects of using each of the four coupling options, each simulation had only one type of coupling between the tasks of a job. For each arrival rate, four simulations were run: one in which all jobs consisted of S-coupled tasks, one in which only EC-coupling was allowed, one in which only CA-coupling was allowed, and one in which only N-coupling was allowed. In addition to these four simulations, we ran one simulation for each arrival rate to obtain the expected job response time in the absence of concurrency control in order to provide a performance bound against which to compare the various coupling modes. In each of our graphs, a curve labeled No-CC shows the results of this simulation.

In Experiment 1, we studied the effect of varying data contention by changing the database size. The effect of varying task complexity (i.e., the number of pages accessed by a task) was studied in Experiment 2. In Experiment 3, the impact of varying the number of tasks in a job on system performance was investigated; finally, in Experiment 4, we varied the amount of data sharing between the tasks of a job. In the remainder of this section, we first discuss the results of our base experiment, and we then describe Experiments 1-4.

5.3. THE BASE EXPERIMENT

Figure 2 shows the expected job response times for the parameter settings shown in Table 5. At low arrival rates, the response time of jobs does not vary significantly with the type of coupling. As the arrival rate is increased, all of the curves begin to move upwards. The S-coupling curve separates from the others first, while the EC-coupling, CA-coupling, N-coupling, and No-CC curves remain fairly close to one another for most arrival rates. Note that, for each type of coupling, job response time grows dramatically when the job arrival rate is increased beyond a certain threshold value. The system becomes very sensitive to changes in arrival rate when this threshold is approached; a small increase in job arrival rate in this region of operation causes the system to enter a state of saturation in which it becomes unstable. The rise in job response time is so dramatic in this region that some of the higher response time values do not fit into the graph and are represented (in all experiments) by extrapolated lines at the right end of each curve. Figure 2 shows that the threshold at which this phenomenon occurs is reached earliest by the S-coupled jobs. EC-coupled jobs reach the point of saturation next, and CA-coupled jobs reach their threshold just before N-coupled jobs.

The elements contributing to the expected response time of a job are the costs of reading and updating pages and the costs of resolving concurrency control conflicts. The workload parameter settings for the base experiment guaranteed that, on the average, the number of pages read and written by a job does not vary with the type of coupling used in the system (in the absence of restarts). As explained in Section 5.2, CPU costs for transaction startup are assumed to be negligible, and transaction commit costs (in the form of the 10 milliseconds logging delay) are small relative to the overall response times observed. Thus, the differences in job response times for the four types of coupling are almost entirely due to concurrency control costs. In order to understand how concurrency control affects performance, we must examine two factors: blocking delays and restart rates. A quantitative display of these factors is given in the next two figures. The expected total amount of time a job has to spend waiting for locks is presented in Figure 3, whereas the expected number of times that a transaction is restarted is presented in Figure 4. Both figures indicate that the manner in which tasks are coupled to form transactions has a dramatic impact on the

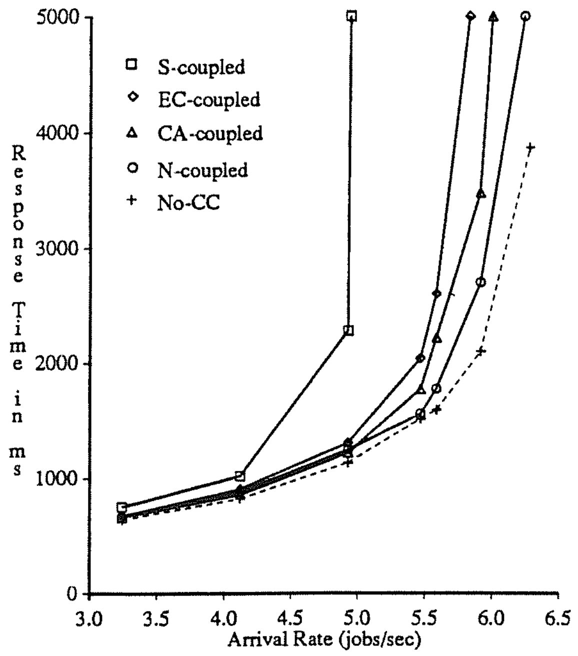

Figure 2: Database Size = 500.
(TaskSizes = [5, 5, 5], No Sharing)

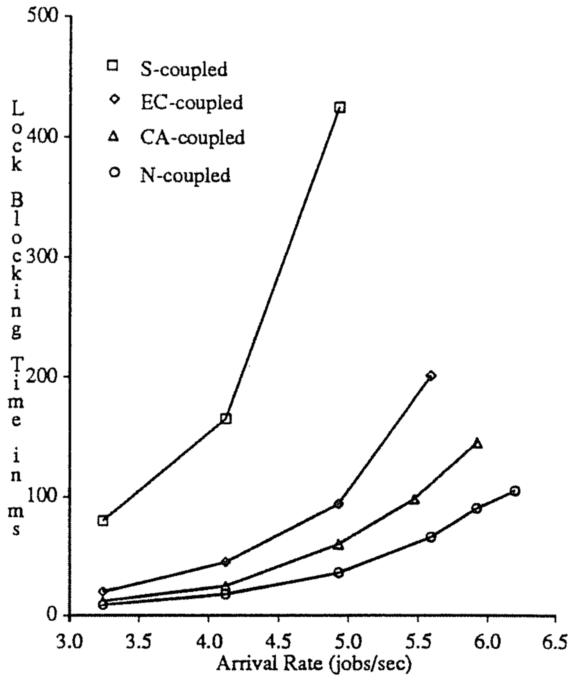

Figure 3: Lock Blocking Time per Job, DBSize = 500.
(TaskSizes = [5, 5, 5], No Sharing)

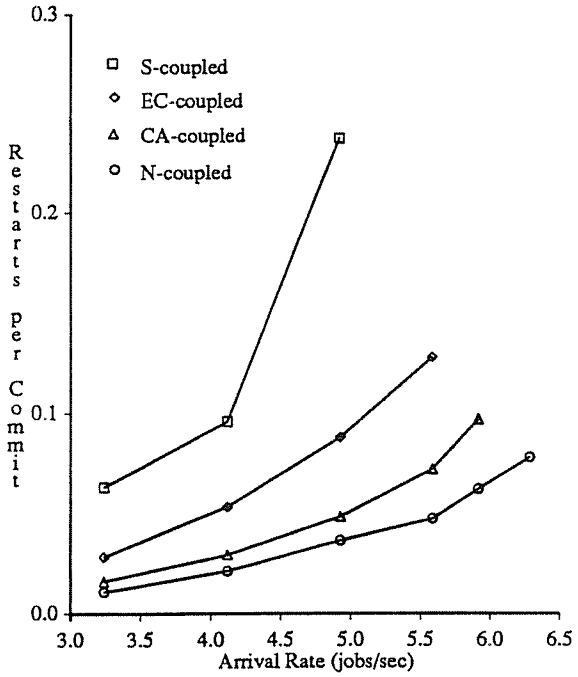

Figure 4: Restarts/Commit, DBSize = 500.
(TaskSizes = [5, 5, 5], No Sharing)

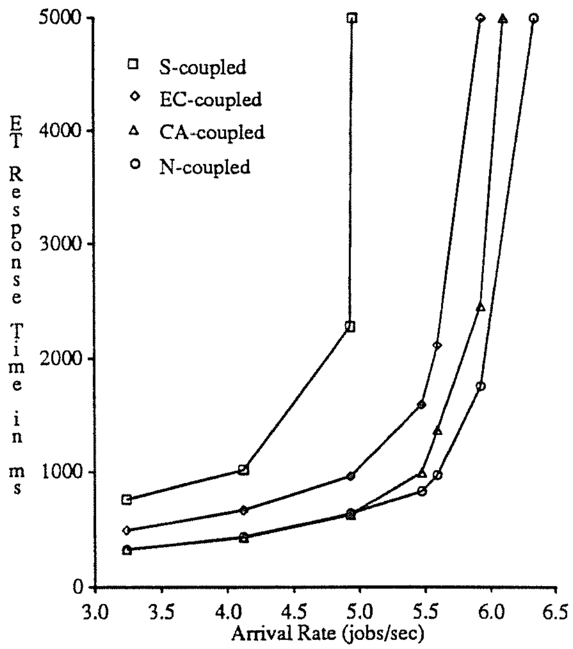

Figure 5: ET Response Time, DBSize = 500.
(TaskSizes = [5, 5, 5], No Sharing)

amount of a time that jobs spend redoing work (due to restarts) or waiting for locks to be granted.

In order to understand how the locking behavior of jobs depends on their coupling mode, consider a job that consist of three tasks: an ET, a Checker, and an Action. When the job is S-coupled, it holds all of its locks until the Action task is completed. With EC-coupling, the job holds its locks until the end of the Checker task, releasing them before the Action task begins. A CA-coupled job releases its locks immediately after the completion of the External Task. Such a job holds the Checker locks (which are all read locks) and the Action locks until it commits. When the job is N-coupled, it releases locks when each task is completed. Two issues related to locking affect the expected blocking time of the four coupling alternatives. The first is the *average duration* for which locks are held by a transaction. The longer a transaction holds on to its locks, the longer other transactions will have to wait to lock a data item. The second issue is the *type of lock* held. Since External Tasks consist of updates as well as reads, both S-coupled and EC-coupled jobs hold write locks for long durations, while CA-coupled jobs hold only read locks for long durations. Since read locks can be shared, while write locks cannot, holding read locks for longer durations does not have as much of a negative impact on job response time as holding write locks. This explains the differences in the expected blocking time of EC-coupled and CA-coupled jobs in Figure 3. Although both modes lead to the same expected transaction size here, EC-coupled jobs hold write locks longer and thus cause longer blocking delays. When write locks are held for long durations, as in EC-coupling or S-coupling, increased data contention results in a significant increase in job response time. S-coupling has the longest blocking time because all of the locks acquired by a job are held until the very last task of the job completes execution.

Another factor related to the performance impact of concurrency control is transaction restarts. Restarts increase resource utilization and lengthen the execution path of transactions. Figure 4 shows that the average number of restarts per commit is low in this experiment. In most cases, there is less than one restart per ten commits. Recall that our Wound-Wait based concurrency control algorithm tends to restart younger tasks more often than it restarts older tasks. These two facts lead us to expect that the increase in resource utilization due to restarts will not be significant — and our measurements show that this increase is indeed small (less than 3% in most cases). However, the increase in the expected number of locks acquired by a transaction had a major impact on job response time. A significant proportion of the time "wasted" due to restarts was spent waiting again for locks rather than utilizing the physical resources of the system. It is also important to note that restarted transactions represent a different amount of "wasted" time for each type of coupling. In the case of S-coupling, restarting a transaction is equivalent to restarting an entire job, since each job consists of only one transaction. In contrast, in the case of N-coupling, the amount of time wasted by a restart is bounded by the time taken by a single task of a job.

So far we have focused on how job response time is affected by the different coupling modes. Another interesting performance measure is the expected time that users must wait for their tasks to complete. This metric (called the ET response time of the system) is presented in Figure 5 for the four coupling modes. This measure represents the user's point of view of the performance of the active DBMS, whereas the job response time represents the system administrator's point of view. For the entire range of arrival rates, coupling RMTs with ETs causes a significant increase in the response time observed by a user. Even for an arrival rate of 4 jobs per second, where there is almost no difference in the expected overall response time of a job, the expected response time of an S-coupled ET is more than twice that of an ET that has not been coupled with its RMTs (i.e., where CA-coupling or N-coupling has been used).

To summarize, then, the results of the base experiment reveal that the way in which transaction boundaries are set in an active DBMS may have a significant impact on both the quality of service experienced by its users and the location of the saturation point of the system. The results also indicate that the key factor that influences job response time in the base experiment is data contention rather than resource contention. For most job arrival rates considered, jobs that were S-coupled performed significantly worse than jobs that did not couple all of their rule management activity to the External Task. Note that, from a performance point of view, augmenting user transactions with code to check conditions (e.g., violations of integrity constraints) and to invoke actions in response to satisfied conditions, is basically the same as S-coupling. We can thus conclude from the results of the base experiment that not only can active DBMSs simplify the task of application programming, but they can also lead to performance gains.

5.4. EXPERIMENT 1: THE EFFECT OF DATA CONTENTION

In this experiment, we examine job response times for the four coupling types as the size of the database is changed.[3] Three database sizes (750, 500, and 250 pages) are studied, and the buffer pool size is kept at 10% of the database size in each case. Each task of a job accesses 5 pages on the average, as in the base experiment.

Figure 6 shows the expected job response times for a database consisting of 750 pages, where the buffer pool can accommodate 75 such pages. Figure 7 presents the response time results obtained when the database size is decreased to 500 pages and the buffer pool size is 50 pages. (Figure 7 is the same as Figure 2, but is repeated here for comparison.) In Figure 8, the job response times for a database size of 250 pages and a buffer pool size of 25 pages are shown. In Figure 9, we compare the job response times for an arrival rate of 5 jobs/second for the three different database sizes. Since in the 250-page case, the system becomes unstable for S-coupled jobs at an arrival rate close to 4 jobs/sec, we ran further experiments with database sizes of between 500 and 250 pages specifically in order to obtain the third point for the S-coupled case.

In Figure 6, where the database size is 750 pages, the response time curves for the four types of coupling remain close to each other for most arrival rates, and the system remains stable for arrival rates as high as 6.2 jobs/sec. As the size of the database is decreased to 500 pages in Figure 7 and then to 250 pages in Figure 8, the curves for S-coupled and EC-coupled jobs move away from the other curves at lower arrival rates than before. The separation between these curves and the curves for CA-coupling and N-coupling also grows, and increased data contention causes the system to become unstable more rapidly with increasing arrival rates. In particular, S-coupled jobs make the system unstable at arrival rates of about 4 jobs/sec when the database contains 250 pages, but when the database size is 750 pages, 50% higher arrival rates can be sustained by the system without losing its stability for S-coupled jobs. The reduction in the size of the buffer pool from Figure 6 to Figure 7 and again from Figure 7 to Figure 8 may at first seem likely to contribute to the positioning of the response time curves in the three cases. However, a closer examination reveals that the No-CC curves in the three figures coincide almost exactly for all but the highest arrival rates shown. Even at arrival rates (between 6 and 6.5 jobs/second) that cause the system to enter a state of saturation when concurrency control is in use, the spread between the No-CC curves in

[3]Varying the database size in this manner can be thought of as a way to vary the extent to which "hot spots" exist in a larger database, as shown in [Tay85].

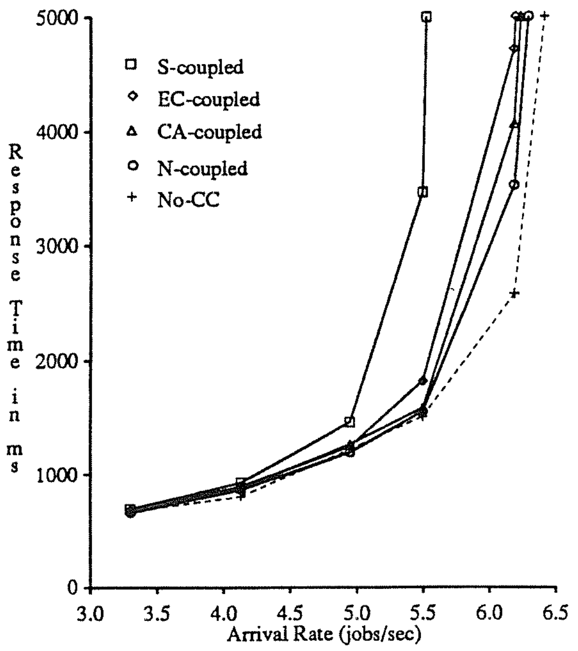

Figure 6: Database Size = 750.
(TaskSizes = [5, 5, 5], No Sharing)

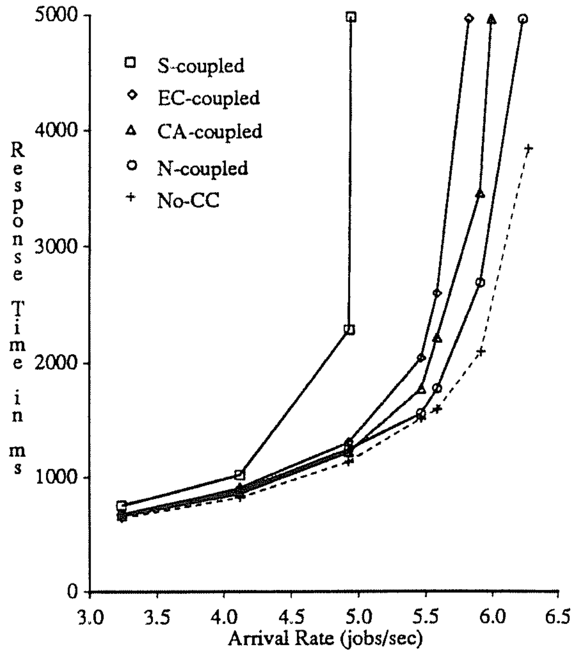

Figure 7: Database Size = 500.
(TaskSizes = [5, 5, 5], No Sharing)

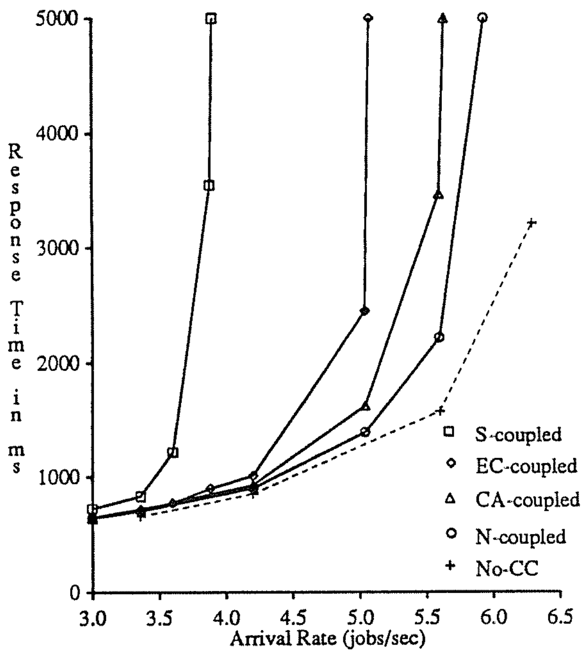

Figure 8: Database Size = 250.
(TaskSizes = [5, 5, 5], No Sharing)

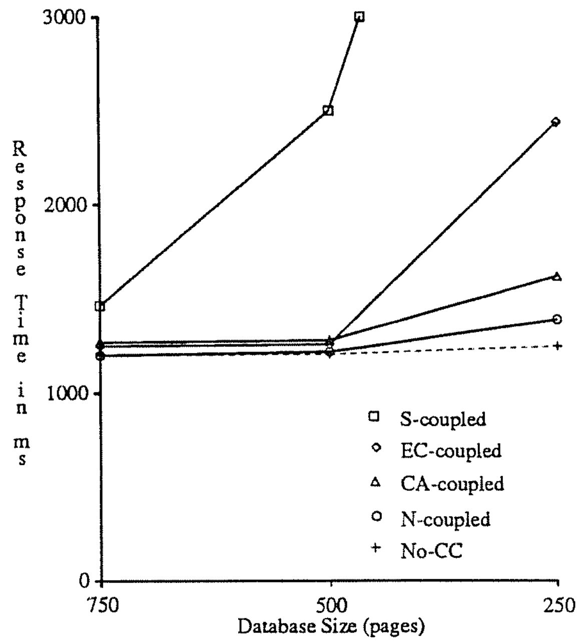

Figure 9: Arrival Rate = 5 Jobs/sec.
(TaskSizes = [5, 5, 5], No Sharing)

the three figures is less than 10% of the minimum response time. Thus, changing the buffer pool size does not bias the results shown in Figures 6-8 significantly, and buffer hits may be ignored as a factor while comparing the three sets of curves.

Figure 9 illustrates the increasingly important impact of task coupling semantics on system performance as data contention increases. S-coupling results in the worst performance, and rapidly leads the system into instability. For low levels of data contention, the differences in performance between the other three types of coupling are negligible. As data contention increases, however, N-coupling leads to better performance than CA-coupling, while CA-coupling performs better than EC-coupling. Since a temporary period of high data contention can easily occur even in a passive DBMS (e.g., when there is a "hot spot" in the data), it should be clear that it is important for performance that the weakest type of coupling permitted by each rule's semantics be selected and used in the rule's definition.

5.5. EXPERIMENT 2: THE EFFECT OF TASK COMPLEXITY

Tasks associated with jobs in active database systems will have application-dependent data access characteristics. External Tasks, Checkers and Actions may vary in the relative amounts of data that they need to access. Also, their data access characteristics will be influenced by the degree to which the physical design of the passive database supports the queries involved in the rules. In order to study the performance impact of variations in task size, simulations with three combinations of task sizes were run. All of these experiments were conducted on a 500-page database.

In the first set of simulations, all of the tasks of each job access 5 pages each, on the average, as in the base experiment. This set of parameters is abbreviated SSS (for Short External Tasks, Short Checkers, and Short Actions). All other simulation parameters are kept exactly as in the base experiment. Figure 10 (which is the same as Figure 2) shows the expected response time of jobs with the SSS task pattern. The second pattern of tasks studied was SLS (Short External Tasks, Long Checkers, and Short Actions). External Tasks and Actions again access 5 pages each, while Checkers now access 20 pages each on the average. Figure 11 shows the response times of jobs with SLS task patterns. Figure 12 presents job response times for jobs having an SLL task pattern — where External Tasks are short (5 pages each) while both Checkers and Actions are long (20 pages each). Figure 13 shows the maximum sustainable arrival rates at which the system remains stable for the SSS, SLS and SLL task patterns.

Figures 10-12 show that significant performance changes occur when the task access patterns are changed. In Figure 10, all of the curves except for the S-coupled one are quite close to one another. When the size of Checkers increases, in Figure 11, the differences between the performance of the four coupling types become much more pronounced. The slope of the curves increases more rapidly, and the maximum sustainable arrival rates become smaller. The difference between the No-CC curve and the N-coupled curve is still relatively minor here, though. When both Checkers and Actions are large, as in Figure 12, the response times curves become very steep even at low arrival rates. An interesting phenomenon in the SLL case is that, for the first time, CA-coupling performs worse than EC-coupling. The reason for this is that, in this case, External Tasks are short whereas Actions are long. Thus, if we consider jobs having three tasks or less (which constitute most of the jobs in the simulation), EC-coupling involves holding locks for a considerably shorter time than CA-coupling. Another feature of the SLL case is that even the N-coupled curve is quite far from the No-CC curve, indicating that the increased sizes of Checkers and Actions cause significant concurrency control conflicts even when job tasks are uncoupled. Figure 13 reveals that, as RMTs become more complex, the relative impact of the coupling mode on performance increases. For example, the ratio of the maximum sustainable arrival rate for N-coupled jobs to the maximum sustainable rate for S-coupled jobs goes up from 1.25 in the SSS case to 1.75 in the SLL case. These results indicate that system performance can be affected significantly by both the design of the RMTs of a rule (i.e., the relative complexity of Checkers and Actions) and the degree to which their data access needs are supported by the physical organization of the passive database.

5.6. EXPERIMENT 3: THE EFFECT OF TRIGGERING PROBABILITIES

In Experiment 3, the probabilities of executing both Checkers and Actions were increased to study the effect of increased job size (in terms of the number of tasks per job) on system performance. By increasing *CheckProb* from 2/3 to 4/5 and *ActionProb* from 1/2 to 4/5, the mean number of tasks per job was increased from 2.33 to 5. (Note that the transaction boundaries imposed by the different coupling semantics on a job consisting of exactly five tasks are shown in Table 1 in Section 3.) The database size was kept fixed at 500 pages. For comparison purposes, Figure 14 again shows the response times of jobs having an SSS task pattern when the average number of tasks per job is 2.33. Figure 15 shows the job

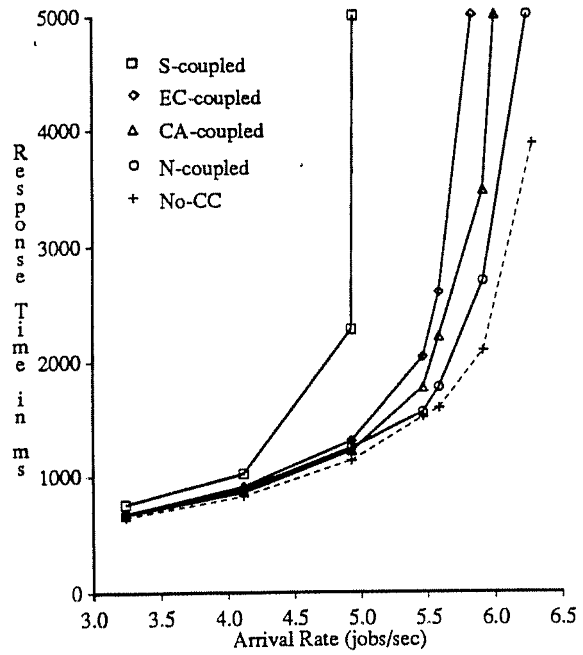

Figure 10: S-S-S Task Pattern
(TaskSizes = [5, 5, 5], DBSize = 500)

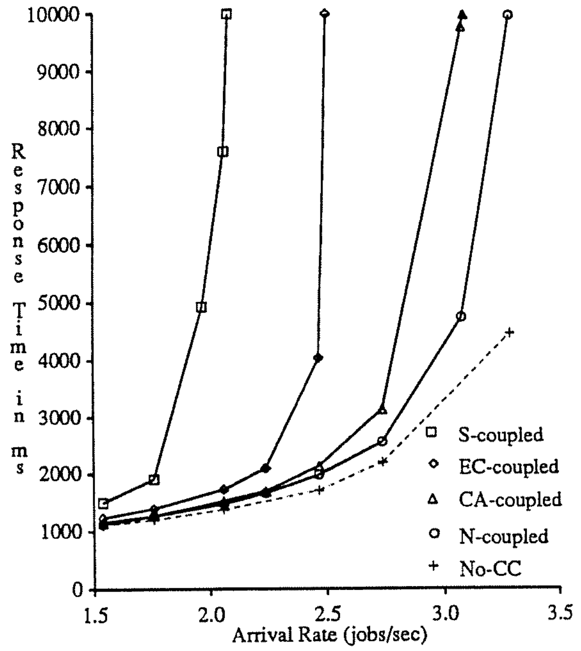

Figure 11: S-L-S Task Pattern
(TaskSizes = [5, 20, 5], DBSize = 500)

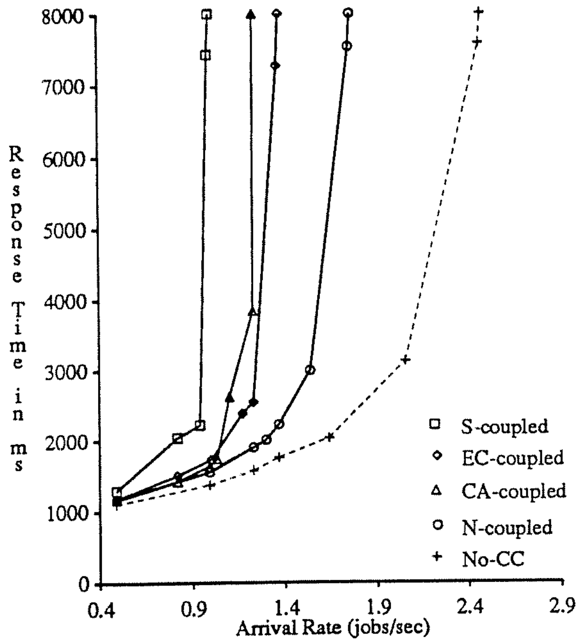

Figure 12: S-L-L Task Pattern
(TaskSizes = [5, 20, 20], DBSize = 500)

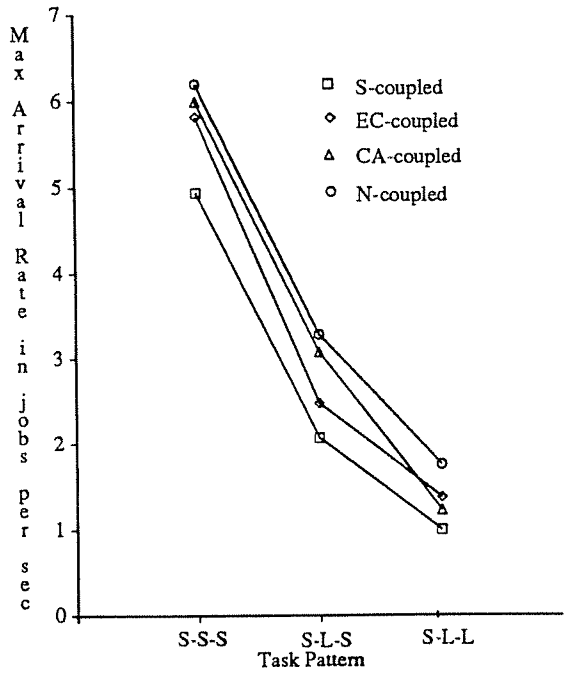

Figure 13: Maximum Sustainable Arrival Rates
(DBSize = 500)

response times of SSS pattern jobs when jobs have 5 tasks on the average. Figure 16 shows the response times of jobs having an SLS pattern, with an average of 2.33 tasks per job. In Figure 17, jobs again have 5 tasks, and the task pattern is now SLS.

Clearly, when job size is increased, the performance of S-coupling deteriorates very rapidly relative to all other coupling modes. Consider a job with six tasks: In the S-coupling case, no locks are given up until all six tasks have completed; in the EC-coupled and CA-coupled cases, locks are given up after each pair of tasks finishes. (Note that a six-task, S-coupled job will access approximately 75 data pages in the SLS case, thus locking up 15% of the entire database.) A second feature of longer jobs is that the performance of EC-coupled jobs and CA-coupled jobs is quite similar, especially in Figure 15 where all job tasks are short. The reason for this is that both of these coupling modes involve releasing locks after alternate tasks of a job, as mentioned above. When the tasks are short and there are many tasks per job, the effect of the type of lock (read versus write) that is held longer is diluted to some extent.

5.7. EXPERIMENT 4: THE EFFECT OF DATA SHARING BETWEEN TASKS

As discussed earlier, it is likely that there will be some degree of data sharing between successive pairs of tasks of a "real" job. In Experiment 4, the extent of data sharing between successive tasks is varied. For comparison purposes, the results of the base experiments in which there is no enforced data sharing between tasks are shown again in Figure 18. (Since data pages are chosen randomly from the database, there may be some inherent sharing of data between tasks even in this case, but it is rare enough to be ignored.) Figures 19 and 20 show the results of experiments with Read-Read sharing, where the read-set of each task of a job overlaps with the read-set of the next task of the same job. Three *levels* of data sharing are studied: no sharing (in the base experiment), 50% sharing, and 100% sharing. Recall that the level of sharing represents the probability that any given page will be accessed by two consecutive job tasks, subject to the constraints of the relative size of the tasks. (While the *average* size of the tasks is 5 pages each, the *actual* size of different tasks may vary between 3 and 7 pages. Thus, if task T_i of a job accesses 3 pages and its successor task T_{i+1} accesses 7 pages, a maximum of 3 pages can be shared between the two tasks.) Figure 21 shows the maximum arrival rates that the system can support for an expected job response time of 2 seconds for the four different types of coupling as the level of sharing is changed. (The points on Figure 21 were interpolated from the data in Figures 18-20.)

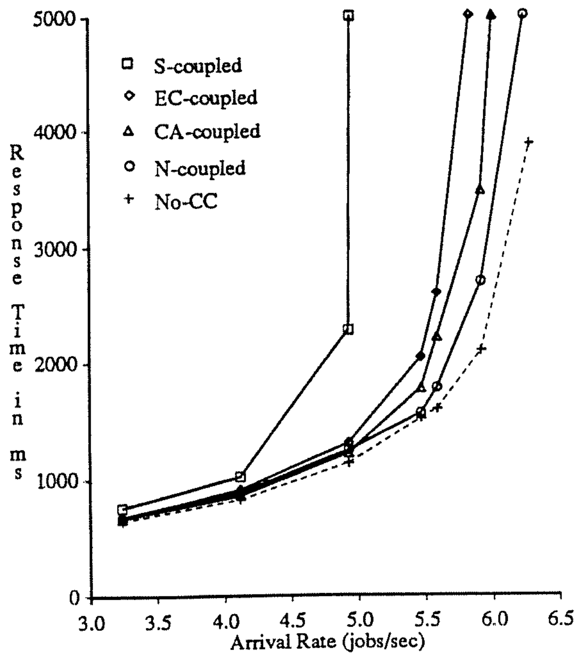

Figure 14: Mean Number of Tasks per Job = 2.33 (TaskSizes = [5, 5, 5], DBSize = 500)

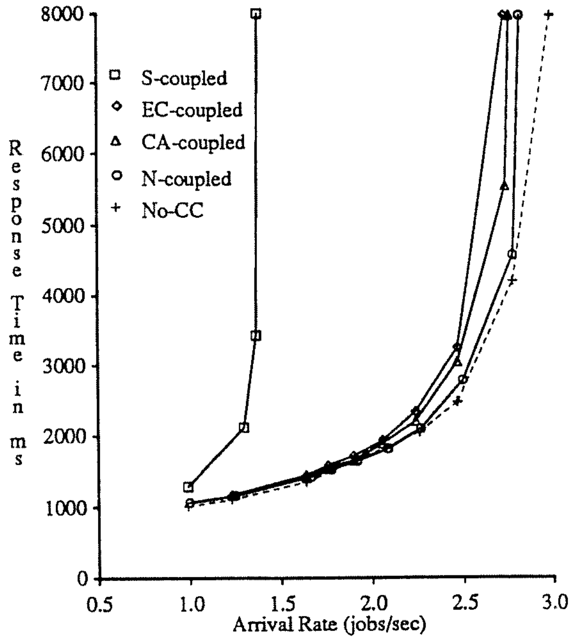

Figure 15: Mean Number of Tasks per Job = 5.0 (TaskSizes = [5, 5, 5], DBSize = 500)

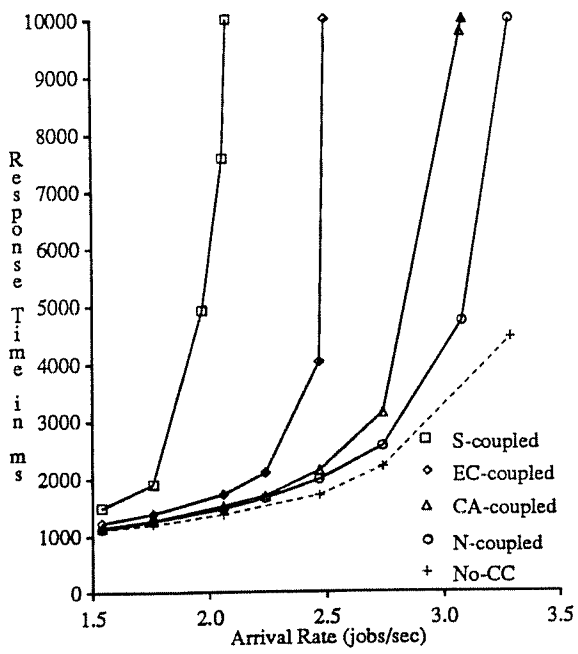

Figure 16: Mean Number of Tasks per Job = 2.33 (TaskSizes = [5, 20, 5], DBSize = 500)

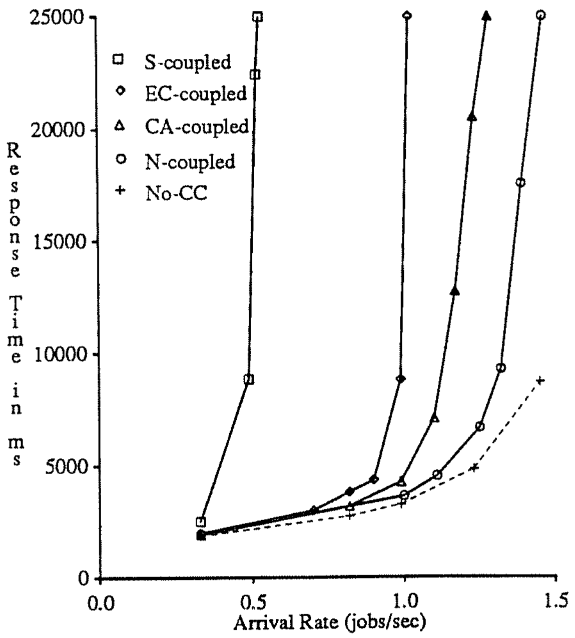

Figure 17: Mean Number of Tasks per Job = 5.0 (TaskSizes = [5, 20, 5], DBSize = 500)

When there is no data sharing between job tasks, the S-coupled curve diverges from the others at an arrival rate of about 4.2 jobs/second, while the other three curves remain close to one another. As the level of sharing is increased from 0 to 50%, the curves for EC-coupling and CA-coupling move still closer, almost merging with each other. The S-coupling curve also moves somewhat closer to the other three, but S-coupled jobs still perform significantly worse than the other types of coupling. Figure 20 shows that when *all* of a job's tasks access the same data, the performance of all four types of coupling becomes very similar. All four curves remain close together for arrival rates of up to 6 jobs/second, and even S-coupling does not saturate the system until the arrival rate reaches 6.75 jobs/second (which is a 35% improvement over the no-sharing case). Figure 21 shows that at low levels of data sharing, the maximum job arrival rate that the system can support in order to provide a given expected response time (2 seconds in this case) varies significantly with task coupling semantics. At high levels of data sharing, however, this difference becomes negligible.

Data sharing affects job response time in two ways. If data is shared between successive tasks, more buffer pool hits are likely. Also, fewer locks need to be obtained in the S-coupling, EC-coupling and CA-coupling cases, as the locks acquired in one task of a job within a transaction do not need to be re-acquired in the succeeding phase. An examination of the results for buffer pool hit ratios, disk utilization, and number of locks avoided by data sharing (not shown here) indicate that, while both buffering and locking factors affect the results, the relative performance of the four types of coupling is affected most by the fact that fewer locks are required (except for N-coupled jobs) as data sharing is increased. In N-coupling, locks are released at the end of every task, so the amount of locking activity is unaffected by data sharing; this is why N-coupling does not show as much improvement as the other types of coupling when the amount of data sharing is increased. The main point illustrated by this experiment is that coupling semantics affect performance significantly even when half the data of each task is shared with the subsequent task. It is only when the ET and the RMTs of a job share data to a very large extent that the coupling semantics lose their relevance to system performance.

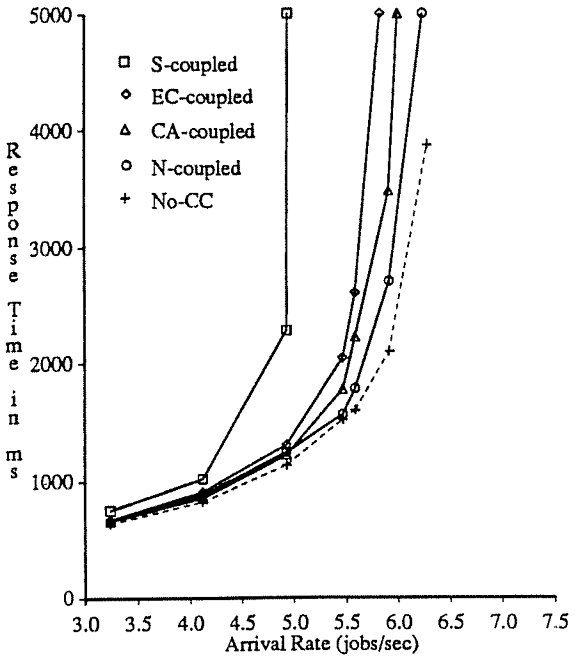

Figure 18: No Data Sharing
(TaskSizes = [5, 5, 5], DBSize = 500)

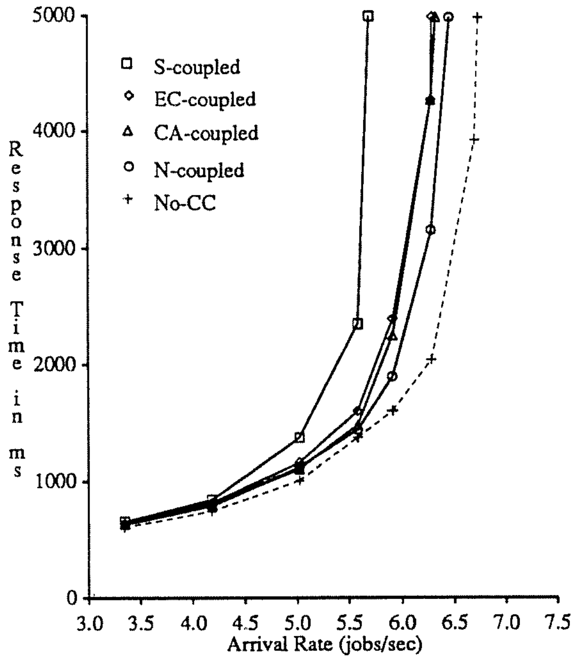

Figure 19: 50% Data Sharing
(TaskSizes = [5, 5, 5], DBSize = 500)

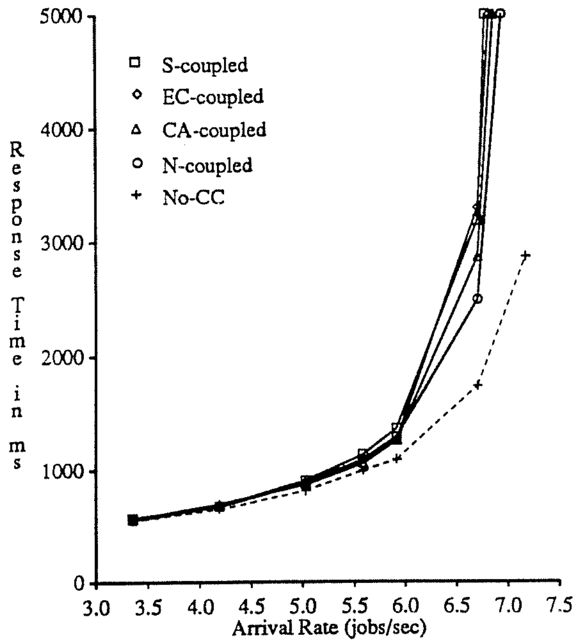

Figure 20: 100% Data Sharing
(TaskSizes = [5, 5, 5], DBSize = 500)

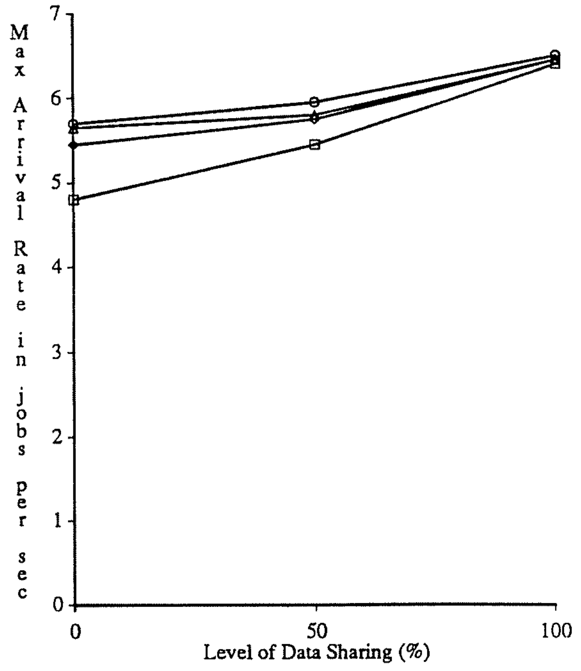

Figure 21: Job Response Time = 2 seconds
(TaskSizes = [5, 5, 5], DBSize = 500)

6. CONCLUSIONS AND FUTURE WORK

The workload of an active DBMS consists of two types of activities: External Tasks, which are the tasks submitted by users or application programs, and Rule Management Tasks, which are caused by the triggering of rules stored in the active database. An External Task (ET) together with its resulting Rule Management Tasks (RMTs) can be thought of as forming a *job*. An interesting question that arises in an active DBMS is: How should ETs and RMTs be combined into transactions? Most proposals for integrating rules with database systems give the user little control, if any, over how tasks are combined into transactions. An alternative is to provide users with the capability to exploit rule semantics by specifying where transaction boundaries should be placed, as in HiPAC [Hsu88, Daya88c]. In this paper, we have studied the performance impact of using such a capability to determine task coupling in an active DBMS. In particular, we have examined the effects of alternative coupling approaches while varying data contention, task complexity, the number of tasks generated recursively by rules, and the amount of data shared by the tasks of a job.

The experiments described here indicate that the performance of an active DBMS can be very sensitive to the coupling semantics of rules. We examined four of the coupling modes possible under the Event-Condition-Action rule model [Daya88c]: S-coupling, where all the tasks of a job are strictly coupled as a single transaction; EC-coupling, where an External Task is coupled with the RMTs (Checkers) that test for rule satisfaction, but not with the RMTs (Actions) that perform the actions for satisfied rules; CA-coupling, where Checkers and Actions are coupled together, but not with the External Tasks; and N-coupling, where each task is a transaction by itself. We found that, under light loads, the type of coupling did not effect job response time appreciably since there were relatively few concurrency control conflicts. However, we did observe that the coupling mode had a significant impact on the mean response time of External Tasks (i.e., the response time from a user's perspective) under such loads. Throughout the study, the response times of External Tasks that were coupled with Rule Management Tasks (i.e., using S-coupling or EC-coupling) were found to be significantly higher than the response times of External Tasks that were executed as stand-alone transactions (i.e., executed using CA-coupling or N-coupling).[4]

[4] Note that user tasks are also decoupled from rule management activity in loosely-coupled knowledge base management systems (where a rule manager polls the DBMS periodically). However, such systems cannot efficiently provide the level of responsiveness possible with an active DBMS [Daya88a].

As data contention was increased in our experiments (by increasing the arrival rate or the size of jobs, or by decreasing the database size), the impact of the various coupling modes on job response time became apparent. Jobs with S-coupled tasks were the first to exhibit a marked increase in response time. In general, N-coupled jobs had the best response times, followed closely by CA-coupled jobs, while EC-coupled jobs performed worse but still better than S-coupled jobs. We also found that the size of RMTs may affect the relative behavior of the coupling modes — for example, when Actions and Checkers were significantly larger than External Tasks, CA-coupling performed worse than EC-coupling. Sharing of data between tasks of a job resulted in improved response times, but the coupling mode choice was found to be important except when the degree of data sharing was extremely high.

It is clear from our results that transactions should be kept as short as possible. This implies that the weakest correct coupling mode should be chosen for each rule in the active database. S-coupling, which combines all of the tasks of a job into a single transaction, always resulted in the worst performance and should be avoided whenever possible. (Note that the S-coupling results are also indicative of what the response times would be for transactions that have been explicitly augmented with code to check conditions and perform actions, as is often done in systems that do not provide rule support.) The experiments involving changes in RMT complexity and in the number of tasks per job also indicate that rules should be designed with care, as both the distribution of work between Checkers and Actions and the likelihood of triggering recursive RMTs affected performance significantly. In addition, they indicate that the passive database should be designed to support efficient access by the RMTs.

A last point brought out by our experiments is that the response of the system degrades rapidly when it is exposed to very high loads. The internal activity generated within the system in response to external interactions, combined with restarts caused by increased data contention, can bring the system into saturation very rapidly as the arrival rate is increased beyond a certain point. Load control techniques to prevent system breakdown under heavy loads (e.g., through the invocation of contingency plans) clearly need to be investigated.

In the study reported here, we have dealt exclusively with transaction management issues in active databases. Our model of an active DBMS provides us with a framework for studying other aspects of active database management as well. The next focus of our research will be the problem of scheduling the physical resources (buffers, CPUs, and disks) of an active DBMS in the presence of time constraints such as deadlines. We intend to study this problem in two phases. Initially, we will restrict our attention to the problem of meeting time constraints in passive databases. We will then extend our work in the area of scheduling to active databases; there we will examine the possibility of invoking contingency plans when time constraints cannot be met by the tasks as scheduled originally. The ultimate goal of this research effort is to design a set of scheduling policies for an active, time-constrained DBMS that encompass both transaction scheduling and resource scheduling.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions: a Performance Evaluation," *Proc. of the 14th Int'l Conf. on Very Large Data Bases*, Los Angeles, CA, Aug. 1988.

[Blak86] Blakeley, J. A., Larson, P-A., and Tompa, F. W., "Efficiently Updating Materialized Views," *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Washington, D.C., 1986.

[Bune79] Buneman, P., and Clemens, E., "Efficiently Monitoring Relational Databases," *ACM Transactions on Database Systems*, Sept. 1979, pp. 368-382.

[Care88] Carey, M., and Livny, W., "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. of the 14th Int'l Conf. on Very Large Data Bases*, Los Angeles, CA, Aug. 1988.

[Daya88a] Dayal, U., et al, "HiPAC: A Research Project In Active, Time-Constrained Database Management," Technical Report CCA-88-02, Computer Corporation of America, Boston, MA, June 1988.

[Daya88b] Dayal, U., et al, "The HiPAC Project: Combining Active Databases and Timing Constraints," *Special Issue on Real-Time Data Base Systems, SIGMOD Record 17*, No. 1, March 1988.

[Daya88c] Dayal, U., Buchmann, A., and McCarthy, D., "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System," *Proc. 2nd Int'l Workshop on Object-Oriented Database Systems*, Bad Muenster am Stein, Ebernburg, West Germany, Sept. 1988.

[Eswa75] Eswaran, K. P., and Chamberlain, D. D., "Functional Specifications of a Subsystem for Data Base Integrity," *Proc. of the 1st Int'l. Conf. on Very Large Data Bases*, Framingham, MA, Sept. 1975.

[Eswa76] Eswaran, K. P., "Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System," IBM Research Report RJ1820 (Aug. 1976).

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Hans87] Hanson, E., "A Performance Analysis of View Materialization Strategies," *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, San Fransisco, CA, 1987.

[Hsu88] Hsu, M., Ladin, R., and McCarthy, D., "An Execution Model for Active Data Base Management Systems," *Proc. 3rd Int'l Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[Huds86] Hudson, S., and King, R., "CACTIS: A Database System for Specifying Functionally-Defined Data," *Proc. 1st Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986, pp. 26-37.

[Koen81] Koenig, S., and Paige, R., "A Transformational Framework for the Automatic Control of Derived Data," *Proc. of the 7th Int'l Conf. on Very Large Data Bases*, Sept. 1981, pp 306-318.

[Kotz88] Kotz, A. M., Dittrich, K. R., and Mulle, J. A., "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism," *Proc. Int'l Conference on Extending Database Technology*, Venice, Italy, March 1988.

[Lind86] Lindsay, B., Haas, L., Mohan, C., "A Snapshot Differential Refresh Algorithm," *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Washington, D.C., 1986.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI, 1988.

[Morg83] Morgenstern, M., "Active Databases as a Paradigm for Enhanced Computing Environments," *Proc. of the 9th Int'l Conf. on Very Large Data Bases*, Florence, Italy, 1983, pp. 34-42.

[Rasc88] Raschid, L., Su, S. Y. W., "A Transaction Oriented Mechanism to Control Processing in a Knowledge Base Management System," *Proc. 2nd Int'l Conf. on Expert Database Systems*, Tyson's Corner, VA, 1988, pp 163-174.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. on Database Systems* 3, 2, June 1978.

[Rous82] Roussopoulos, N., "View Indexing in Relational Databases," *ACM Trans. on Database Systems 7*, No. 2, June 1982, pp. 258-290.

[Sell88] Sellis, T. K., and Roussopoulos, N., "Deep Compilation of Large Rule Bases," *Proc. 2nd Int'l Conf. on Expert Database Systems*, Tyson's Corner, VA, 1988.

[Stan88] Stankovic, J., Zhao, W., "On Real-Time Transactions," *Special Issue on Real-Time Data Base Systems, SIGMOD Record 17*, No. 1, March 1988.

[Ston86a] Stonebraker, M., "Triggers and Inference In Database Systems," *On Knowledge Base Management Systems, Brodie and Mylopoulos (Eds.), Springer-Verlag, 1986.*

[Ston86b] Stonebraker, M., and Rowe, L.(editors), "The POSTGRES Papers," Memorandum No. UCB/ERL M86/85, Electronics Research Laboratory, University of California, Berkeley, CA, 5 November 1986.

[Ston86c] Stonebraker, M., and Rowe, L., "The Design of POSTGRES," *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Washington, D.C., 1986.

[Ston88] Stonebraker, M. (editor), "Readings in Database Systems," Morgan Kaufman Inc., 1988.

[Tay85] Tay, Y., Goodman, J., and Suri, R., "Locking Performance in Centralized Databases," *ACM Transactions on Database Systems*, 10, 4, December 1985, pp. 415-462.

[Tzvi88] Tzvieli, A., "On The Coupling of a Production System Shell and a DBMS," *Proc. 3rd Int'l Conf. on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

APPENDIX: A Coupling Mode Example

The purpose of this Appendix is to illustrate how one can exploit rule semantics to determine where transaction boundaries should be placed in an active DBMS. In particular, we use an example based on an inventory control rule to show how changes in the semantics of the rule affects the way in which ETs and RMTs can be coupled into transactions. The coupling alternatives discussed here are a subset of the transaction coupling modes that can be specified for rules in the HiPAC system [Daya88c].

Parts(pno, pname, size, weight, importance)
Stock(pno, quantity, ordered)
Prices(pno, supplier, cost)

Figure A1: Information About Parts in Stock.

In this example, we will assume that we have an active DBMS controlling the operation of a completely automated warehouse, with parts being requested at terminals and obtained from the warehouse via a computer-controlled delivery system. Figure A1 summarizes the part-related information that is kept in the passive portion of the database for this application. The DBMS stores a description of each part, the quantity of each part both in stock and on order, and the current price for each part from each supplier that carries that part. Checkouts from the warehouse are monitored by the inventory control rule presented in Figure A2.[5] When a checkout of a critical part reduces the sum of the amount in stock and the amount on order to a level below 100, the rule places an order for additional 1000 parts from the supplier who is offering the part at the lowest price at the moment. The ordering action itself is performed by a procedure called *PlaceOrder*, which generates a requisition and also updates the number of part instances on order.

Figure A2 says nothing about how transaction boundaries should be imposed for the inventory control rule. One possible coupling mode in the HiPAC system is **N-coupling**. With N-coupling, a user transaction that updates the quantity on hand for a part will cause HiPAC to schedule the condition-check for the rule to be performed as a separate transaction. Then, if this condition-checking transaction detects that the part is both critical and running low, the part-ordering action will be scheduled as a third, separate transaction. These scheduling decisions are reliable in the sense that the system guarantees that the condition will be checked eventually (despite system failures) and that the action will eventually be performed if the condition is indeed found to be satisfied. However, all rule-related processing takes place outside the user transaction (and only if it completes successfully). N-coupling's advantage is therefore that it minimizes

[5] The language used to express the rule in our example is not the actual HiPAC rule language; it is simply an informal, relational calculus-like language intended to convey the flavor of ECA rules. Identifiers preceded with $ are variables used to capture and pass values from one part of a rule to another.

both the impact of the rule on the user transactions and the length of the rule-related transactions; however, it can potentially permit the placement of multiple orders (if several orders for the same critical, low-quantity part occur within a sufficiently short time period).

A rare possibility of multiple orders may be tolerable in some applications, but other applications may deem it intolerable. In the latter case, HiPAC's **CA-coupling** mode can be used to prevent double-ordering. With this coupling mode, the condition-check and the action (if needed) take place as a single, separate transaction; again, this transaction is scheduled (reliably) by a user transaction if it updates the quantity on hand of a part. As in the fully decoupled (i.e., N-coupled) case, CA-coupling adds no rule-related processing to the user transaction. And, since the condition-check and the action occur together as a single transaction in this case, it is guaranteed that double orders will not be placed. However, it is possible for potentially long delays to occur between the update to the quantity on hand and the subsequent placement of an order, as might happen if the system is heavily loaded.

If the needs of the application dictate that the placement of an order is a prerequisite for the successful delivery of a critical part which is running low, then the entire execution of the rule must take place as part of the user transaction. In this case, HiPAC's strictest coupling mode, **S-Coupling**, must be specified for the rule. With this coupling mode, HiPAC will ensure that the delivery of the part will occur only if the ordering action (if required) is performed successfully.

```
on:
  update quantity of Stock($pno, $quantity, $ordered)

if:
  (S in Stock where S.pno = $pno and S.quantity + S.ordered < 100)
  and (P in Parts where P.pno = $pno and P.importance = "critical")

then:
  PlaceOrder($pno, $supplier, 1000)
  using $supplier =
    (P1.supplier from P1 in Prices where P1.pno = $pno and
        P1.cost = (min(P2.cost) from P2 in Prices where P2.pno = $pno))
```

Figure A2: Inventory Control Rule for Critical Parts.