A Solution to the Parallel Trigger Termination Problem

Nabeel Al-Fayoumi, Eric N. Hanson

Rm. 301 CSE, P.O. Box 116120
CISE Department
University of Florida
Gainesville, FL 32611-6120
nabeel@cis.ufl.edu
hanson@cis.ufl.edu
http://www.cis.ufl.edu/~hanson/

February 1996

TR-96-009

Abstract

In the design of a parallel active database system for a shared-nothing hardware platform, a problem that arises is how to quickly determine when triggers have terminated. This problem is important because triggers normally run as part of the update transaction that triggered them. This update transaction cannot commit until its triggers have finished executing. Hence, for fast update response time, trigger termination must be detected as soon as possible. In addition, in a parallel active database system for a shared-nothing system, it is important to allow trigger actions to execute concurrently on multiple processors to get intra-transaction parallelism. The parallel trigger termination problem is to determine correctly and as quickly as possible when all trigger activations running on behalf of an update transaction have terminated. This paper presents a solution to this problem based on use of a state transition graph for each concurrently executing rule agenda, and proves the solution correct.

1 Introduction

In order to get very high performance in an active database system it is possible to both test trigger conditions and run trigger actions in parallel. We are currently adding a parallel trigger system to the Paradise DBMS [6] in a project called Phalanx. Paradise is a parallel object-relational DBMS based on a shared-nothing hardware model. The parallel processing features of Paradise are comparable to those of the Gamma system [5]. Tables in Paradise are partitioned horizontally across the processors.

The Phalanx parallel trigger system supports a kind of event-condition-action (ECA) rules [12]. The general form of a trigger in Phalanx is as follows:

```
create trigger <triggerName>
[propertyList]
from <fromList>
[on <event-specification>]
[when <qualification>]
for each [row | statement]
begin
  command-list
end
```

The propertyList can be a list of zero or more clauses describing properties of the trigger. One type of property clause can have the following form (others are omitted as not relevant here):

```
sequence_group = Identifier
```

The meaning of this property will be described shortly.

An example of a Phalanx trigger is as follows:

```
create trigger trackSalaryDept17
sequence_group = group1
from emp
on update to emp.salary
when emp.job = "programmer"
for each row
begin
  insert into salaryHistory(emp.eno, Date(), OLD emp.salary)
end
```

The meaning of the above rule is that when the salary of a programmer is updated, their old salary will be logged in the salaryHistory table along with a date stamp. In the remainder of the paper, we use the terms "rule" and "trigger" interchangeably.

In the Phalanx rule language, a rule action can contain a list of arbitrary database commands, including updates, and a special "raise event" command for sending notifications to application programs [11]. Since trigger actions can issue arbitrary update commands, the action of a trigger can cause other triggers to fire on any processor. In addition, in a parallel active database system for shared-nothing hardware, it is important to allow trigger actions to execute concurrently to get intra-transaction parallelism. To achieve intra-transaction rule execution parallelism, we have introduced the notion of *trigger sequence groups* in Phalanx. The total set of triggers is partitioned into non-overlapping subsets called trigger sequence groups. The sequence_group clause in the **create trigger** command is used to identify the sequence group a trigger belongs to. By default, if a trigger is not explicitly assigned to a sequence group, it is in a sequence group all by itself.

A *rule agenda* is a priority queue of rules from one execution group which are waiting to run. During execution of rules for a transaction, that transaction has one logical rule agenda for each trigger sequence group, except for sequence groups of size one, as noted below. If a trigger sequence group has more than one trigger in it, then the agenda for that sequence group is always on a single processor, so that trigger actions for that group can be executed serially. If a sequence group has only one trigger in it, the trigger can fire from any processor where data satisfies its condition, and hence there may be more than one agenda active for that trigger at one time.

After a transaction does its updates, but before rules have been run for that transaction, there may be several agendas containing rules triggered by the transaction which are ready to run. Prior to transaction commit, these agendas are told to start executing rules in parallel. Agendas may be on different processors, and thus may execute rule actions concurrently. After rules terminate, the transaction commits, making the transaction and all rule executions that depend on the transaction atomic.

Trigger sequence groups are provided so the programmer can explicitly divide the set of triggers into several groups which can safely execute in parallel. Within a trigger sequence group, rules are run serially. This allows the rule programmer to serialize rules when needed for correctness, and also exploit parallelism when possible. The notion of a trigger sequence group is comparable to the *production set* concept proposed for the PPL parallel production language [1, 2].

With a trigger execution model that allows multiple trigger actions to fire in parallel, but still treats a transaction and all triggers that depend on it as an atomic unit, an important task is to quickly determine when all rules triggered as a result of a transaction have terminated. We call the problem of finding out when parallel trigger execution has terminated the *parallel trigger termination problem*. Finding out when rules are in a quiescent state as soon as possible is crucial because the transaction cannot commit until rules have terminated. In what follows, we elaborate on the trigger termination problem and present a solution and correctness proof.

2 Background

We will discuss in some detail the architectural platform on which our parallel database and trigger system will be running. This is necessary to comprehend the different elements that

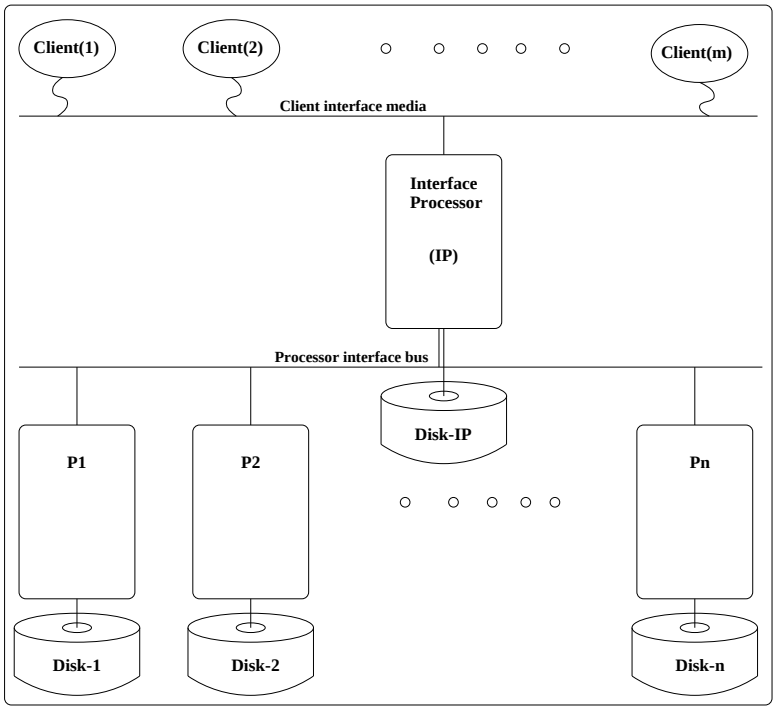

Figure 1: Shared-nothing system architecture

play a roll in the parallel trigger termination problem. The shared-nothing multiprocessor system we assume is similar to that used in the Gamma project [5], where each processor in the system has its own disk storage unit, main memory, and disk controllers. An interconnection network provides a communication media between all the processors. In our case, we will use a high speed switch to reduce contention, and support the transfer of large data chunks efficiently. Figure 1 shows the basic hardware model we'll be using.

One processor will have to take charge of interfacing the client processes to the system processors, which we call the *interface processor* (IP). It can be either preassigned or elected to control system processors. The IP will carry out the task of receiving users queries, planning and distributing processors tasks, synchronization between processors, and other related tasks. One nontrivial task the IP is involved with, which in fact is our main concern in this paper, is starting trigger action execution for a transaction by commanding all involved processors to begin executing rules on their local agendas, and then determining when all the processors have finished executing all the rules. When rules for a transaction have terminated, we say that an *idle steady state* has been reached.

In the following sections, we discuss more background related to parallel trigger termination, paving the way to a discussion of a solution to the parallel trigger termination problem.

2.1 Trigger termination

Here, we discuss trigger termination issues in conventional, single-processor trigger systems, single-processor production-rule systems used for AI programming, parallel AI production

rule systems, and parallel trigger systems.

Single-processor trigger systems such as the Starburst rule system, Ariel, HiPAC, the POSTGRES rule system, Ode, Chimera, DATEX etc. [10, 18, 15, 7, 17, 3] normally either execute rules as procedure or method calls at the lowest level of update command execution (e.g., as in POSTGRES and Ode) or sequentially at the end of the triggering transaction (e.g., as in Ariel). Hence, determining when rules are finished with execution is trivial.

In the case of AI production rule systems such as OPS5 [4], single-processor implementations have a single agenda. When no rules are left on the agenda, rules have terminated. Parallel versions of OPS5 [8], as well as related parallel rule languages like that of the PARULEL system [14] and PPL [1, 2] normally are not concerned with testing for termination of rules. These parallel rule systems can execute rules in parallel, and PPL does allow multiple concurrent rule agendas. However, normally a human is in the loop when these systems are executing. When they stop producing output, it is clear that they are done. It is not crucial to detect rule termination at the earliest possible moment. This is in contrast with the parallel trigger termination problem addressed in this paper, where detecting termination quickly is important to allow fast transaction response time. In parallel production rule systems running on shared-memory multiprocessors (e.g., [9]) termination can be detected when all threads executing to perform production rule condition testing and action execution have halted.

2.2 Parallel trigger termination issues

The above discussion illustrates that testing for trigger or production rule termination is fairly straightforward in single-processor trigger systems and parallel shared-memory trigger systems, and that it is not essential from a practical point of view for shared-nothing parallel production rule systems.

In our parallel trigger environment, testing for fast termination is important, yet the shared-nothing environment presents significant difficulties with respect to termination detection. A major complexity factor is that the processors act and communicate asynchronously, where several events take place at unpredictable time points. This makes it hard to monitor all concurrent activities, and decide their starting and ending points. Since our hardware model is based on the concept of a shared-nothing multiprocessor system, and we want to keep all processors busy to achieve maximum parallel speedup, it is intuitively appealing to use asynchronous communication. If we can efficiently utilize such a scheme and avoid its disadvantages, we can elevate the independence level, avoid blocking, and achieve much faster response time. We mention all these complications and hurdles, because we believe it is useful to give some feeling of the complex environment we are dealing with, so that the reader can appreciate the magnitude of the problem, and then see the beauty in a simple solution for a complex problem.

3 The idle steady state and its recognition

As mentioned before, each active processor in the system can accommodate zero or more agendas locally, and each agenda can have zero or more trigger activations enqueued. Single

trigger agendas are for one trigger, and correspond to unary trigger sets which may be replicated among all active processors. Multiple trigger agendas hold more than one trigger, and correspond to sequence groups of size two or more. These can only be located on a single processor to guarantee the serialization of the rule actions located on one agenda, which must run sequentially according to our parallel trigger language design. The action of a trigger fired on one processor may produce data that needs to be stored on another processor. This is a side effect of partitioning the data among system disks. The same data that was produced by a trigger action on some processor other than the data's owner, can now fire some rules when received and processed by the data's home processor. This in turn can cause a trigger matching that data to go on an agenda *on any processor*. In general trigger execution can cause activation of arbitrary rules on arbitrary agendas (and thus arbitrary processors) depending on the rules, the data, and the updates issued.

Now, we can take a look at the system from the air and see what goes on, beginning at the user end. Suppose a user runs an application program that submits a transaction. The client application process submits the transaction to the IP, which plans the query or queries in the transaction, and then distributes tasks to the processors according to the plan and data partitioning. The IP then broadcasts a "go" command to all involved processors to start executing their part of the plan. Each processor will run its part of the query or update. In our implementation, updates to individual tuples or objects cause "tokens" to be generated. These tokens are update descriptors, describing the old and new value of the updated object, and the type of operation that caused the update. Tokens are mapped to the processor where the new tuple value is supposed to reside, according to the built-in partitioning criteria of Paradise. Trigger condition testing is initiated on the processor to which the token is mapped (full details of the trigger condition testing mechanism are beyond the scope of this paper). If a rule condition match occurs, the corresponding trigger will be activated, and inserted in the proper agenda so its action can be executed.

If a produced token belongs to a nonlocal partition, then before the token is transmitted to its destination processor, it is passed to a local process called the *slave activity monitor* (SAM). After passing a token to the SAM, the process that generated the token will pursue other activities if needed.

The SAM process operates concurrently with other local processes. Its main task is to count the outgoing tokens, and corresponding acknowledgments from receivers, which confirm the end of any nonlocal effects caused by those tokens. This process maintains a single counter that we call the *slave activities counter* (SAC), and a *slave activity table* (SAT), where each transmitted token's ID is recorded along with the destination processor ID and any other attributes needed to keep track of associated nonlocal activities. The destination processor is also referred to as the slave processor. Whenever a token is sent out, it is recorded along with the necessary information in the SAT, then the SAC is incremented, and finally the token is transmitted to its destination.

After the home processor of a sent token receives it, the processor will treat it as a local token and process it accordingly, but it has to remember such tokens and monitor their effects, so that it can acknowledge the senders when all such activities have come to rest. When an acknowledgment is returned to the sender's SAM process, it will first decrement the SAC by one, then remove the corresponding token entry from the SAT. This way, any

processor initiating non-local activity, which we refer to as the *master processor* for that activity, can check (1) whether it has any nonlocal activities, and (2) how many non-local activities are being handled by other processors which we refer to as *slave processors* since they are doing work on behalf of the master.

In general, each processor in the system can know when it has finished both its local tasks, and all its nonlocal activities that are carried out by its slaves. At this point, that master can claim that it has finished its assigned work, and should guarantee that no activities belonging to it are active on other processors.

Figure 2 shows a relatively simple scenario, in which three processors are involved. This example involves the following steps:

1. P1 sends token t1 to P3 for processing (assume it's an insert of a tuple). The SAM process in P1 will increment its counter, and add an entry for t1 in the slave activity table.

2. P3 receives t1. It will insert it, but this insert causes a rule to fire, and token t2 is sent to P2 for processing. The SAM process at P3 will follow the protocol for sending a token.

3. P2 processes t2, and when it's completely done, it sends P3 (its current master) an acknowledgment for t2.

4. P3 receives t2's acknowledgments, it detects the end of all activities caused by t1, then it sends a t1 acknowledgment to P1 (its current master). When P1 receives the t1 acknowledgment, the SAM process will clear t1's table entry, and decrement its counter.

If P1 has already finished its local activities, it will check the SAM's counter. If it has reached zero, then P1 will go to idle state and report the transition to the coordinator. Otherwise, it will wait for the rest of acknowledgments. If P1 is busy doing local work, it won't be affected by the activity of its SAM process.

Specifically, a processor involved in parallel rule execution can be in one and only one of three states, *Idle* state, *Master-Active* state, or *Slave-Active* state. After the IP distributes tasks for a command, when a processor receives the "go" command from the IP, the processor starts by entering its Master-Active state. It will stay in that state until all rule processing activity that it initiated has completed. This occurs when all agendas become empty, and more important, when the SAM's counter (SAC) contains a zero, which implies no slave activities are taking place. At this point, the processor will go into its idle state. When it makes the transition from Master-Active to Idle state, it has to report the end of its activity to a coordinator process (referred to as the *All-Idle Monitor process, or AIM*) which resides at the IP. Any time a processor receives tokens to process after it has moved from the Master-Active state to the idle state (and reported that fact to the coordinator), then it will go to the Slave-Active state, and will be executing on behalf of the processor that sent the tokens.

The complete state transition diagram is shown in Figure 3. The main advantage in this state transition scheme is that a processor in the Master-Active state can never go back to

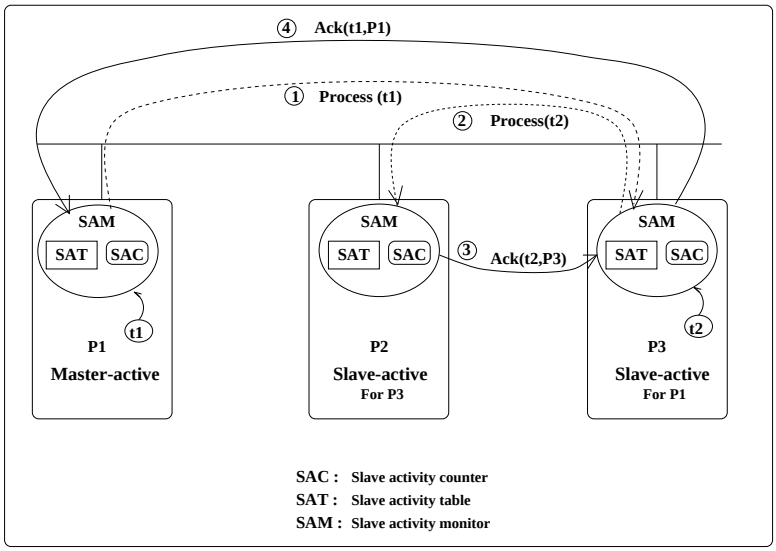

Figure 2: Master-Slave processors interaction

that state once it has moved to the Idle state. If it becomes active again, then it will go to the Slave-Active state, because it will be working as an extension of another processor's activity. This state transition logic is the main component of the solution to the trigger termination problem. An important feature of the approach is that each processor bears the responsibility of monitoring its activities even if they are taking place on other processors. This simplifies the termination logic.

To collect the processors' idle signals, the all-idle monitor process (AIM) was introduced. This process will maintain an *active processor counter*. When the coordinator assigns processors to do some work, the number of these processors is recorded in the AIM's counter. When the AIM receives an idle report from any processor, it will decrement the counter, so that at any time, the counter will hold the number of those processors in the master-active state. A zero value in the AIM's counter indicates that all processors have returned to the idle state, and there is no trigger processing activity of any kind taking place on any processor. Based on that, the coordinator can confidently claim an *idle steady state*. This result can be stated as the following theorem.

Theorem When the last processor in Master-Active state moves to its Idle state, then it and all other processors must be idle.

Proof: Recall that during a single transaction, if a processor enters the Idle state after being in the Master-Active state, it can never go back to Master-Active state again. If the processor ever becomes active again, it will be in its Slave-Active state, that is, it will be working on behalf of some other processor, which can be in any active state. A processor will never acknowledge any tokens received from another processor until it is certain that it has finished all related activities completely. Hence, an active processor will never be able to go idle until its SAM process receives all acknowledgments for every activity carried out on the processor's behalf by any other processor (i.e. SAM's counter contains zero).

Based on the above discussion, we will prove the theorem by contradiction. Suppose that the coordinator has received idle signals from all processors in the system but the last

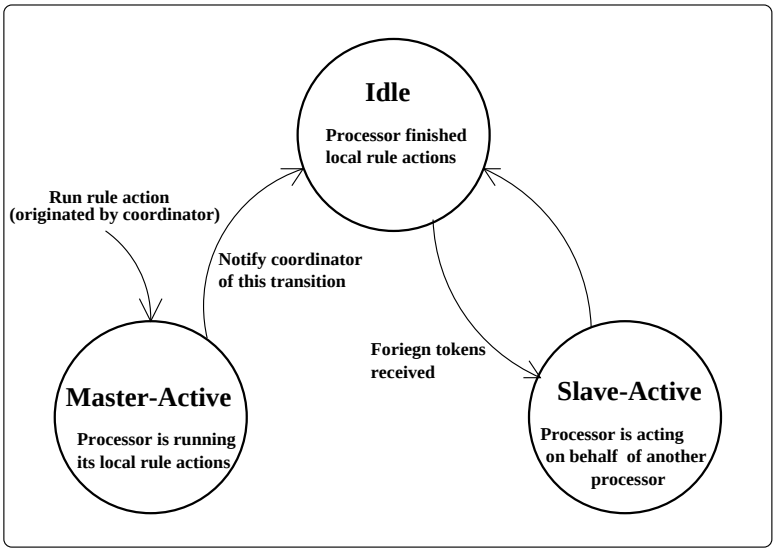

Figure 3: Processor state diagram

one, which is still in the Master-Active state. At this point, this last processor finishes its work, and sends the coordinator its idle signal. Assume that after this final idle signal is sent, there are still some other processors in an active state. Since all other processors have reported an Idle state, and no one is allowed to go back to Master-Active state, then they are in the Slave-Active state. If that is the case, then all active processors must be slaves of the last master processor, because they already finished their own work. But we know that the master processor has already sent its idle signal, which implies that all of its slaves have sent it their acknowledgments. We can conclude that all processors finished their own work plus the work done on behalf of the master, so they all must be in Idle state along with the master, which contradicts our assumption that some processors were in an active state. *Hence, when the last Master-Active processor goes to Idle state, all other processors in the system must be idle, and the coordinator can recognize an idle steady state. Based on that, the coordinator can safely commit the current transaction.*

4 Related work

In addition to the previously mentioned work on non-parallel active database systems, non-parallel production-rule systems, and parallel production rule systems, a number of commercial products that support simple forms of triggers in parallel server configurations are available. These include that parallel Sybase [16] and Oracle [13] servers. The the internal architecture of the trigger subsystems in these products has not been published. We conjecture that they run triggers in a synchronous fashion at one or more points within update transactions. It appears they are capable of parallel trigger condition testing and action execution to a degree, but that trigger actions are executed using blocking local or remote procedure calls. Hence, the processes or threads that start trigger execution know that the triggers they initiated have terminated when the associated procedure calls return.

Using blocking, procedure-call-style control flow clearly will limit potential intra-transaction trigger processing speedups by limiting available concurrency. Fast intra-transaction parallel trigger processing is one of the main goals of the Phalanx project, and has motivated us to investigate asynchronous trigger condition testing and action execution. This lead to the identification and solution of the parallel trigger termination problem presented in this paper.

5 Conclusions

This paper has outlined a parallel, asynchronous trigger execution strategy based on use of multiple rule agendas which can execute rule actions concurrently. Within this framework, the parallel trigger termination problem was identified. Fast detection of parallel trigger termination is crucial in parallel database trigger systems since triggers run as part of the triggering transaction, any delay in determining when triggers have terminated will lengthen transaction response time.

A solution to the parallel trigger termination problem based on use of a state transition diagram at each processor in a shared nothing parallel machine was presented, and proven correct. The solution relies on careful tracking of acknowledgments of requests to other processors to perform work, and the fact that once a processor enters its Idle state, it can never enter its Master-Active state again.

The parallel trigger termination problem presented arose in the design of a trigger system extension to the Paradise parallel DBMS, as part of the Phalanx project. Currently, our extended Paradise system supports triggers when running on a single processor. The high-level design of the parallel trigger extensions for Paradise has been done, including the design of the trigger termination algorithm outlined here.

Though the asynchronous parallel trigger execution environment's behavior can be quite complex, the actual solution presented to the parallel termination problem is relatively simple. This simplicity adds to the appeal of the solution, making it feasible to implement to effectively detect rule termination in real parallel trigger systems.

References

[1] Anurag Acharya. PPL: an explicitly parallel production language for large scale parallelism. In *Proceedings of the IJCAI Workshop on Production Systems and Their Innovative Applications*, August 1993.

[2] Anurag Acharya. Scalability in production system programs. Technical Report CMU-CS-94-211, Carnegie Mellon University, November 1994. PhD thesis.

[3] David A. Brant and Daniel P. Miranker. Index support for rule activation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 42–48, May 1993.

[4] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming*. Addison Wesley, 1985.

[5] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[6] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-server Paradise. In *Proceedings of the 20th VLDB Conference*, 1994.

[7] N. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, September 1991.

[8] Anoop Gupta. Implementing OPS5 production systems on DADO. In *IEEE International Conference on Parallel Processing*, pages 83–91, August 1984.

[9] Anoop Gupta, Charles Forgy, D. Kalp, A. Newell, and M. Tambe. Results of parallel implementation of OPS5 on the Encore multiprocessor. In *International Conference on Parallel Processing*, 1987.

[10] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[11] Eric N. Hanson, I-Cheng Chen, Roxana Dastur, Kurt Engel, Vijay Ramaswamy, and Chun Xu. Flexible and recoverable interaction between applications and active databases. Technical Report 94-033, University of Florida CIS Department, September 1994. `http://www.cis.ufl.edu/cis/tech-reports/`.

[12] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–224, June 1989.

[13] Running Oracle in a parallel environment, 1995. http://www.oracle.com/ products/oracle7/server/whitepapers/parallel/html/pararchit.html.

[14] S. J. Stolfo, O. Wolfson, P. K. Chan, H. M. Dewan, , L. Woodbury, J. S. Glazier, and D. A. Ohsie. PARULEL: parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4), December 1991.

[15] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):125–142, March 1990.

[16] Sybase SQL server 11 white paper, 1995. http://www.sybase.com/Offerings/ System11/sqlserver11_whitepaper.html.

[17] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.

[18] Jennifer Widom, Roberta J. Cochrane, and Bruce G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 1991.