Institut für Informatik der Universität Zürich

A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST

Andreas Geppert
Stella Gatziu
Klaus R. Dittrich

Nr. 95.18 May 95

A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST

Andreas Geppert, Stella Gatziu, Klaus R. Dittrich

Institut für Informatik, Universität Zürich[1]

Abstract

A benchmark for active database management systems is described. We are particularly interested in performance tests that help to identify performant and inefficient components. Active functionality that is relevant with respect to performance is identified, and a series of tests is designed that measure the efficiency of the performance-critical components. Results obtained from running the benchmark for a concrete system are presented.

Keywords: active database systems, benchmarks, object-oriented database systems

1 Introduction

Active database management systems (ADBMSs) [e.g., 2, 9] have recently found great interest as a topic of database research, and restricted ADBMS-functionality is already offered by some commercial systems [e.g., 19]. An ADBMS implements "reactive behavior" since it is able to detect situations in the database and beyond and to perform corresponding actions specified by the user. Applications using reactive behavior are freed from performing "polling" in order to detect interesting situations. ADBMSs release such applications from encoding (possibly redundantly) situation detection and reactions.

As for any system, ADBMSs should implement their functionality *efficiently*. In fact, the evolution of ADBMSs is currently in a state where performance arguments play a crucial role, both from an application as well as from a system point of view:

- ADBMS researchers have developed different techniques for tasks of an ADBMS such as composite event detection [e.g., 11, 15], and it is thus interesting to compare these approaches with respect to performance.
- Different architectural approaches have been developed and need to be compared. For instance, the layered architecture for ADBMSs is often claimed to be less efficient than an integrated ADBMS [3].

1. Authors' address: Institut für Informatik, Universität Zürich, Winterthurerstr. 190, CH-8057 Zurich, Switzerland. Fax: +41-1-363 0035, Email: {geppert | gatziu | dittrich}@ifi.unizh.ch

- Authors have claimed that an ADBMS outperforms polling in applications [8], which has still to be proven by appropriate measurements.

Nevertheless, figures describing the performance of ADBMSs are not yet available. Moreover, it is so far still unclear what the relevant *performance measures* of an ADBMS might be, and an approach how to methodically measure the (in)efficiency of ADBMSs has not yet been proposed.

The objective of this paper is to describe a benchmark for (object-oriented) ADBMSs. Such a benchmark would be useful for at least three purposes:

- potential users can run it to compare the performance of multiple ADBMSs,
- ADBMS designers can run it to identify performance weaknesses of their system in comparison to others, and
- it can be used to compare the performance of an ADBMS with that of a passive DBMS, where the active behavior is encoded manually in the applications.

The BEAST[2] benchmark focuses on the second point, i.e., our intention is that of designers that want to assess their system. BEAST is intended for testing the active functionality of DBMSs, since appropriate benchmarks for passive DBMSs have already been developed [e.g., 4, 5, 14]. Furthermore, we concentrate on *object-oriented* ADBMSs, since — although we focus on the active part— the underlying data model has some influence on ADBMS performance. Even passive object-oriented and relational systems are intended for different application domains. Therefore, one type of system would be at a disadvantage when BEAST is used to compare relational and object-oriented ADBMSs.

BEAST considers relevant aspects of an ADBMS that need particularly efficient implementation:

- event detection,
- rule management, especially rule retrieval, and
- rule execution.

For each component, we define a group of tests that serve to measure its performance. We have used these tests to measure the performance of the ADBMS SAMOS [10]. Further ADBMSs will be tested in the near future.

The remainder of this paper is organized as follows. The next section gives a short introduction of ADBMSs, as far as necessary to comprehend the benchmark. Section 3 describes the benchmark, and section 4 presents benchmark results for SAMOS. Section 5 concludes the paper.

2. BEnchmark for Active database SysTems

2 Active Database Management Systems

In order to make the benchmark better understandable, we give a short introduction of ADBMSs. An ADBMS is a DBMS that in addition to its regular features supports the specification and implementation of reactive behavior. Most ADBMSs support event-condition-action rules (ECA-rules) for specifying reactive behavior. An event is an implicitly or explicitly defined point in time and specifies when the rule has to be executed. Events can be *simple* (e.g., modification of data items, message sending, transaction begin and commit, time events, abstract events[3], etc.) or *composite* (e.g., conjunction, sequence, disjunction, negation, repeated occurrence, etc.). Currently, different systems support rather different kinds of events. The condition is either a boolean function or a database query. If the condition evaluates to true (or returns a non-empty result), the action is executed. An action is typically written in the data manipulation language (DML) of the ADBMS, and can include the sending of messages in the case of object-oriented ADBMSs.

The *execution model* of an ADBMS specifies how rules are actually executed. It determines how condition evaluations and action executions are realized in terms of the transaction model. The *coupling mode* of a rule specifies when the condition or action parts of a rule are executed with respect to the transaction that triggered the event. Popular coupling modes are *immediate* (directly after the event occurred), *deferred* (at the end of the triggering transaction, but before commit), or *decoupled* (in a separate, independent transaction). In this paper, we assume that the coupling modes for conditions relate condition evaluation to the triggering event, and that the coupling mode for actions relate action execution to condition evaluation. Finally, the execution model also defines how to process multiple rules that are all triggered by the same event. One possibility is to let the user specify (partial) orders, e.g., by means of *rule priorities*.

3 BEAST: A Benchmark for ADBMSs

In this section, we describe the BEAST benchmark for ADBMSs. Before the benchmark is described in detail, we identify relevant requirements and design decisions. We use the steps proposed in [16] as a reference model of how to proceed in benchmark design.

3. *Abstract events* (or *external events)* are events that are not detected by the ADBMS, but that have to be signalled explicitly by the application or the user.

1. Definition of goals and the tested system. For any performance measurement the goals must be defined beforehand. The goal of BEAST is to measure the execution time of the services offered by an ADBMS. We currently assume a single-user ADBMS.

2. Definition of services. There is only one tested service: active behavior. For this service, BEAST proposes a collection of *tests*, each of which focuses on a specific subtask of active behavior. The selection of goals and services should be fair, i.e., it should not favor specific systems. BEAST tests the features that are common to most current ADBMSs, and does not stress special features that are available only for a few systems. The proposed tests are described in detail in section 3.1.

3. Selection of metrics. The performance measure we have chosen for BEAST is *response time*. Since BEAST has no access to internal interfaces of a tested ADBMS, we cannot precisely measure the performance of active behavior subtasks. Thus, BEAST tests invoke active behavior, which always performs several phases such as rule execution. Response time is then defined as the time interval that starts directly before event occurrences and ends directly after rule execution (i.e., when the control returns to the application)[4].

4. Selection of factors. *Factors* are the parameters that influence performance. Typical factors for passive DBMS are buffer size, database size, etc. In section 3.1.4 we identify additional factors for ADBMS performance measurement (e.g., number of defined rules).

5. Workload definition. Workload definition is a prime task in developing a benchmark. In BEAST, the workload is defined such that basic ADBMS functionality can be tested. Note that BEAST does not propose a typical application, since (1) the possible application domains are rather different and (2) knowledge on how to use ADBMSs for real life applications is still evolving. BEAST does not propose a typical application and test its performance, but determines relevant functionalities and their performance.

3.1 Benchmark Design

BEAST is based on the 007 benchmark [4]. It uses the schema of 007 as well as the corresponding databases (i.e., programs to create and fill databases). One reason for reusing parts of 007 is to easily obtain a schema and database. Moreover, for a given object-oriented ADBMS, BEAST and 007 together allow to measure the performance of the active and the passive parts of a system, respectively.

BEAST considers three components where performance is crucial:

- event detection,

4. Exceptions to this definition are necessary for coupling modes other than immediate (see below).

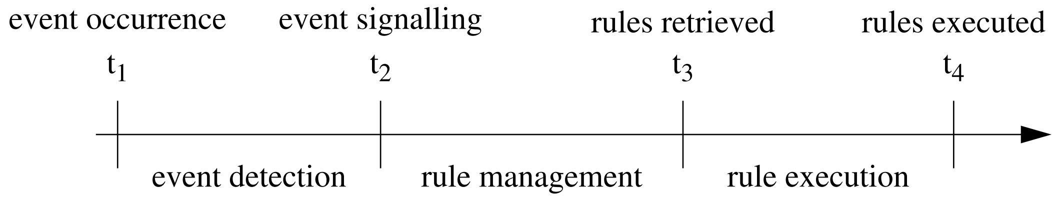

Figure 1. Phases of Active Behavior

- rule management, and
- rule execution.

We have selected these components since they implement the three phases that comprise active behavior (see Figure 1). They are thus contained in most ADBMS-architectures [3, 6, 13, 17].

After an event occurs, it must be *detected*, i.e., ADBMS components must realize (or be notified) that the event has happened. At the end of the event detection phase, the event is *signalled*[5]. The second phase (rule management) starts as soon as the event has been signalled and determines whether (and which) rules must be executed. Internal information must be consulted that links event descriptions with rule definitions. In the simplest case (the `immediate` coupling mode), rule management is directly followed by the rule execution phase (starting at t_3). In this phase, the triggered rules are executed. Each of the three phases is relevant with respect to performance.

Event detection is realized by the components that recognize the occurrence of specific events of interest. Two subtasks of event detection affect performance: detection of primitive and of composite events. In general, we expect that performance is better when the set of signalled primitive events can be small (i.e., only events that have rules attached or that contribute in composite events are actually signalled). Second, composite event detection can be realized in several ways that may have different performance characteristics.

Rule management also affects the performance of an ADBMS. Rule management refers to the storage and retrieval of rules and to the modification of the rulebase. After a primitive event has been signalled, the ADBMS determines whether it is used in ECA-rules and/or whether it participates in a composite event. Since information on rules must be retrieved after the signalling of a triggering event, efficient identification and retrieval of corresponding rules is crucial for ADBMS performance.

5. In general the precise point in time when an event occurred is not known. In BEAST tests, however, we enforce event occurrence and thus know this point in time.

Rule execution refers to the identification of condition and action parts that have to be executed after event occurrences as well as the execution of these parts. In particular, it is interesting how efficiently the various coupling modes are implemented and how efficiently multiple rules triggered by the same event can be executed.

BEAST tests each component with a series of tests. The result of running BEAST is therefore a collection of figures instead of a single figure for each ADBMS. Note that we cannot test the performance of each component directly, due to lacking access to internal interfaces of an ADBMS. Therefore, most BEAST tests specify one or more rules that are triggered when executing the test, i.e., the test actually causes the event occurrence. In order to stress performance of single phases, we keep all other phases as small as possible. For instance, a rule testing event detection performance simply defines the condition to be `false`, such that condition evaluation is cheap and the action is not executed.

We elaborate on each group of tests subsequently. Tested functionality is described, and possible interpretations are given. The rule schema can be found in Appendix A. Note that the tests are not always enumerated consecutively, since some of the ones described in [12] have been omitted in this paper.

3.1.1 Tests for Event Detection

For event detection tests, BEAST focuses on the time it takes to detect an event. Both, the detection of primitive and composite events are tested.

3.1.1.1 Tests for Primitive Event Detection

BEAST contains five tests for primitive detection:

1. detection of value modification (ED-01),
2. detection of message sending (ED-02),
3. detection of transaction events (ED-03),
4. detection of a *set* of different primitive events (ED-04),

We illustrate the execution of tests with the test ED-02. The first operation of this test is to record the actual time. The next operation causes the event in question to occur. In this case, a message is sent. Note that in this way we know the point in time of event occurrence. The ADBMS subsequently detects the event, determines attached rules, and executes them. It then returns control to the test program. Finally, the test program again records the time and computes the required CPU time.

The first three tests measure detection of single events. The corresponding rules for all tests have a false condition and an empty action in order to restrict the measured time to event detection, as far as possible. Coupling modes for action and condition parts are `im-`

`mediate`. Another possible kind of primitive event would be *time events*, but technically no meaningful way for measuring their detection exists.

The test ED-04 runs a transaction that raises multiple abstract events and measures the time needed for this transaction to executed. Directly afterwards, the same transaction is executed again. Hence, information about each raised event is required and thus must be retrieved twice. Buffering of event information can decrease the time needed for the second event occurrence and rule retrieval. Thus, a system that applies buffering for event and rule information might outperform others that do not cache this kind of information.

3.1.1.2 Tests for Composite Event Detection

Composite event detection typically starts after a (primitive or other composite) event has been detected. The event detector then checks whether the occurred event participates in a composite event. This is generally possible in two ways. In the first alternative, the ADBMS records each event occurrence, determines whether it can participate in a composite event, and checks whether other participating events have already occurred. The second alternative is to perform detection of composite events in a stepwise manner, e.g., by means of automata [15] or Petri nets [11]. Of course, the different approaches may have different performance characteristics and therefore need to be compared with respect to efficiency. This is accomplished through tests ED-06 through ED-11.

In order to stress the time needed for composite event detection, we use abstract events in the definitions of composite events wherever possible. Using abstract events enables more accurate measurements, since only the time for event signalling is required and primitive event *detection* is not necessary. In order to measure the entire composite event detection process (even for stepwise detection), the tests raise the component events directly one after the other. Thus, the measured time includes all steps of composite event detection.

BEAST contains six tests for the detection of composite events:

1. detection of a sequence of primitive events (ED-06)
2. detection of the non-occurrence of an event within a transaction (negative event, ED-07),
3. detection of the repeated occurrence of a primitive event (ED-08),
4. detection of a sequence of events that are in turn composite (ED-09),
5. detection of a conjunction of method events sent to the same receiver object (ED-10),
6. detection of a conjunction of events raised by the same transaction (ED-11).

We are interested in the time it takes to detect the events, and therefore conditions, actions, and coupling modes are kept as simple as possible. Tests ED-06 through ED-08 measure event detection for common composite event constructors. Test ED-09 considers one specific constructor applied to events that are in turn composite. Finally, the last two tests measure the performance of event detection when the events of interest are further restricted by event parameters.

3.1.2 Tests for Rule Management

The second group of tests considers *rule management*. It is based on the observation that an ADBMS has to store and retrieve the definition and implementation of rules, be it in the database, as external code linked to the code of the ADBMS, or as interpreted code. Apparently, the time it takes to retrieve rules influences ADBMS performance. Rule management tests measure rule retrieval time, but they do not consider *rule definition* and *rule storage*. These services are executed rather seldomly, and therefore their efficient implementation is less important.

The test RM-1 raises an abstract event, evaluates a `false` condition, and therefore does not execute any action. The three parts are kept such simple in order to restrict the measured time to the rule retrieval time as far as possible.

3.1.3 Tests for Rule Execution

The tests for rule execution are subdivided into two groups: one for the execution of single rules, and one for the execution of multiple rules. The first subgroup of tests determines how fast rules can be executed. The execution of a single rule consists of loading the code for conditions and actions and of processing or interpreting these code fragments. Again, different approaches exist for linking and processing condition and action parts, and can be compared by means of the tests in this group.

Different strategies can also be applied for executing multiple rules all triggered by the same event (e.g., concurrent or parallel execution). The performance characteristics of these approaches are tested by the second subgroup.

For the execution of single rules, we consider one rule with different coupling modes. The coupling mode of the condition is always `immediate`. The coupling modes of the actions are `immediate`, `deferred`, and `decoupled`, respectively. The intention of these tests is to measure the overhead needed for storing the fact that the action still needs to be executed at the end of the transaction (`deferred`), as well as the overhead necessary to start a new transaction in the `decoupled` mode. In order to stress these aspects of rule execution, we use an abstract event in order to avoid event detection, and use a simple `true` condition and an empty action. Note that the performance of condition evaluation and action execution is not of interest, because it is determined by the "passive" part of the DBMS.

The second group of tests for rule execution considers multiple rules. The first test (RE-04) uses four rules all triggered by the same event. Conditions and actions are more complex than in the previous tests, in order to better be able to observe effects of optimization of condition evaluation (ED-04) and concurrency (ED-05). All rules have the same conditions, respectively. Hence, an ADBMS that recognizes equality of conditions (e.g., if it is able to optimize sets of conditions) will perform better than a non-optimizing ADBMS.

All rules have the coupling modes `(immediate, immediate)`. A total ordering is defined for the four rules. In addition to the rule execution, this test measures the overhead obtained through enforcing the ordering of the rules.

The second test in this group (RE-05) again considers four rules all triggered by the same event. However, no ordering is given. An ADBMS that is able to process conditions and actions in parallel or at least concurrently will thus perform better in this test. This test uses the same conditions and actions as test RE-04, such that both tests can be used to observe the impact of orderings on performance.

3.1.4 Factors and Modes

A crucial step when designing a benchmark is the proper identification of *factors* [16], i.e., parameters that influence performance measurements. Several parameters of a database can have an impact on the performance of an ADBMS. In addition to the database parameters relevant for benchmarking a passive DBMS (e.g., buffer size, page size, number of instances stored in the database), these include:

- the number of defined events,
- the fraction of composite events, and
- the number of defined rules.

The time to detect events is ideally constant, i.e., independent of the number of defined events. However, especially for composite events it may be the case that a large number of events slows down the event detection process for single events. Furthermore, an ADBMS needs to store and retrieve internal information on event definitions during (or after) event detection. Apparently, a large number of event definitions can increase the time needed to retrieve event information. It is thus interesting to investigate for each tested ADBMS how large response times are when the number of events increases. We therefore include the number of defined events as a factor.

The second factor (fraction of composite events) determines how many of the events are composite ones. We specify this number in terms of the nesting depth of composite events, i.e., how often composite event constructors are applied recursively. A nesting depth of 0 means that there will not be any composite events, and a nesting depth of 1 specifies that always two primitive events will form one composition. Generally speaking, a nesting depth of `n` means that `n+1` primitive events will be used to form `n` composite events.

Furthermore, the total number of rules defined by a concrete database is relevant for performance. Recall that rule information has to be retrieved before rule execution. While a small number of rules can be entirely loaded into main memory without problems when the ADBMS starts execution, this is no longer possible if the rulebase is large. In the latter case, rules must be selectively loaded into main memory from secondary storage upon rule execution. It is therefore an important question how efficient an ADBMS can handle large sets of rules, and how the system behaves when the number of rules grows larger. For example, an ADBMS that stores rules as objects can make use of the clustering and indexing mechanisms already offered by the passive part of the DBMS. Note also that some tests consider buffering of rule and event information.

For the three factors, we choose three possible values for a small, a medium, and a large rulebase (see Table 1). Tests for larger rulebases are easily possible, since the values of all factors can be specified as parameters of the rulebase creation program.

parameter	rulebase size		
	small	medium	large
#events	50	250	500
nesting depth	2	3	4
#rules	50	250	500

Table 1. Parameter Values for Different Rulebase Sizes

Many rules and events will actually not be used by the benchmark, i.e., their execution is not measured. However, they are important in order to increase the load of the ADBMS as well as the data/rulebase size. These “dummies” therefore yield information whether the ADBMS is able to handle large sets of rules with a performance comparable to small numbers of rules.

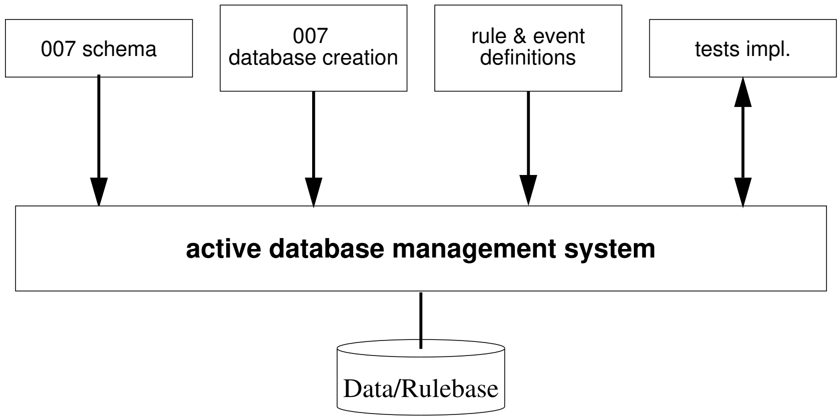

Figure 2. BEAST implementation

3.2 Benchmark Implementation

The implementation of BEAST for an ADBMS consists of the following parts (Figure 2):

- the 007 schema and database creation programs,
- event and rule definition through the rule definition language of the ADBMS,
- specific new classes for the benchmark tests (e.g., response time measuring).

In order to run the benchmark for a concrete ADBMS, the 007 schema and the database creation programs must be adopted for the data model of the ADBMS. The next step consists of specifying and compiling the ECA-rules for the system. In the final step, the desired tests are executed. Each test computes the CPU-time the operating system process has spent for the test execution (due to the fact that the process is subject to operating system scheduling, process-specific CPU-time can be a fraction of the total elapsed time). Each test is executed separately in order to avoid “cross-test” buffering effects.

It is not necessary to execute all tests for each ADBMS. Alternatively, a designer can choose the tests where results are interesting, and can easily configure and instantiate the benchmark for his/her needs.

4 Benchmark Application

We have run the benchmark on our home-grown ADBMS SAMOS [10, 13]. SAMOS offers a rich collection of event definition facilities and uses Petri nets for composite event detection [11]. We therefore are especially interested in the performance of the Petri net approach and how well SAMOS scales for medium and large rulebases.

4.1 Benchmark Results

In this section, we present the results of running the benchmark on SAMOS. We also discuss differences to earlier measurements documented in [12]. Each test has been run multiple times for the same database/rulebase size. Arithmetic means, standard deviations, and confidence intervals have been computed. Table 2 shows means and confidence intervals for a 90% confidence level (i.e., the mean of all possible executions of a test is within the interval with 90% confidence [16]). All results refer to CPU time in milliseconds.

Test	Parameter	Configuration (Rulebase Size)			
		empty	small	medium	large
ED-02	mean	150	280	246	878
	conf. interval	[124, 176]	[264, 296]	[235, 257]	[857, 899]
ED-04	mean	768	984	1472	1830
	conf. interval	[750, 786]	[941, 1027]	[1422, 1522]	[1799, 1861]
	mean	674	878	1080	1056
	conf. interval	[653, 694]	[857, 899]	[1044, 1116]	[1021, 1091]
ED-06	mean	532	758	1210	2596
	conf. interval	[509, 555]	[721, 794]	[1186, 1234]	[1570, 2622]
ED-08	mean	1090	1428	2480	4814
	conf. interval	[1074, 1106]	[1410, 1446]	[2436, 2524]	[4768, 4842]
ED-09	mean	1102	1494	2566	5350
	conf. interval	[1069, 1138]	[1448, 1540]	[2516, 2616]	[5327, 5373]
ED-11	mean	428	538	1142	1826
	conf. interval	[389, 467]	[490, 586]	[1081, 1203]	[1789. 1863]
RM-01	mean	164	176	358	184
	conf. interval	[143, 184]	[135, 217]	[332, 384]	[173, 195]
RE-01	mean	152	272	588	266
	conf. interval	[126, 178]	[238, 306]	[535, 641]	[245, 287]
RE-02	mean	178	252	476	232
	conf. interval	[152, 204]	[217, 287]	[453, 499]	[215, 249]
RE-04	mean	384	550	496	1296
	conf. interval	[355, 412]	[516, 583]	[467, 525]	[1259, 1333]

Table 2. BEAST Results for SAMOS

The tests have been run on a SUN- SparcServer 4/690 server under SUNOS 4.1.3. Each test has been compared with four different configurations that vary in the number of dummy events and rules (cf. Table 1): (1) no dummy events/rules, (2) 50 dummy events, (3) 250 dummy events, and (4) 500 dummy events. Accordingly, we use the small (1 and 2), the medium (3) and large (4) 007 databases.

4.1.1 Results for Event Detection Tests

The test for primitive event detection (ED-02) shows a dependency on the rulebase size (concretely, the total number of defined events). Ideally, we would expect that ED-02 is independent of the rulebase size, or at least that the slope of the increase is much smaller than in the shown results. The reason for the increase is that event objects must be retrieved. In other words, SAMOS currently unfortunately scans the entire extension of event definitions upon event signalling in order to find appropriate event objects, and applies string comparisons for determining the appropriate event object.

Surprisingly, ED-02 shows slightly smaller times for medium rulebases. Similar effects have been observed for RE-04 as well as for RM-01, RE-01, and RE-2 for large rulebases. We have executed much more runs for these tests and have observed that these effects are stable (i.e., no peculiar exceptional measurements). We have currently no explanation for this strange behavior, but suspect that ObjectStore internal storage placements and indexing strategies are responsible for this effect.

ED-04 contains two sequences of event occurrences (ED-04a and ED-04b). It can be observed that the second sequence requires much less time than the first one, i.e., there is a buffering effect of event objects. Moreover, in comparison to ED-02 much more event objects must be retrieved, but measured times are only twice as large (for large rulebases). We therefore conclude that the expensive actions in ED-02 (and elsewhere) are not the retrievals of *single* event objects, but querying the entire *event extension*. Therefore, in ED-04, there is not only a buffering effect between the two event signalling sequences, but also within each of them.

The tests for composite event detection also show a strong dependency on the rulebase size. For these tests, we would also expect nearly constant behavior. Like for ED-02, the increase is due to the use of queries for determining information on (composite) events. Since Petri net structures are much more complex than the data structures for primitive events, more queries are used and results are more dramatic. Since the times required by all other tests are much smaller, it is obvious that the event detection component itself is responsible for these large times.

For most tests the time required for the detection of the component events is also interesting. The reason to measure the detection of component events is twofold:

- since component events typically do not occur directly one after the other, the component detection time tells how much an application is slowed down whenever a participating event occurs.
- for system designers, it is interesting to see where composite event detection spends which fraction of the total time.

It turns out that for most constructors the detection of the second component event requires much more time than the detection of the first. This is only partially due to the fact that the second time also includes rule retrieval and rule execution. After the detection of the second component event, the composite event must be signalled, and (in SAMOS) event parameters must be determined for the composite event. These additional actions obviously are responsible for the larger required time for the second participating event. For more details, see previous results documented in [12].

4.1.2 Results for Rule Management Tests

Rule retrieval time also depends on the rulebase size (RM-01). Again, the increase is due to the inefficient querying of the event extension. Once the event object has been found, corresponding rule objects can be retrieved via pointer traversals, which has only marginal effects on the response time.

4.1.3 Results for Rule Execution Tests

Three tests have been performed for rule execution: test RE-03 and RE-05 have been omitted since decoupled rules and priorities are not yet implemented in SAMOS. The rule execution tests show that rule execution time is also dependent on the rule/database size. Test RE-04 (in comparison to ED-02) shows that rule execution is quite cheap in comparison to event detection. ED-02 performs one condition evaluation and no action execution, while RE-04 evaluates four conditions and executes four actions, but response time of RE-04 is approx. 25% higher than that of ED-02. Hence, retrieval of event information from the extension of event objects is **the** dominating factor in the current implementation of SAMOS.

4.1.4 Discussion of Results

Although we have not yet enough comparative figures for other systems, we feel that management of events, their retrieval, and event detection (particularly of composite events) is not yet acceptable from a performance point of view.

We have drawn two major observations from the results. First, the storage and retrieval of events must be significantly improved. In our future work, a re-design and re-implementation of some components, pointers instead of queries (joins) will be used for connecting related event and rule information. Likewise, the facilities offered by ObjectStore for physical storage (indexing and clustering) will be better used.

Secondly, we have observed strange effects for some tests and different rulebases (execution times are sometimes smaller for larger rulebases). Apparently, ObjectStore internals are responsible for this behavior, and more investigations using performance analysis tools [e.g., 18] are necessary in order to understand these effects. On the other hand, these tests show that response time *can* be decreased, and the challenge is to enforce such improvements deliberately. Hence, the tests show possibilities for ADBMS tuning, which also will be a topic of our future work.

More figures are necessary in order to assess the performance of composite event detection. It would be nice to have performance figures for applications that require reactive behavior but are implemented on top of a passive DBMS. It then might (or might not) turn out that — though time spent for composite event detection is large — it is still smaller than the time needed to perform equivalent tasks in a passive system. Note further that usually the total time needed to detect composite events is not spent in one piece, but typically is required in slices distributed among multiple executions of applications.

5 Conclusion and Future Work

We have presented a benchmark for active object-oriented database management systems, and have tested the ADBMS SAMOS with this benchmark.

As designers, we are particularly interested in identifying inefficient components. In this respect, we have seen that the most complex SAMOS component — composite event detection — is also the most expensive one (by orders of magnitude), and that management and retrieval of event information is not yet tolerable. These components are the ones most worthwhile to be optimized and tuned.

In order to definitely assess the performance of SAMOS, we need comparative figures for other systems. We have measured the performance of ACOOD [1], and will run BEAST on NAOS [7] and possibly other systems in the near future. Further systems will be tested as soon as they are available for us.

BEAST currently tests ADBMS in single-user mode, while results may be quite different when multi-user mode is investigated as well. Especially, it is interesting whether performance of composite event detection depends on the rulebase and the number of concurrently active transactions. However, ADBMSs are difficult to measure in multi-user mode, and finding the right “transaction mix” is a problem, too. We are currently investigating concurrency on composite event detectors, however in an analytical way.

Acknowledgments

We gratefully acknowledge the discussions with Dimitris Tombros on the BEAST benchmark and the work of Hans Fritschi on the SAMOS implementation.

References

1. M. Berndtsson, B. Lings: *On Developing Reactive Object-Oriented Databases*. Bulletin of the TC on Data Engineering 15:1-4, 1992.
2. A.P. Buchmann: *Active Object Systems*. In A. Dogac, T.M. Ozsu, A. Biliris, T. Sellis (eds): Advances in Object-Oriented Database Systems. Computer and System Sciences Vol 130, Springer, 1994.
3. A.P. Buchmann, J. Zimmermann, J.A. Blakeley, D.L. Wells: *REACH: A Tightly Integrated Active OODBMS*. Proc. 11th Intl. Conf. on Data Engineering, Taipei, Taiwan, March 1995.
4. M.J. Carey, D.J. DeWitt, J.F. Naughton: *The 007 Benchmark*. Proc. ACM SIGMOD Intl. Conf. on Management of Data, Washington, DC, May 1993.
5. R.G.G. Cattell, J. Skeen: *Object Operations Benchmark*. ACM ToDS 17:1, 1992.
6. S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, R.H. Badani: *ECA Rule Integration into an OODBMS: Architecture and Implementation*. Proc. 11th Intl. Conf. on Data Engineering, Taipei, Taiwan, March 1995.
7. C. Collet, T. Coupaye, T. Svensen: *NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System*. Proc. 20th Intl. Conf. on Very Large Data Bases, Santiago, Chile, September 1994.
8. U. Dayal: *Active Database Management Systems*. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988.
9. U. Dayal, E. Hanson, J. Widom: *Active Database Systems*. W. Kim (ed): Modern Database Systems. ACM Press / Addison Wesley, 1995.
10. S. Gatziu, A. Geppert, K.R. Dittrich: *Integrating Active Mechanisms into an Object-Oriented Database System*. Proc. of the 3rd Intl. Workshop on Database Programming Languages (DBPL), Nafplion, Greece, August 1991.
11. S. Gatziu, K.R. Dittrich: *Detecting Composite Events in an Active Database Systems Using Petri Nets*. Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, February 1994.
12. A. Geppert, S. Gatziu, K.R. Dittrich: *Performance Evaluation of an Active Database Management System: 007 Meets the BEAST.* Technical Report 94.18, Computer Science Department, University of Zurich, November 1994.
13. A. Geppert, S. Gatziu, K.R. Dittrich: *Architecture and Implementation of an Active*

Object-Oriented Database Management System: the Layered Approach. Technical Report, Institut fuer Informatik, Universitaet Zuerich, 1995.

14. J. Gray (ed): *The Benchmark Handbook for Database and Transaction Processing Systems*. 2nd ed., Morgan Kaufmann Publishers, 1993.
15. N.H. Gehani, H.V. Jagadish, O. Shmueli: *Composite Event Specification in Active Databases: Model & Implementation*. Proc. 18th Conf. on Very Large Data Bases (VLDB), Vancouver, British Columbia, Canada, August 1992.
16. R. Jain: *The Art of Computer Systems Performance Analysis. Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.
17. D.R. McCarthy, U. Dayal: *The Architecture of an Active Data Base Management System*. Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, Oregon, May/June 1989.
18. *Quantify User's Guide*. Pure Software Inc., 1992.
19. Sybase Inc.: *SYBASE - Data Server.* Berkeley, CA, 1988.

Appendix A The BEAST Rule Schema

The rule schema is given in pseudo-syntax in Table 3. The first four columns of this table are self-explanatory. The column "CM" specifies the coupling mode of the rule, and "P" defines priorities. For event definitions, we use the following conventions:

- properties of objects (instance variables and methods) are referred to through the dot notation,
- transaction events are represented as "BOT" and "EOT", followed by the name of the transaction whose begin or commit has to be detected,
- the prefixes "Ev.." is used for abstract events,
- ";" is the event constructor for sequences,
- "&" is the event constructor for conjunctions,
- "|" is the event constructor for disjunctions,
- "!" is the event constructor for negative events (the `within` clause is used to express the time interval in which the event should not occur),
- `times` is the event constructor for repeated occurrence,
- `oid` that is used in the tests RE-04 and RE-05 is an event parameter representing an instance of class `Document`,
- `DoNothing`, `searchString`, `replaceText`, `setAuthor`, and `setDate` are methods of 007 classes, and
- `GenerateAtomicPart` is the name of a transaction program.

<table>
<tr><th>Test</th><th>Event</th><th>Condition</th><th>Action</th><th>CM</th><th>P</th></tr>
<tr><td>ED-01</td><td>update (AtomicPart.docId)</td><td rowspan="4">FALSE</td><td rowspan="4">;</td><td rowspan="4">i/i</td><td rowspan="4">-</td></tr>
<tr><td>ED-02</td><td>before AtomicPart.doNothing</td></tr>
<tr><td>ED-03</td><td>EOT GenerateAtomicPart</td></tr>
<tr><td>ED-04</td><td>EvED-04i (i=1..10)</td></tr>
<tr><td>ED-06</td><td>EvED-061 ; EvED-062</td><td rowspan="7">FALSE</td><td rowspan="10">;</td><td rowspan="8">i/i</td><td rowspan="14">—</td></tr>
<tr><td>ED-07</td><td>! EvED-07 within (BOT GenerateAtomicPart ,
EOT GenerateAtomicPart)</td></tr>
<tr><td>ED-08</td><td>times (EvED-08, 10)</td></tr>
<tr><td>ED-09</td><td>times (EvED-091, 3);(EvED-092 | EvED-093);EvED-094</td></tr>
<tr><td>ED-10</td><td>Module.doNothing & Module.setDate: same_object</td></tr>
<tr><td>ED-11</td><td>update (AtomicPart.x) & update (AtomicPart.y):
same_transaction</td></tr>
<tr><td>RM-01</td><td>EvRM-01</td></tr>
<tr><td>RE-01</td><td>EvRE-01</td><td rowspan="3">TRUE</td></tr>
<tr><td>RE-02</td><td>EvRE-02</td><td>i/def.</td></tr>
<tr><td>RE-03</td><td>EvRE-03</td><td>i/dec.</td></tr>
<tr><td rowspan="4">RE-04</td><td rowspan="4">EvRE--04</td><td rowspan="4">oid->.searchString (“I am”)
> 0</td><td>cout << “doc contains word I am”</td><td rowspan="8">i/i</td></tr>
<tr><td>oid.replaceText(“I am”, “This is”)</td></tr>
<tr><td>oid->setAuthor()</td></tr>
<tr><td>oid->setDate()</td></tr>
<tr><td rowspan="4">RE-05</td><td rowspan="4">EvRE-05</td><td rowspan="4">oid->.searchString (“I am”)
> 0</td><td>cout << “doc contains word I am”</td><td>1</td></tr>
<tr><td>oid.replaceText(“I am”, “This is”)</td><td>2</td></tr>
<tr><td>oid->setAuthor()</td><td>3</td></tr>
<tr><td>oid->setDate()</td><td>4</td></tr>
</table>

Table 3. The BEAST Rule Schema