Performance Evaluation of Object-Oriented Active Database Systems Using the BEAST Benchmark

Andreas Geppert
Department of Computer Science, University of Zurich, Winterthurerstr. 190, CH-8041 Zürich, Switzerland. E-mail: geppert@ifi.unizh.ch

Mikael Berndtsson
University of Skövde, Box 408, S-541 28 Skövde, Sweden. E-mail: spiff@ida.his.se

Daniel Lieuwen
Lucent Technologies/Bell Labs Innovations, Bell Laboratories, 700 Mountain Ave., 2B-212, Murray Hill, NJ 07974, USA. E-mail: lieuwen@research.bell-labs.com

Claudia Roncancio
University of Grenoble, Lab. LSR - IMAG, BP 72, F-38402 Saint Martin d'Hères Cedex, France. E-mail: Claudia.Roncancio@imag.fr

This paper uses the BEAST benchmark to present the first comprehensive performance study of object-oriented active database management systems (ADBMS). BEAST stresses the performance-critical components of active systems: event detection, event composition, rule retrieval, and rule firing. Method invocation events and transactional events are taken into account. Four systems, namely ACOOD, NAOS, Ode, and SAMOS, have been tested with the benchmark tests of BEAST. The performance measurements demonstrate achievements in the area of active database technology, but also indicate trade-offs (e.g., between performance and functionality). Finally, the benchmark identifies optimizations and provides hints to ADBMS designers about producing systems with adequate performance and functionality — as well as some open issues.

1. Introduction

In recent years, *active database management systems* (ADBMS) (e.g., [34]) have become a hot topic of database research, and restricted ADBMS-functionality is already offered by some commercial systems (e.g., [30, 31]). 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 and/or DB-administrator). Applications using reactive behavior do thus not require "polling" techniques in order to detect relevant situations.

Like any system, an ADBMS should implement its functionality *efficiently*. Indeed, performance issues have recently been considered as one of the most important topics to be addressed to meet the requirements of applications and potential users [33]. Furthermore, performance aspects also play a crucial role from a system point of view:

- ADBMS researchers have developed different techniques for ADBMS tasks such as composite event detection [9, 17, 19]; thus, it is necessary to compare the performance of these approaches.
- Different architectural approaches have been developed and need to be compared (e.g., integrated v. layered architectures [5]).

Systematically benchmarking prototype ADBMS will help advance the state of the art by allowing those building new ADBMS to properly analyze the trade-offs of the various implementation strategies. It will also help improve existing ADBMS and identify open issues in ADBMS construction. The object-oriented systems we have tested are prototypes, and as such are available on a restricted set of platforms. Furthermore, their functionality is not yet standardized, and they thus exhibit a

Received June 1, 1997; accepted February 19, 1998
Recommending editor: Ron Morrison

high degree of heterogeneity with respect to functionality and applied implementation techniques. As a consequence, the performance of these systems cannot be compared in a straightforward manner. Nevertheless, we regard benchmarking of these prototypes as a crucial effort, helping developers understand performance characteristics and trade-offs when incorporating active functionality in products.

Currently, performance measurements even for single ADBMS are quite rare [22, 28]. In this paper we therefore first identify performance-critical aspects of ADBMS and then introduce the BEAST benchmark [22] (BEnchmark for Active database SysTems). We also report on results obtained from applying BEAST to four object-oriented ADBMS (ACOOD [2], NAOS [11], Ode [29], and SAMOS [18]).[1] We interpret the benchmark results obtained for each system and make some general conclusions. The interpretations not only show achievements in recent ADBMS-research but also illustrate performance drawbacks and open problems with respect to performance. Moreover, BEAST verifies some assumptions made elsewhere on the performance of ADBMS, while rejecting others.

BEAST focuses on basic ADBMS-tasks such as event detection, rule retrieval, and rule execution. It is intended primarily for testing the active functionality of DBMS, since appropriate benchmarks for passive DBMS have already been developed (e.g., [7, 25]). Furthermore, we concentrate on *object-oriented* ADBMS, since — although we focus on the active part — the underlying data model has some influence on ADBMS performance. Finally, BEAST is a single-user benchmark, because developing a benchmark is a complex task anyway and considering performance in multi-user mode aggravates complexity significantly.

The next section gives a short introduction of ADBMS, performance-critical aspects of ADBMS, and the tested systems. Section 3 describes the benchmark, and Section 4 presents the results. Section 5 concludes the paper.

2. Active Database Management Systems

In this section, we give a short introduction of ADBMS. Details can be found in [34]. We then briefly describe the most important features of the systems tested with the BEAST benchmark.

2.1. ECA-Rules

An ADBMS is a DBMS that supports the specification and implementation of reactive behavior in addition to standard database functionality. Most ADBMS support event-condition-action rules (ECA-rules) [14] for defining reactive behavior. An event is either an explicitly specified point in time or a description of a "happening of interest" to the user (that is detectable by the database system). After an event is detected, the corresponding rule will be fired. Events can be either *primitive* (e.g., a method invocation, a transaction begin or commit, a time event, an abstract event[2]) or *composite* (e.g., conjunction, sequence, disjunction, negation, repeated occurrence). *Composite event restrictions* are conditions that must hold for component events in order to form a legitimate composition. Examples are same object (the component events are all method invocation events for the same receiver object) and same transaction (all the component events have occurred within one transaction).

A 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.

The *execution model* of an ADBMS determines how event detection, condition evaluation, and action execution are performed. The *consumption mode* defines which event occurrences to use for event composition when multiple candidates exist. Examples of consumption modes include *recent* (select the youngest possible component event occurrence and discard older ones), *chronicle* (select the oldest possible one), and *continuous* (select all existing candidates).

The *coupling modes* of a rule specify when the condition and action parts of a rule are executed with respect to the transaction that triggered the event. Typical coupling modes are *immediate* (directly after the event has been detected), *deferred* (at the end of the triggering transaction, but before commit), and *decoupled* (in a separate, independent transaction). The coupling modes for conditions relate condition evaluation to the triggering event; the coupling modes for actions relate action execution to condition evaluation. The execution model also defines how to process multiple rules that are triggered by the same event. One possibility is to let the user specify (partial) orders, e.g., by means of *rule priorities*.

2.2. ADBMS-Architecture

In order to implement reactive behavior, an ADBMS contains several components not contained in a passive DBMS. Defining ECA-rules requires a rule definition language and a corresponding compiler. Rule execution requires one or more event detectors, a rule manager (for creating/retrieving information on rules and events), and a rule execution component. The latter collects rules ready to be executed and triggers the execution of each rule in a manner consistent with its coupling mode.

BEAST considers event detection, rule management, and rule execution since they implement the three phases that comprise active behavior. They are thus contained in all ADBMS-architectures (e.g., [5, 9, 23]) and must be considered by performance measurements. After an

event occurs, it must be *detected*, i.e., ADBMS components must recognize (or be notified) that the event has happened. At the end of the event detection phase, the event is *signaled*.[3] The second phase (rule management) starts as soon as the event has been signaled and determines whether (and which) rules must be executed. Internal information that links event descriptions with rule definitions must be taken into account. In the simple case of immediate coupling, rule management is directly followed by the rule execution phase. In this phase, the triggered rules are executed.

2.3. *ADBMS-Applications*

Benchmarks usually test the performance of systems under typical applications. Thus, in order to develop an application-oriented benchmark for ADBMS, a canonical application has to be found. However, applications of ADBMS are quite varied (see, e.g., [8]), and in fact the tested prototypes are intended for rather different application domains. For instance, the tested systems have been applied in the following types of applications and systems:

- consistency constraints [21],
- control of application systems (e.g., stock trading systems) [20],
- coordination of agents in cooperative information systems [3],
- platform for process-centered software engineering platforms [12],
- execution engine in workflow management [24, 32].

Some requirements and principles are common among all the applications; e.g., an object-oriented context and composite events to express complex situations. However, other aspects differ greatly, such as the required lifetime of events, consumption modes (and thus the size of event histories), coupling modes, etc.

We therefore conclude that a single typical application for object-oriented ADBMS does not exist, and that selecting one out of the various applications for benchmarking very likely would put some systems at an advantage (namely those, that implement exactly those features required by the application), while others would be put at a disadvantage (namely those that offer too much functionality). Instead, the benchmark described below will focus on the basic functionality of ADBMS (such as event detection), and will be used to find out the impacts of ADBMS-features on performance.

2.4. *Performance-Critical Aspects of ADBMS*

In this section, we identify the aspects that are very likely to influence ADBMS-performance. We also hypothesize as to how these aspects influence ADBMS-performance. Later in the paper, we will use the BEAST benchmark to verify or falsify these hypotheses.

2.4.1. Functionality vs. performance. In general, we are interested in the relationship between functionality and performance. By functionality, we mean the expressiveness of the rule specification language, including the set of provided event type constructors, coupling modes, and further rule execution constraints such as priorities. In general, we would expect that performance benefits can be achieved if the functionality of the ADBMS is restricted. Alternatively, ADBMS-implementors might sacrifice performance for the sake of functionality.

We hypothesize that large sets of event type constructors (especially for composite events) degrade performance, since they also imply that a more complex composite event detector is required. Moreover, if restrictions on composite events can be formulated in the language (same transaction), then event composition is expected to be less efficient, since the selection of component events is slowed down.

2.4.2. Event detection. The event detection method employed is expected to have great impact on performance, since there are different strategies which can be pursued:

- event detection can be done locally or centrally,
- it can be implemented according to different techniques (e.g., finite state machines, syntax trees, Petri Nets).

Local event detection means that events are detected "within" objects or that dedicated event detectors are used. There are two varieties. Class-specific local event detectors (e.g., as in Ode) have events/rules specified in class definitions. In this case, each object (whose class defines one or more triggers) detects events that happen at that object. Since events are always class-specific, composite events involving component events defined for different classes cannot be formulated easily.[4] Event-specific local detection uses one event detector per event type (e.g., each method event type used in one or more ECA-rules has its own event detector). Programs or objects that can generate events of a certain type maintain references to the corresponding event detector, which is notified whenever the event occurs. This approach partitions the global event detector. It is thus beneficial with respect to performance, since the relevant portion of the "global" event detector need not be identified upon event occurrences, and because each detector can exactly maintain the information (e.g., on component events) that is needed.

The centralized approach uses a small set of event detectors, each one being responsible for a class of event types (e.g., transaction events, composite events). Thus, all events are signaled through the same function, and relevant structures needed for composite event detection must first be identified/located before composite event

detection can proceed and rules to be executed can be determined. Such a centralized detector is likely to become a bottleneck.

In event-specific and centralized approaches, event and rule definitions are likely to be stored as database objects, while in the class-specific approach they are part of class definitions. Since storing ECA-rules as database objects means pursuing a (partially) interpretative approach, we expect performance to be worse than for a compilation-based alternative using class-specific ECA-rules and event detection.

2.4.3. Event (history) management and event consumption. Since a composite event is built using component events, the composite event detector has to check whether there are candidate components available, and — if yes — it has to select appropriate ones for forming the new composition. Thus, it is performance-critical to maintain these histories in a way that allows fast retrieval of component occurrences, since the set of component candidates can be very large.

The consumption mode can also be performance-critical — especially if large sets of component occurrences exist (and therefore have to be browsed in order to determine components for composition). In particular, we expect the *recent* mode to be the most efficient consumption mode, since it is rather immune to large backlogs of component occurrences.

2.4.4. Rule execution. Condition evaluation and action execution also influence ADBMS-performance. In the brute-force solution, each condition evaluation or action execution interrupts the application program execution and the ADBMS-process; if there are multiple rules to be triggered, the ADBMS executes them sequentially. In more sophisticated implementations, the ADBMS is able to evaluate conditions and to execute actions concurrently with its "normal" processing. If there are multiple rules to be executed at one point in time, they could also be executed concurrently (unless their execution order is constrained by priorities). Such a concurrent execution is likely to speed up the ADBMS.

A sophisticated ADBMS might apply techniques proposed for query evaluation and production rule processing to optimize condition evaluation (e.g., several similar or even identical conditions may have to be evaluated together at times; then, the optimizer could recognize that the common parts need to be evaluated only once, decreasing overall condition evaluation time).

2.4.5. Impact of architectural style. By "architectural style," we mean how the ADBMS is structured. In an *integrated architecture*, the active components are integrated with the passive parts. Thus, components such as event detectors and rule execution components can easily interface with other components and can access their information. It is also possible to modify or extend the passive part when implementing the active behavior. In the *layered architecture*, the active functionality is implemented on top of a passive one, and passive components cannot be extended or modified. It has been claimed elsewhere [5] that the better opportunities to integrate active and passive components in the integrated architecture leads to better performance.

2.4.6. Impact of (growing) rulebase size. While the aforementioned aspects refer to the ADBMS itself and its components, the final performance-critical aspect is the rulebase, i.e., the set of currently defined event types and rules. A practically useful ADBMS must be able to handle large rulebases with reasonable performance, i.e., it should scale well for growing rulebases. Since indexing techniques can be used to efficiently retrieve event/rule definitions, we generally expect that event detection and rule execution performance is independent from the rulebase size (i.e., constant).

Furthermore, the ADBMS should not only scale well for growing numbers of event types, but also for growing sets of event occurrences (i.e., large event histories). At least for composite events without restrictions such as same transaction, we also expect that event detection performance does not depend on the size of the event history.

2.4.7. Cross-effects. Each of these aspects cannot be considered separately, but there will typically be dependencies between them. For instance, performance can be traded for functionality, and the efficiency of an event detection technique depends on the event consumption mode.

2.4.8. Summary. Below we summarize the aforementioned hypothesis.

- **H1:** More expressive power of the ECA-rule model leads to worse runtime performance.
- **H2:** A large set of event type constructors degrades runtime performance.
- **H3:** Support for composite event restrictions degrades performance.
- **H4:** Centralized event detectors perform worse than dedicated or class-specific event detectors.
- **H5:** Class-specific ECA-rules and event detection are more efficient.
- **H6:** Support for event histories degrades performance.
- **H7:** The recent consumption mode is the most efficient one.
- **H8:** Concurrent rule execution improves performance.

H9: Incremental condition evaluation of sets of conditions improves rule execution performance.

H10: Integrated architectures are more efficient than layered architectures.

H11: ADBMS-performance should be independent of the rulebase size.

H12: Event detection should scale well for growing event histories.

2.5. *The Tested Prototypes*

We have tested four ADBMS prototypes whose relevant features are briefly described below:

- ACOOD [2, 15] is built on top of the commercial OODBMS ONTOS DB 3.0,
- NAOS [11, 10] which extends the OODBMS O_2 [1] (version 4.5.6),
- Ode [29] exists in two variants: a disk-based version built on EOS [4] and a main-memory version built on Dali [26] (the disk-based one was benchmarked), and
- SAMOS [18] uses the commercial OODBMS ObjectStore as a platform.

All the systems support method events and abstract events; only ACOOD does not offer transaction events. The systems support different sets of composite event constructors; however most of those required for the BEAST tests can be expressed in the rule definition language of each system. The systems use four different techniques for composite event detection: arrays (ACOOD, [15]), extended finite state machines (Ode, [19, 29]), event graphs (NAOS, [10]), and Petri Nets (SAMOS, [17]). The provided consumption modes [9] are chronicle (Ode, SAMOS), continuous (NAOS), and recent (ACOOD).

ACOOD and SAMOS allow arbitrary Boolean functions as conditions. Conditions in NAOS are written in the Object Query Language (OQL). In Ode, *masks* can be defined, which are conditions that must be evaluated to determine if a (sub)event of an event has occurred or not.[5] Triggers must be *activated* to have any effect. There can be many activations per trigger.

Actions in each of the systems are arbitrary statements in the underlying database programming language (C++ for ACOOD and SAMOS, O2C for NAOS, the C++-extension O++ for Ode). ACOOD and SAMOS can pass event parameters to conditions and rules; in Ode, parameters used in masks/actions are passed to the associated trigger at activation time. NAOS can pass parameters only in case of update events.

3. BEAST: A Benchmark for ADBMS

In this section, we first identify design decisions for the BEAST benchmark (see [27] for how to design a benchmark). We then describe BEAST in detail.

STYLE OF TESTS

BEAST is intended to test the basic functionality of ADBMS and to determine performance drawbacks of ADBMS-designs and implementations. It does not propose a typical application or test the performance of ADBMS for such an application (see Section 2.3). We are thus measuring the performance of ADBMS on a micro level from a designer's perspective.

INFLUENCE OF PASSIVE COMPONENTS

ADBMS use the functionality of passive DBMS. They need services from the passive part, such as persistence, transactions, and query processing. BEAST thus tests the entire active DBMS, and the performance of passive parts typically will influence ADBMS-performance. We do not test ADBMS at a finer-grained level (e.g., by turning off locking/logging) because the required functionality is not available in all systems. Consequently, an ADBMS that uses a slow platform or does not exploit the capabilities of the underlying system in an optimal way will incur a performance penalty. Nevertheless, as the various measurements have shown, BEAST makes it possible to identify performance bottlenecks of the active part by comparing the different tests performed for a specific system.

SELECTION OF METRICS

Our major metric is elapsed time. Each BEAST test repeatedly raises events, which are then detected by the ADBMS. Elapsed time is defined as the time that passes until control returns to the test program (i.e., after rule execution completes).

3.1. *Benchmark Design*

BEAST uses the schema and the programs of the OO7 benchmark [7] to populate the test databases. One reason for reusing parts of OO7 is to easily obtain a schema and database. Moreover, for a given object-oriented ADBMS, BEAST and OO7 together measure the performance of both the active and the passive parts of a system, respectively.

BEAST defines several tests for event detection, rule management, and rule execution (see Section 2.2). Thus, the result of running BEAST is a collection of figures instead of a single figure for each ADBMS (much like OO7). 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 oc-

currence. To stress the 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, so that condition evaluation is cheap and no action is executed. Additionally only one rule is triggered by such an event to minimize the rule management overhead.

BEAST tests each ADBMS in single-user mode, i.e., the performance of systems in the presence of multiple application programs concurrently generating events is not measured. In this context, "single-user" mode means that BEAST test programs are the only active application programs, while functionality pertaining to multi-user mode (e.g., concurrency control) is not turned off during testing. The reason for this restriction is that we first wanted to develop systematic performance evaluation tests for ADBMS and their implementation techniques, a problem that is significantly aggravated for multi-user mode (see [6] for a discussion of the problems that have been encountered for the development of a multi-user OO7). For instance, the number of dimensions/factors that have an impact on performance grows significantly, and there are much more complex interrelationships between them with respect to performance.

Next, we describe each test and show the rule(s) in pseudo-code (see Table 1, where "CM" stands for "coupling mode" and "P" for "priority"). Note that the tests are not always enumerated consecutively, since some of the ones originally proposed [22] have been omitted in this paper (e.g., because some of the tested systems do not support the required functionality).

3.1.1. Tests for event detection. Event detection tests focus on the time it takes to detect primitive or composite events.

TESTS FOR PRIMITIVE EVENT DETECTION

Three BEAST tests refer to primitive event detection:

1. detection of object updates (ED-01),
2. detection of method invocation (ED-02),
3. detection of transaction events (ED-03).

The tests ED-01, ED-02, and ED-03 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 actions and conditions are immediate.

We illustrate the execution of tests with the test ED-02. First, the actual time is obtained, and then the event is forced to occur multiple times (in this case, a method is invoked). Note that in this way we know the point

TABLE 1. The BEAST rule schema.

Rule	Event	Condition	Action	CM	P
ED-01	update(AtomicPart)	false	alert ...	imm	-
ED-02	before AtomicPart->DoNothing				
ED-03	before commit(ED03_TX)				
ED-06	EvED-061 ; EvED-062				
ED-07	! EvED-07 within [begin(ED07_TX), commit(ED07_TX)]				
ED-08	times (EvED-081, 10)				
ED-09	times (EvED-091, 3); (EvED-092 | EvED-093) ; EvED-094				
ED-10	Module->DoNothing; Module->setDate: same object				
ED-11	AtomicPart->setX & AtomicPart->setY: same transaction				
RM-01	EvRM-01				
RE-01	EvRE-01	true		imm.	
RE-02	EvRE-02			def	
RE-03	EvRE-03			dec	
RE-04a	Document->DoNothing	o->searchStr("I am") > 0	alert...	imm	
RE-04b			o->setAuthor()		
RE-04c			o->setDate()		
RE-04d			o->replaceTxt("I am", "This is");		
RE-05a	Document->DoNothing2	o->searchStr("I am") > 0	alert...		1
RE-05b			o->setAuthor()		2
RE-05c			o->setDate()		3
RE-05d			o->replaceTxt("I am", "This is")		4

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 elapsed time.

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 detected event participates in a composite event. This is generally done in a stepwise manner, e.g., by means of syntax trees [9], automata [19], or Petri nets [17]. Composite event detection is measured by 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 signaling is required and primitive event *detection* is not necessary. In order to measure the entire event composition, the tests raise the component events directly one after the other.

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 (ten times) 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 occurring for the same object (ED-10),
6. detection of a conjunction of events raised within the same transaction (ED-11).

Since we are interested in the time for event detection, 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, ED-10 and ED-11 measure the performance of event detection when the events of interest are 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 used infrequently, and thus their efficient implementation is less important.

The test RM-1 raises an abstract event, evaluates a condition to false, and therefore does not execute any action. The three parts are kept that 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 separated into two groups: one for the execution of single rules, and one for the execution of multiple rules. The first group of tests (RE-01 through RE-03) determines how quickly 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. Different approaches for linking and processing condition and action parts 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., sequential or concurrent execution). The performance characteristics of these approaches are tested by the second subgroup.

For the execution of single rules, we consider three rules with different coupling modes. An abstract event is used, the condition is always true, and the action is an alert command in rules RE-01, -02, and -03. The coupling mode of the condition is always immediate. The coupling modes of the actions are immediate (RE-01), deferred (RE-02), and decoupled (RE-03). 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 a simple 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 fourth test (RE-04) for rule execution considers four rules all triggered by the same event. Conditions and actions are more complex than in the previous tests, in order to observe the effects of optimizing the condition evaluation and of concurrency. All RE-04 rules have the same condition. Hence, an ADBMS that recognizes that the actions do not affect the truth of the condition and that the conditions are identical (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). No ordering is defined for the four rules. An ADBMS that is able to process conditions and actions in parallel or at least concurrently will thus perform better in this test. RE-05 is similar to RE-04, with the exception that priorities are specified.

3.2. Factors and Modes

A crucial step when designing a benchmark is the proper identification of *factors* [27], 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, etc.), these include:

- the number of defined events,
- the number of defined rules,
- the number of partially processed (i.e., not yet consumed) events in the system.

In the ideal case, the time to detect events is constant, i.e., independent of the number of defined events. However, especially for composite events, it may be the case that the event detection process for single events slows down as more events are added to the system. 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 worth investigating how large execution times are when the number of events increases. This number is therefore included as a factor. In general, about 50% of the events are defined as composite events.

Furthermore, the total number of defined rules 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 — rules must be selectively loaded upon rule execution. Thus, determining how efficiently an ADBMS can handle large sets of rules and how the system behaves when the number of rules grows larger is important.

Ultimately, the performance of composite event detectors can depend on the previous event history. Specifically, we expect the performance of event composition to depend on the number of events that are candidate components for composite event detection. For the tests ED-06 and ED-09 through ED-11, the number of component events which are used to initialize the composite event detector is thus a parameter.

For the three factors, we choose four possible values for an empty, a small, a medium, and a large rulebase (see Table 2). Tests for larger rulebases are simple to produce, since the values of all factors can be specified as parameters of the rulebase creation program. 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 indicate whether the ADBMS is able to handle large sets of rules with a performance comparable to small numbers of rules.

TABLE 2. Parameter values for different rulebase sizes.

	rulebase size			
factor	empty	small	medium	large
#events	0	50	250	500
#rules	0	50	250	500
# of component event occurrences	0	25	50	100

3.3. Benchmark Implementation

In order to run the benchmark for a concrete ADBMS, the OO7 schema must be defined for the tested system and OO7 databases must be created. The next step specifies/compiles the ECA-rules for the system and the chosen configuration (see Section 3.2). In the final step, the desired tests are executed (see Figure 1). Each system was tested with several dozen test iterations. Each test was run once per iteration; in each test, the corresponding rule(s) was (were) triggered a fixed number of times (currently, 10).

Each test computes the time that is spent for its execution. We primarily consider elapsed time, but also record CPU-time (due to the fact that this process is subject to operating system scheduling, process-specific CPU-time can be a fraction of the total elapsed time).

BEAST is a fairly generic benchmark that should apply to any ADBMS. Nevertheless, we also encountered several problems in implementing it for the various ADBMS. These problems are mostly related to functional differences apparent in the tested systems.

The prime difference is between Ode and the other three tested systems. While ACOOD, NAOS, and SAMOS implement the paradigm of traditional ECA-rules as described above, Ode uses class-specific triggers. Moreover, while an ECA-rule in one of the three systems is either enabled or disabled, triggers in Ode are activated per object on the basis of activation parameter values. Thus, many activations of a trigger can exist for one object.

During the implementation, several decisions had to be made that (potentially) affect fairness of the benchmark tests. First, while dummy rules are not attached to any specific class in the three systems, they have to be assigned to classes in Ode. Assigning as many as possible dummy rules to a class that also defines triggers used in the tests would slow down Ode. In the other extreme, no dummy triggers for those classes would put Ode at an advantage. Similarly, what is the correct number of activations used for the tests? Since this concept does not exist in other systems, it is hard to find a fair solution. No activations for the event detection tests would favor Ode, while too many activations would again put it at a disadvantage.

Similar problems have been encountered for the other systems. While SAMOS stores each component event occurrence in the event history until it is consumed, NAOS does not store these occurrences after the session

```
for i = 1 to number_of_iterations
    for t in Tests
        for k = 1 to number_of_runs
            generate t's event
            compute elapsed time
               for execution of t
```

FIG. 1. Algorithm for executing a BEAST test series.

in which they have been raised is terminated. ACOOD — due to its consumption mode — stores only a fraction of the event history. Thus, as far as event detection is concerned, SAMOS is at a disadvantage, too.

Such problems are not specific to BEAST or its design, but we claim that they are typical for a designer benchmark. Since the major aim is to assess the performance of designs and implementations in general, finding fair benchmark implementations for all systems might be hard where there are great functional or architectural differences. Alternatively, in an application-oriented benchmark, the focus is on performance of entire systems for typical applications. Once such a typical application has been agreed upon, finding fair implementations is easier — whichever system offers "too much" functionality will suffer performance penalties, and systems functionally inadequate will not pass the tests.

This situation has several implications on how to approach a designer benchmark. First, fairness is more likely whenever systems are evaluated within a collaborative effort involving designers from each tested system. Second, the obtained results can be used to assess performance of implementation techniques and bottlenecks of each system. A generic benchmark is important, so that — apart from the core tests — more tests or configurations can be added and customized so that the performance problems of the system under consideration can be identified. However, the results obtained cannot be used to compare systems directly, i.e., to draw conclusions as to which system definitely is the fastest one.

4. Benchmark Results

In this section, we present the results obtained by running BEAST on each of the four systems.

All the results represent elapsed time in milliseconds (ms). Further statistical data (including min/max values, standard deviation, and 90% confidence intervals [27]) have been computed to calibrate the test series, but are not reported here. Below we present the results and then discuss them in Section 4.5.

4.1. Results for ACOOD

ACOOD has been tested on a SUN SPARC-Server10/51 under SUNOS 4.1.3.

ACOOD scales well (Figure 2), i.e., the measured times are almost constant and independent of the rule-base size. The functionality in ACOOD is restricted to the recent event consumption mode and the immediate coupling mode. This increases the performance in ACOOD as, e.g., the recent consumption mode is not sensitive to the event history. On the other hand it also restricts the number of BEAST scenarios that can be tested on ACOOD. As a consequence, test results for RE-02 and RE-03 are not available.

The results obtained for event detection, rule management, and rule execution are acceptable for our purposes. ACOOD is based on an array technique [15] for detection of composite events which is efficient for the detectable events. Rules in ACOOD are indexed by events [2], which speeds up the process of selecting triggered rules. Due to the architectural style of ACOOD (layered approach), we have not been able to efficiently implement transaction events. Thus, results for ED-03 and ED-11 are missing. ACOOD does not offer a large set of event type constructors. For instance, it is not possible to specify the detection of a non-occurrence of an event. As a consequence, results for ED-07 are not available.

Rule execution in ACOOD is acceptable, but it can be considerably improved since the current implementa-

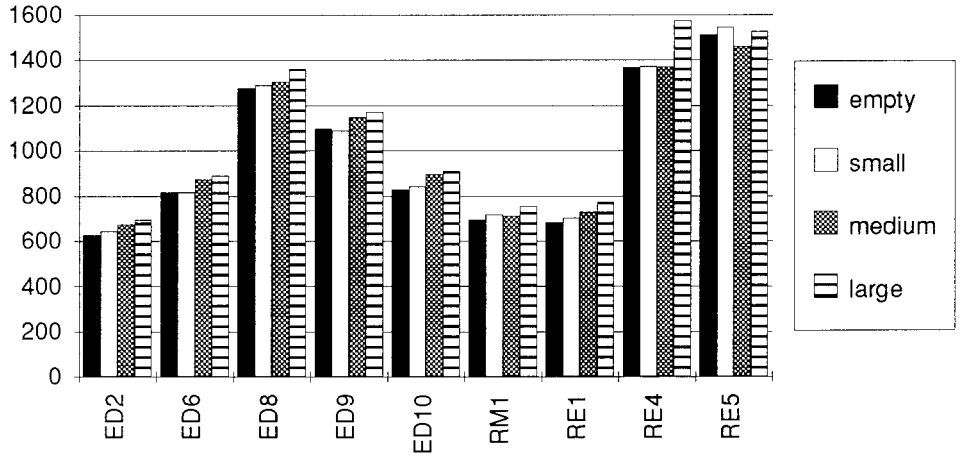

FIG. 2. Results for ACOOD.

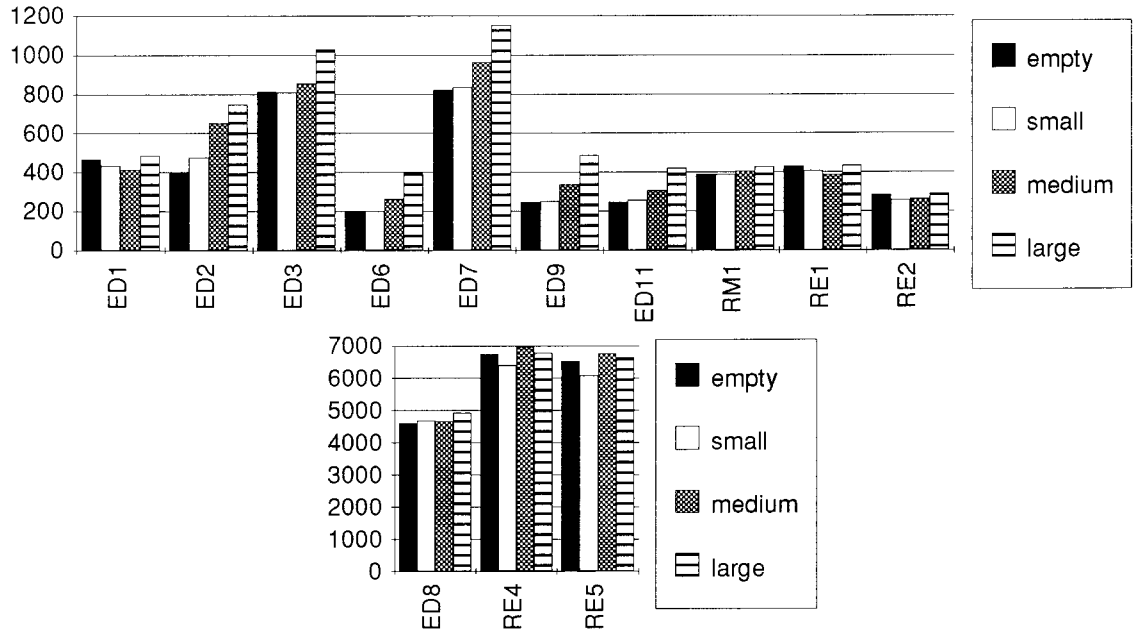

FIG. 3. Results for NAOS

tion does not support optimization of conditions, concurrent evaluation of conditions, and concurrent execution of rules.

4.2. *Results for NAOS*

NAOS has been tested on a Sun Sparc Station IPX under SunOS 4.1.3 using O2 (version 4.5.6). It scales well for growing rule bases (Figure 3).

The time obtained for ED-03 comes mainly from the validation and begin transaction statements necessary to test. The same test performed without rule execution spends almost the same time (100ms for the large configuration). So the global overhead introduced here by rules is negligible. The times for composite event recognition (ED-06, -07, -09, -11) are fine except for ED-08 (10 occurrences of the same event). For this test we consider a sequence of 10 events as NAOS does not currently

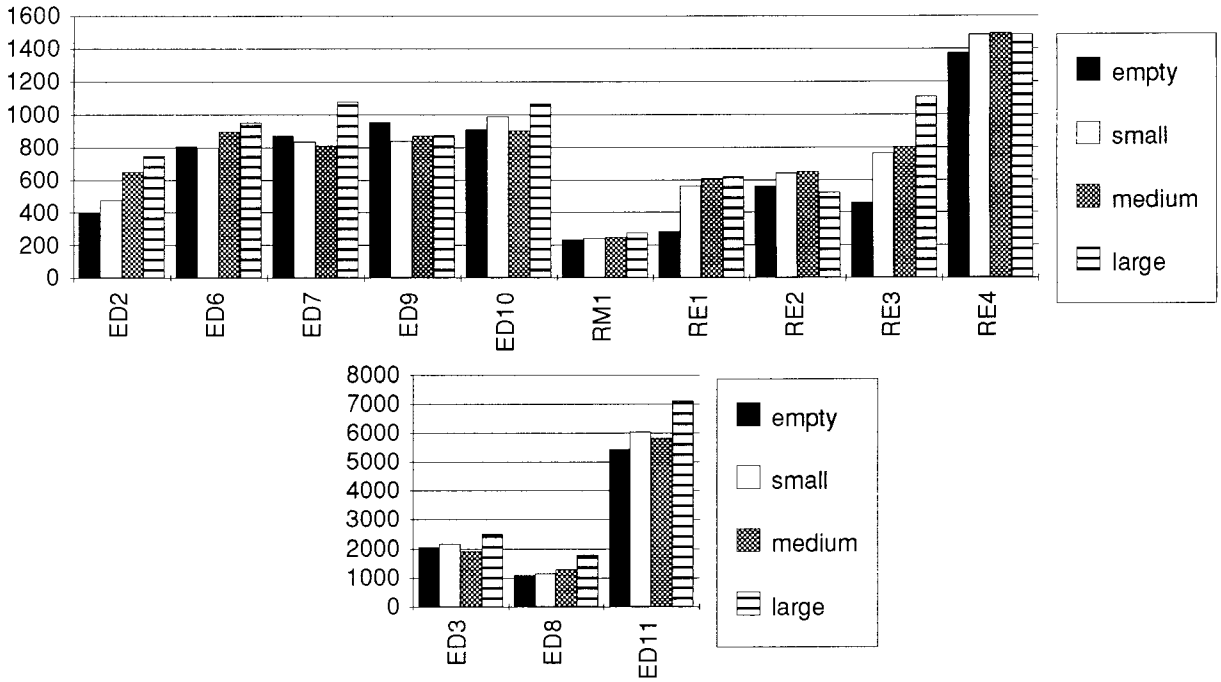

FIG. 4. Results for Ode.

offer a specific times-operator. ED-11 is good because same transaction is an implicit restriction in NAOS. Since NAOS does not maintain the event history across different sessions, there is no performance penalty for growing event histories.

Rule retrieval time (RM-01) is quite good because NAOS works with a main memory representation of rules (created at database open time) indexed by event types. The results for RE-04 and RE-05 show that there is no overhead due to the rule ordering. These results were expected as rules are ordered at definition time. However, since the rules are executed sequentially, the global time for these tests is not really good. For each rule the condition is evaluated to take into account the effects of the rules executed before. In tests RE-04 and RE-05 the method used in the condition is time consuming (400ms).

4.3. *Results for Ode*

Ode has been tested on a SUN-SparcServer 4/690 under SUNOS 4.1.3 (see Figure 4).

An Ode trigger must be activated or it will never fire. If the corresponding event occurs, the event mask is evaluated, and if it evaluates to true then the trigger is fired. Given that Ode identifies both complex and primitive events using the same extended finite state machine mechanism [29], it takes exactly the same amount of time to detect that an event of interest has occurred whether the event is simple or complex unless masks (conditions) must be evaluated.[6] If a mask involves an expensive computation or if several masks must be evaluated, identifying a composite event will take proportionately more time. However, in the experiments the masks were simple enough that identifying the occurrence of either a primitive or a complex event of interest to a trigger activation took roughly the same amount of time.

Initially, we considered the event mask as the analogon to conditions in ECA-rules, and consequently specified them in such a way that they always evaluated to false for event detection tests. However, trigger activations are not deleted if this mask evaluates to false because they have never fired (and never will fire). Thus, the number of trigger activations in the system grows over time, and each activation must be alerted when an event is posted to its corresponding object. This is the reason for the increase in the measured times (e.g., for ED-08).

4.4. *Results for SAMOS*

SAMOS has been tested on a SUN-SparcServer 4/690 under SUNOS 4.1.3.

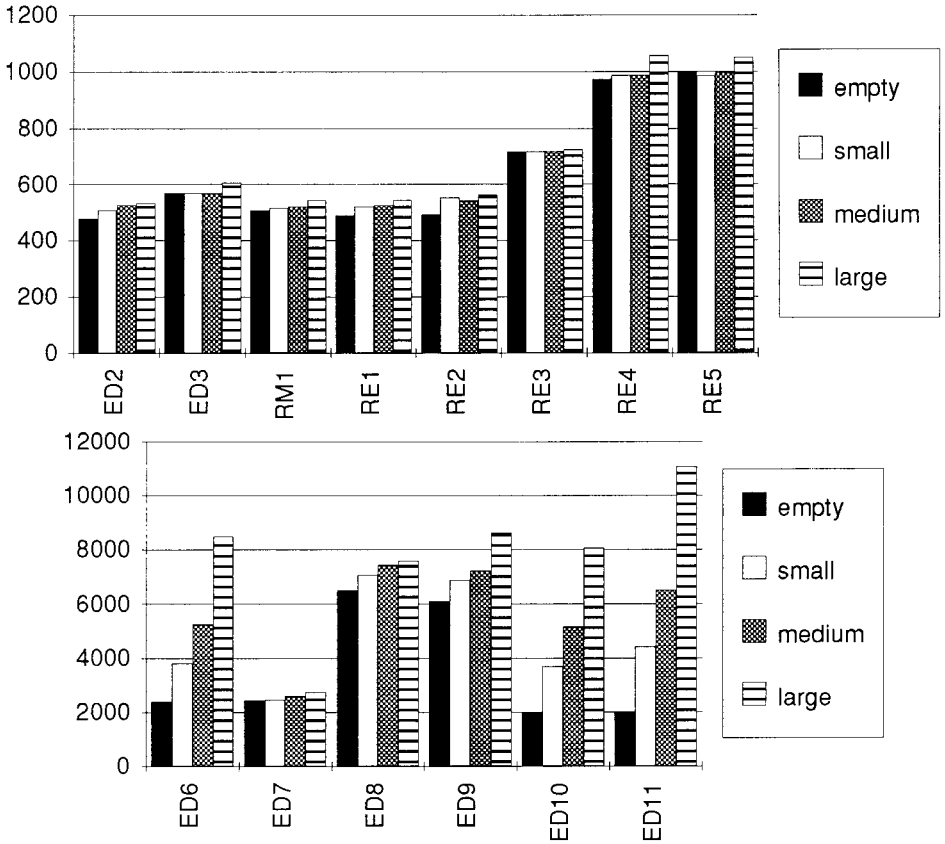

FIG. 5. Results for Samos.

The major reasons for the high execution times (see Figure 5) in SAMOS are the complexity of the system, the additional functionality it has, and the way event detection is implemented.

SAMOS scales quite well for growing rulebases as far as primitive event detection and rule execution is concerned. This is due to indexing and clustering event descriptions and rule information. Measured times are rather high for composite event detection, since (1) lots of objects forming the Petri Net used for composite event detection are stored on disk and, (2) no clustering is applied to those objects.

Furthermore, SAMOS (i.e., its composite event detector) is sensitive to the number of existing component event occurrences. In ED-11 for the large rulebase, e.g., 100 component events already exist when are raised before the tests actually start (see Table 2). These events are stored persistently and are considered for event composition during each test ED-11. Without these (useless) component events, the average execution time of ED-11 is 2515 ms for the large rulebase, and thus the execution time is almost constant for this test (see Figure 6).

4.5. Discussion of Results

We consider it an achievement that several object-oriented ADBMS-prototypes are now available. As the tests show, they perform some of their tasks in a timely manner (e.g., action execution does not seem to cause major performance problems).

Let us now generalize the performance results. We are thereby primarily considering runtime performance while neglecting design decisions related to functionality or compile time performance. We are mostly interested in understanding the performance characteristics of the various implementation techniques and architectures, while constructing a ranked list of fast and slow systems is a non-issue.[7]

4.5.1. Trade-offs. The first conclusion to be drawn from the test results is the trade-off between functionality and performance. NAOS and Ode offer less functionality than SAMOS and ACOOD in that they do not support event parameters being passed to conditions and actions for all kinds of event types. Primitive event detection and event composition are therefore potentially faster in NAOS and Ode. Second, NAOS does not support explicit event restrictions (such as same transaction), which are supported in SAMOS. Upon event composition, NAOS is thus potentially faster since it can take *any* event occurrence for composition, but does not need to select those event occurrences that fulfill the event restriction.

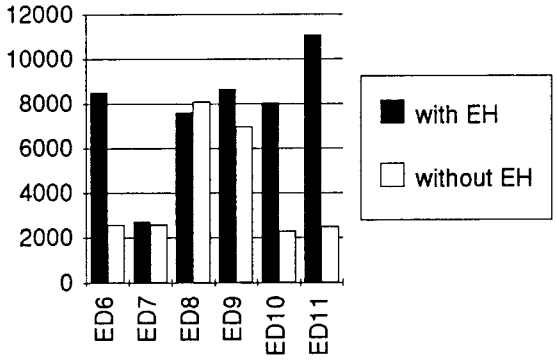

FIG. 6. Results for Samos (with and without initialization of event history).

One the other hand, event parameters and event restrictions are considered useful constructs. If they are thus desired, then one has either to build them into the ECA-rule model (as is done in Ode for the same object restriction), or to accept the runtime cost. Another trade-off is that between compile-time and runtime performance. For instance, if class-internal rules are supported, compile-time performance is worse, but runtime performance is improved.

4.5.2. Event detection. A centralized event detector is notified about all events. It then needs to retrieve the event description from the rulebase and to determine event parameters (in ACOOD/SAMOS). Furthermore, in SAMOS the composite event detector has to reconstruct the Petri Net parts it needs, which in turn are represented as objects and spread all over the database. In the local approach, most of the information is already available, since it is kept local to objects (as in Ode). This explains why Ode detects composite events faster and scales well for growing rulebases.

4.5.3. Event history management. For some tests, the number of initially raised component events is a factor, i.e., the event history is not empty when the tests start. Especially if event parameters are required for subsequent rule execution, then the event history must be maintained, either explicitly or implicitly in the state of the event detector(s). Two observations are apparent with respect to event history management:

- The recent consumption mode seems to be more efficient, since upon event composition the entire event history might have to be scanned in chronicle consumption mode. This is the reason why ACOOD (which uses the recent mode) — unlike SAMOS — is not sensitive to the size of the event history.
- If the chronicle consumption mode is used, then garbage collection of old event occurrences is a crucial task. For instance, in ED11, the initially raised component event occurrences are of no use, since a same transaction restriction is specified. Garbage collection would discard these occurrences even before the tests actually start and thus would make SAMOS five times faster for some tests.

- Two systems (ACOOD and NAOS) do not need to maintain full event histories due to their consumption modes. NAOS uses a main memory graph for event detection which is constructed when a database is opened and discarded when the database is closed. Since NAOS does not maintain the event history across multiple sessions, it does not suffer the performance penalties of SAMOS (and to a lesser degree also Ode) implied by event history management.

4.5.4. Condition evaluation. The current prototypes do not perform any kind of optimization or pre-analysis of conditions. A very basic approach might be to perform pre-analysis, during which constant expressions would be detected (none of the systems recognized the constant, false condition or mask in event detection tests). Condition evaluation optimization might also be improved, since none of the systems recognized that the same condition was used in each of the four associated rules for RE-04. Thus, optimizing *sets of conditions* together or incrementally might further improve performance of rule execution (note that this has already been investigated for relational and production-rule systems, e.g., [16]).

4.5.5. Observations on architectural styles. It is not generally justified based on our results to conclude that integrated architectures have better performance than layered architectures — performance depends on a variety of implementation choices. Even integrated architectures use some kind of lower-level platform (e.g., Ode uses EOS) and both the performance of this platform and the chosen implementation techniques significantly impact performance. Nevertheless, in integrated architectures there is a higher degree of freedom when choosing techniques (e.g., for event detection) — for instance, some of Ode's techniques are not applicable in a layered system. Thus, well-designed integrated architectures can be expected to have better performance because they can choose any appropriate technique. However, the performance difference between a layered and a non-layered architecture employing the *same* techniques may or may not be significant.

4.5.6. Summary. The measurements allow us to verify or falsify most of the hypothesis formulated above (see Table 3).

We now conclude this section by addressing two groups of questions:

- Which open problems need to be addressed in order to improve ADBMS-performance?
- What do the tests teach about better ADBMS implementations? To which extent do the tests help in decisions on implementation techniques, functionality, and optimization?

The most important open problem is the missing agreement on ADBMS-application requirements. If we can identify the functional requirements an ADBMS must satisfy, we can pick implementation techniques that offer optimal performance for those requirements.

Another still open problem is physical database design for event histories. In systems such as SAMOS, which store potentially large backlogs of event occurrences, the major part of the increase in event detection time is due to large event histories. Thus, whenever the event history must be kept persistent, sophisticated clustering and indexing techniques must be tailored to event history management in order to keep performance tolerable. Furthermore, especially when conditions are typically complex and may share parts with other conditions, sophisticated condition optimization techniques might improve rule execution performance. Finally, while most of the current systems execute rules sequentially and block the application program for the rule execution time, concurrent execution of actions can also speed up rule execution [13]. This feature requires more work to be done in rule scheduling and ADBMS-architecture (e.g., multithreading).

In addition to these open problems, the measurements allow one to draw several conclusions concerning the performance of implementation techniques. The measurements show that avoiding a central event detector and a central function used to generate events (such as raise_event in SAMOS) is crucial. Event detection is generally more efficient if primitive events can be detected either within objects or by dedicated event de-

TABLE 3. Evaluation of hypothesis.

	Hypothesis	Result
H1	More expressive power leads to worse runtime performance	partially true
H2	A large set of event type constructors degrades runtime performance	false
H3	Support for composite event restrictions degrades performance	true
H4	Centralized event detectors perform worse than dedicated event detectors	true
H5	Class-specific ECA-rules and event detection are more efficient	true
H6	Support for event histories degrades performance	true
H7	The recent consumption mode is the most efficient one	true
H8	Concurrent rule execution improves performance	could not be tested
H9	Sophisticated condition evaluation improves rule execution performance	could not be tested
H10	Integrated architectures are more efficient than layered architectures	not necessarily true
H11	ADBMS-performance should be independent on the rulebase size	true
H12	Event detection should scale well for growing event histories	true

tectors. The structures of the composite event detectors must be kept as simple as possible — the structures and the data needed by a composite event detector should not be spread across many persistent objects. Our experience is that if two event detectors have the same functionality, then the one with the more complex structure and the larger number of required persistent objects will perform worse.

Further conclusions concerning the trade-off between functionality and performance can be made. In other words, provided a thorough knowledge of the requirements to be met, we can reason about sufficiently efficient ADBMS-implementation techniques.

Whenever (in the intended application domain) rules are mostly related to single classes and there are not too many rules per class, class-specific rules generally imply a performance gain. Likewise, restricting the set of event type constructors to be as small as possible simplifies the event detector, and thus also leads to performance improvements. Furthermore, a consumption mode that requires only bounded storage per trigger (e.g., recent, chronicle as implemented by Ode) should be used whenever appropriate for the intended applications, since it leads to faster composite event detection. The event history should only then be stored persistently if application classes need to monitor composite events across sessions. Ultimately, if typically component event (parameters) are only of interest if they occur within the same transaction, then one should distinguish event detection local to transactions and global event detection (as in Sentinel [9]).

5. Conclusion and Future Work

Four ADBMS have been benchmarked. This benchmarking was not possible when the work on BEAST started in 1994, since far fewer systems were operational back then. The systems we have tested are quite powerful and efficient for certain tasks.

The tests have also helped stabilize each of the systems, since implementing a predefined benchmark determined several bugs and limitations, and also helped understanding the performance of ADBMS. Concretely, we learned about the performance characteristics of event detection techniques (using a centralized, single event detector vs. usage of many event detectors dedicated to objects or event descriptions) as well as the performance of composite event detectors. We also better understand when factors such as the rulebase size or the event history size influence performance. The remaining performance problems can be subdivided into two classes:

- trade-offs between performance and functionality in some aspects, where either functionality must be reduced or its cost be accepted (e.g., class-independent rules),
- open problems still to be addressed (e.g., event history garbage collection and condition optimization).

As for future work, it would be interesting to test further systems (e.g., Sentinel [9] or Monet [28]) as soon as they are available. Furthermore, ADBMS-performance evaluation in multi-user mode is a challenging topic.

Acknowledgments

We gratefully acknowledge the fruitful discussions about BEAST with Robert Arlein, Alex Buchmann, Klaus R. Dittrich, Andreas Eklund, Narain Gehani, and Martin Kersten.

The work of the authors from Skövde and Zürich has been supported in part by ACT-NET. ACT-NET was a HCM network funded by the Commission of the European Union; the Swiss part in ACT-NET has been funded by the "Bundesamt für Bildung und Wissenschaft," BBW. NAOS has been developed as part of the ESPRIT-project GOODSTEP.

Notes

1. REACH [5] was also tested in the beginning of 1996 but is currently unavailable to us.
2. *Abstract events* are events that are not detected by the ADBMS, but that have to be signaled explicitly by the application or the user.
3. In general the precise point in time when an event occurred is not known. However, in the BEAST tests, we enforce event occurrence and thus know this point in time.
4. Composite events involving different objects/classes *can* be detected by having triggers post abstract events to event detector objects explicitly.
5. Masks are a generalization of conditions. Rather than merely checking a single condition after an event is detected, the system may evaluate several masks in the course of detecting an event.
6. If a composite event involves several masks, more than one mask may need to be evaluated in response to a single basic event occurrence.
7. The measurements cannot be compared directly to each other due to the inherent differences of the systems as mentioned in Section 3.3 and because different hardware had to be used due to licensing problems.

References

[1] Bancilhon, F., Delobel, C., and Kanellakis, P., eds. (1992). *Building an Object-Oriented Database System. The Story of O2.* Morgan Kaufmann.

[2] Berndtsson, M., (1994). *Reactive object-oriented databases and CIM.* In: *Proc. 5-th Int. Conf. on Database and Expert System Applications,* pp. 769–778, Athens, Greece, September 1994.

[3] Berndtsson, M., Chakravarthy, S., and Lings, B. (1997). *Task sharing among agents using reactive rules.* In: *Proc. 2-nd IFCIS Conf. on Cooperative Information Systems,* pp. 56–65, Charleston, South Carolina, June 1997.

[4] Biliris, A., and Panagos, E. (1995). *A high performance configurable storage manager.* In: *Proc. 11-th Int. Conf. on Data Engineering,* pp. 35–43, Taipeh, Taiwan, March 1995.

[5] Buchmann, A., Zimmermann, J., Blakely, J., and Wells, D. (1995). *Building an integrated active OODBMS: Requirements, architecture, and design decisions.* In: *Proc. 11-th Int. Conf. on Data Engineering,* pp. 117–128, Taipeh, Taiwan, March 1995.

[6] Carey, M. J., DeWitt, D. J., Kant, C., and Naughton, J. F. (1994). *A status report on the OO7 benchmark.* In: *Proc. 9-th Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 414–426, Portland, Oregon, October 1994.

[7] Carey, M. J., DeWitt, D. J., and Naughton, J. F. (1993). *The OO7 benchmark.* In: *Proc. ACM-SIGMOD Int. Conf. on Management of Data,* pp. 12–21, Washington, DC, May 1993.

[8] Ceri, S., and Widom, J. (1996). *Applications of active databases.* In: *Active Database Systems,* J. Widom and S. Ceri, eds., pp. 259–291. Morgan Kaufmann.

[9] Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S.-K. (1994). *Composite events for active databases: Semantics, contexts and detection.* In: *Proc. 20-th Int. Conf. on Very Large Data Bases,* pp. 606–617, Santiago, Chile, September 1994.

[10] Collet, C., and Coupaye, T. (1996). *Composite events in NAOS*. In: *Proc. 7-th Int. Conf. on Database and Expert Systems Applications,* pp. 244–253, Zurich, Switzerland, September 1996.

[11] Collet, C., Coupaye, T., and Svensen, T. (1994). *NAOS: Efficient and modular reactive capabilities in an object-oriented database system.* In: *Proc. 20-th Int. Conf. on Very Large Data Bases,* pp. 132–143, Santiago, Chile, September 1994.

[12] Collet, C., Habraken, P., Coupaye, T., and Adiba, M. (1994). *Active rules for the software engineering platform GOODSTEP.* In: *Proc. 2-nd Int. Workshop on Database and Software Engineering, 16-th IEEE Int. Conf. on Software Engineering,* Sorrento, Italy, May 1994.

[13] Collet, C., and Machado, J. (1995). *Optimization of active rules with parallelism.* In: *Proc. 1-st Int. Workshop on Active and Real-Time Database Systems,* pp. 82–103, Skövde, Sweden, June 1995.

[14] Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ladin, R., McCarthy, D., Rosenthal, A., and Sarin, S. (1988). *The HiPAC project: Combining active databases and timing constraints.* ACM SIGMOD Record, 17(3):51–70.

[15] J. Eriksson, J. (1993). *CEDE: Composite Event Detector in an Active Object-Oriented Database.* Master's Thesis, Department of Computer Science, University of Skövde.

[16] Fabret, F., Regnier, M., and Simon, E. (1993). *An adaptive algorithm for incremental evaluation of production rules in databases.* In: *Proc. 19-th Int. Conf. on Very Large Data Bases,* pp. 455–466, Dublin, Ireland, August 1993.

[17] Gatziu, S., and Dittrich, K. R. (1994). *Detecting composite events in an active database systems using Petri nets.* In: *Proc. 4-th Int. Workshop on Research Issues in Data Engineering: Active Database Systems,* pp. 2–9, Houston, Texas, February 1994.

[18] Gatziu, S., Geppert, A., and Dittrich, K. (1991). *Integrating active concepts into an object-oriented database system.* In: *Proc. 3-rd Int. Workshop on Database Programming Languages,* Nafplion, Greece, August 1991.

[19] Gehani, N. H., Jagadish, H. V., and Shmueli, O. (1992). *Composite event specification in active databases: Model & implementation.* In: *Proc. 18-th Int. Conf. on Very Large Data Bases,* pp. 327–338, Barcelona, Spain, August 1992.

[20] Gehani, N. H., and Lieuwen, D. (1997). *Ode triggers: Monitoring the stock market.* Software Practice & Experience, 27(8):905–927.

[21] Geppert, A., and Dittrich, K. (1995). *Specification and implementation of consistency constraints in object-oriented database systems: Applying programming-by-contract.* In: *GI-Fachtagung Datenbanken in Büro, Technik und Wissenschaft (BTW)*, pp. 322–337, Dresden, Germany, March 1995.

[22] Geppert, A., Gatziu, S., and Dittrich, K. (1995). *A designer's benchmark for active database management systems: OO7 meets the beast.* In: *Proc. 2-nd Int. Workshop on Rules in Database Systems,* pp. 309–323, Athens, Greece, September 1995.

[23] Geppert, A., Gatziu, S., Dittrich, K. R., Fritschi, H., and Vaduva, A. (1995). *Architecture and Implementation of the Active Object-oriented Database Management System SAMOS.* Technical Report 95.29, Department of Computer Science, University of Zurich.

[24] Geppert, A., Kradolfer, M., and Tombros, D. (1995). *Realization of cooperative agents using an active object-oriented database management system.* In: *Proc. 2-nd Int. Workshop on Rules in Database Systems,* Athens, Greece, September 1995.

[25] Gray, J., ed., (1993). *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann.

[26] Jagadish, H. V., Lieuwen, D., Rastogi, R., Silberschatz, A., and Sudarshan, S. (1994). *Dali: A high performance main memory storage manager.* In: *Proc. 20-th Int. Conf. on Very Large Data Bases,* pp. 48–59, Santiago, Chile, September 1994.

[27] Jain, R. (1991). *The Art of Computer Systems Performance Analysis. Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley.

[28] Kersten, M. L. (1995). *An active component for a parallel database kernel.* In: *Proc. 2-nd Int. Workshop on Rules In Database Systems,* pp. 277–291, Athens, Greece, September 1995.

[29] Lieuwen, D. F., Gehani, N., and Arlein, R. (1996). *The Ode active database: Trigger semantics and implementation.* In: *Proc. 12-th Int. Conf. on Data Engineering,* pp. 412–420, New Orleans, March 1996.

[30] Oracle Corporation, (1995). *Oracle7 Server: SQL Reference. Release 7.2,* April 1995.

[31] Sybase Inc., Berkeley, CA. (1988). *SYBASE — Data Server.*

[32] Tombros, D., Geppert, A., and Dittrich, K. R. (1996). *Design and implementation of process-oriented environments with brokers and services.* In: *Object-Orientation with Parallelism and Persistence,* B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, eds., Chapter 10, pp. 197–216. Kluwer.

[33] Widom, J. (1994). *Research issues in active database systems: Report from the closing panel at RIDE-ADS '94.* ACM SIGMOD Record, 23(3):41–43.

[34] Widom, J., and Ceri, S., eds. (1996). *Active Database Systems.* Morgan Kaufmann.