An Active Component for a Parallel Database Kernel

M.L. Kersten

CWI, Kruislaan 413, 1098 SJ Amsterdam
{mk@cwi.nl}

Abstract. The Monet parallel database server is an experimentation platform for a variety of datamodels and novel applications. In this paper we describe its active behavior based on the notion of trigger abstractions and an event notification scheme. Trigger abstractions can be used to construct intricate trigger instance networks. The events notified in the DBMS kernel threads flow into a shared event pool. The trigger-event-monitor watches this pool for event combinations to enable trigger firing. We illustrate how these concepts can be used by a rule compiler and describe a performance metric to guide the search for an efficient architectural solution.

1 Introduction

Recent years have shown an increased research interest in active database support [15]. The stream of publications find their origin in the area of rule-based programming [7], i.e. rule processing, and transaction management [14]. The former looks for better ways of rule processing in knowledge intensive applications. The latter aims at simplified transaction management of complex business environments through database triggers [14]. Rules and triggers are the declarative and procedural ends of active behavior which require comparable measures for their implementation.

Generic solutions to both dimensions are sought in the design and implementation of active databases, i.e. a database system that responds to events generated internal or external to the system by activation of a routine. Event-Condition-Action rules form the key procedural concept for specifying active behavior and the approaches taken primarily differ in their choice of abstractions for the components involved. The Event, Condition, and Action components are either explicitly used in the interface language or hidden behind a declarative façade and derived by a rule compiler. A classification of systems against several dimensions is given in [11, 13].

The main contributions of this paper are twofold. First, we present an overview of trigger implementation in Monet, a parallel DBMS kernel under development since 1993 [2]. [1] It complements earlier approaches in aiming for a re-targetable active kernel in a truly parallel setting. We strongly believe that rule-based

[1] The work is in part sponsored by Stichting Informatica Onderzoek Nederland (SION) and European Union (Pythagoras EU 7091).

systems should be built around an optimizing compiler that can exploit the model/language semantics. Such a compiler relies on triggers that can be efficiently implemented within a database kernel. The Monet extension presented is geared towards providing this functionality. It is designed around a minimal set of orthogonal concepts, primarily dealing with event streams, trigger abstractions and their management in a parallel setting.

Active behavior is modelled as procedures, whose bodies are (repeatedly) executed when their event condition holds. A trigger admission policy detects (and rejects) trigger instances whose firing state can not be reached. This leads to early warning of 'useless' active triggers and aids the design of active applications. Furthermore, the DBMS primitives for queue management supports all coupling modes at any level of precision required, such as parallel execution of the actions. Together they provide the building blocks for code generation by a rule compiler.

Second, we demonstrate the performance of our prototype implementation using a benchmark core. This benchmark is our driving force for finding efficient implementation techniques. The results obtained form a reference point for quantitative comparison with other systems.

The remainder of this paper is organized as follows. Section 2 provides an architectural overview of the Monet DB kernel and choices mode for active behavior. Section 3 describes the event and trigger model. Applicability of the model is presented in Section 4 and the performance metric is presented in Section 5. An outlook on future research activities concludes the paper.

2 Architectural Overview.

In section 2.1 we give an overview of the Monet system architecture [2]. [2] The design considerations for inclusion of active behavior are sumarized in Section 2.2. A more detailed description is given in the remainder of this paper.

2.1 Monet Architecture

Monet is a customizable database system developed at CWI and University of Amsterdam, intended to be used as the database back-end for widely varying application domains. It is designed to get maximum database performance out of today's workstations and multiprocessor systems. It has already achieved considerable success in supporting a Data Mining application [8], and work is well under way in a project where it is used in a high-end GIS application. Monet is a type- and algebra-extensible database system and employs shared memory parallelism of SGIs and SUNs. The distributed store version based on [1] is under development and targeted at an IBM SP1 multiprocessor.

The principal assumptions and ideas to achieve Monet's design goals are:

[2] For more details on Monet and related projects, see `http://www.cwi.nl/cwi/projects/monet.html`

– *Use large main memories* Monet makes aggressive use of main memory by assuming that the database hot-set fits into main memory. All its primitive database operations work primarily on main memory structures, no hybrid (memory-disk) algorithms are used. For large data sets it fully exploits the virtual memory manager capabilities of the underlying operating system.
– *Decomposed storage model with deltas.* Monet uses a simple data model based on Binary Association Tables (BATs). This allows for flexible object-representation using the Decomposed Storage Model (DSM)[10]. This vertical decomposition also helps partitioning the database such that the tables fit easier in main memory. Moreover, the BATs come with a *delta* facility, providing access to all elements added (**alpha**) and deleted (**delta**) since transaction begin.
– *Extensible interface.* The Monet Interface Language (MIL) provides for an execution-level binary table algebra and a complete set of imperative programming constructs. Furthermore, the Monet Extension Language (MEL) permits extension of the core functionality through abstract data types and user-defined commands. Such extensions can be dynamically added to a running server.

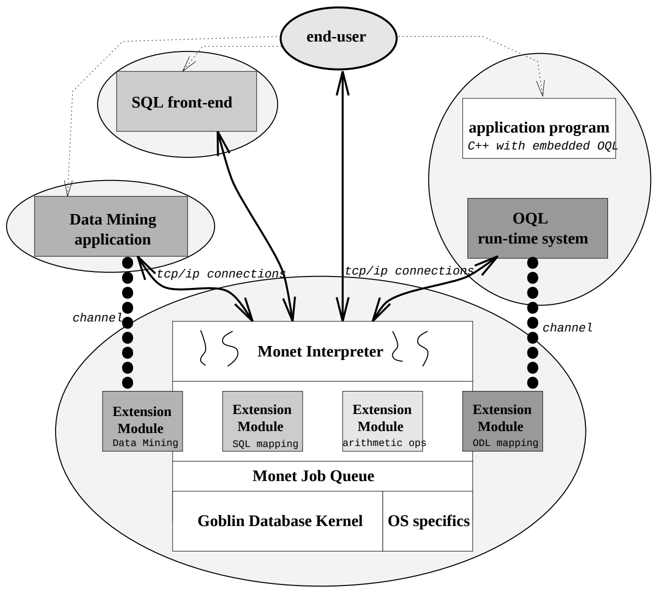

Fig. 1. The Monet Architecture

Figure 1 shows the Monet Interpreter as a multi threaded process, connected to its clients via TCP/IP links. The basic interaction is through the Monet Interface Language (MIL), a simple C-like scripting language. Applications typically accompany themselves by a specific extension module, which provides the extra functionality needed. Extension modules provide operations ranging from arithmetic operations on BATs, geometric library for GIS, to statistical routines for data mining.

2.2 Options for Active Behavior

An active database management system is build around *events*, *event detectors*, a *trigger definition language*, and a *trigger-event-monitor* (TEM). Events are the smallest pieces of information emitted from a system to 'trigger' active behavior upon detection by a trigger-event-monitor. In designing an architecture for a wide spectrum of active models it is mandatory to take the following considerations into account:

– to differentiate enough DBMS kernel events;
– to avoid excessive overhead in their detection;
– to avoid a complex analysis to determine the eligible ECA-rule(s);
– to aim for a simple and open execution model;
– to provide hooks and tools for debugging and performance assessment.

These considerations have been taken into account during system design and experimentation. The baseline has been to consider active behavior a well-identified layer around the physical database and its algebraic engine. This way experimentation with different data and execution models becomes feasible. To set the stage, a short overview of the Monet active behavior is given first.

Event notification Active behavior starts with explicit notification of an event at some point in the system code. Therefore, event detectors are hardwired at specific places in the system kernel, while user-defined events are raised explicitly by user-supplied code. For example, NAOS [3] and SAMOS [5] uses a compiled approach where methods are wrapped by event notification code. This gives the compiler designer and user precise control over the granularity required.

In the Monet architecture there are two obvious places for event notification. Either notification takes place within the database kernel routines or within the MIL interpreter. The former leads to fine-grained event detection with the disadvantage of processing overhead due to generating events under all circumstances. Attachment routines to the storage manipulation operations (e.g. Starburst) only marginally improve the situation, because checking for such attachments consumes recognizable time in a main-memory oriented DBMS.

We have chosen for an flexible approach where event notification is coupled with the operations known by the MIL interpreter. Depending on the mapping from MIL operation to kernel operation this results in fine- or coarse- grain event notification. Furthermore, the extensibility of Monet permits the user to refine any MIL operation to provide a different event notification policy.

Event properties An event comes with properties to identify its environment such that a decision can be made on the old- and new- value of the objects affected or the table with net-effects. In SAMOS the event structure contains the time of the occurrence, the enclosing transaction identifier, and user responsible for its initiation. The event records are represented by persistent objects in the server, which makes them accessible to an object browser.

The Starburst system takes a hybrid approach. Each triggering event places a parameterized procedure call on the prepare-to-commit queue through its attachment mechanism. Moreover, the transaction log keeps track of the net-effect of successive transitions. The log, however, is difficult to analyse with the standard tools offered.

Our approach is to store all event properties within the database. An object identifier is used to re-locate these properties later on. Furthermore, the primitive data structures are set up such that old- and new- values can always be obtained; independent of active behavior. A consequence of our Spartan approach is a potentially higher workload on the kernel, because more query interactions are required to determine the outcome of the condition part. The benefit is a clear distinction between event handling and database activities.

Trigger definitions A trigger defined in a front-end language, such as SQL or rule-language, can be either translated into Monet Interface Language (MIL) constructs or a Monet Extension Language (MEL) module. We have opted for the former, i.e. provision for trigger concept within MIL, because it supports ease of experimentation at a slight overhead in execution speed.

Monet trigger definitions are abstractions, much like procedures. A trigger definition consists of a formal parameter list, an event expression and an action part. The action part is a sequential or parallel MIL statement block. Since MIL is a computationally complete programming language, its provides for a rich environment to construct and experiment with different active applications.

The alternative is to use the Monet Extension Language which supports dynamic linkage of arbitrary C-code with the system kernel. Although extremely powerful, this interface is not meant for casual users without experience in C-programming or limited understanding of the Monet internals. Yet, if need arises, the trigger can be (hand-) compiled to remove interpretation overhead and go for speed.

Trigger execution model Trigger execution encompasses decisions on two issues: identification of fireable triggers and the effect on the process thread raising the event(s). A naive approach is to store the event in the pool and let a separate monitor process inspect the pool repeatedly for eligible combinations. Although this increases parallelism (given multiple CPUs), it also leads to many process switches and dependency on the process priority scheme. Instead, we have chosen for a direct call of the TEM whenever an event is raised. The pool is then inspected and trigger instances are scheduled for execution accordingly. The event is stored in the event_pool otherwise.

The execution model of trigger, i.e. what happens with the thread of control causing the event, differs considerably between systems. Many models differentiate between immediate and deferred execution. In SAMOS the event may cause instances to be executed immediately, effectively suspending the main thread of control. In Starburst the event leads to a delayed procedure call. Once executed at the transaction boundary, they block the main thread of control until active behavior has come to rest[3]. Both solutions are influenced by the system architecture and impose limitations on the active behavior that can be modelled.

We assume that most event expressions are rather simple and that deferred execution semantics can be modelled in the rule language (compiler) using immediate mode of execution and availability of low-level queue management operations. Therefore, all instances raised by an event are scheduled for parallel execution and the main thread of control awaits their termination. Deferred mode can be realized using an event that blocks instances from firing until the transaction boundary is reached. Detached mode is obtained by using the Monet primitives to install (and activate) MIL commands and their dependencies in the request queue explicitly.

In the next section we summarize the events and triggers semantics for Monet. Their formalization is beyond the scope of this paper. The architectural overview of the components involved is shown in Figure 2.

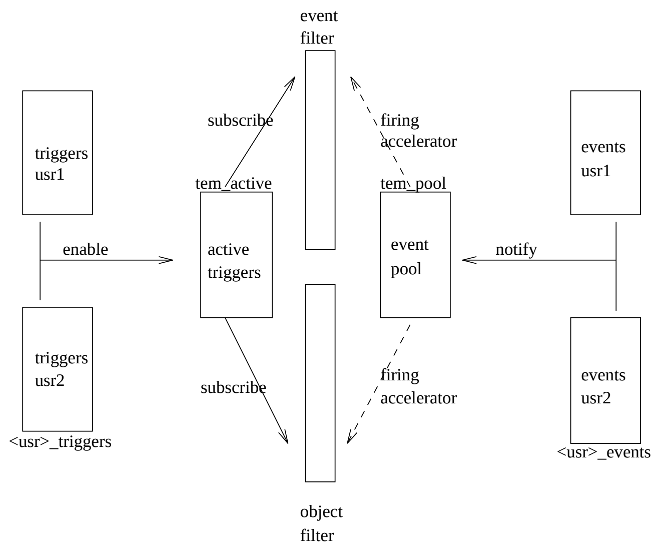

Fig. 2. The active components data structures

[3] attachment procedures can be used to realize immediate actions.

3 Monet Event Model

Events are classified into primitive -, time -, and abstract- events. They are identified by a unique symbolic name and an internal event number administered in a user-readable Monet table.

Primitive events detectors are 'hardwired' into the Monet kernel. In particular, all built-in MIL commands raise an event upon entry and return from their body. Since event detection is the potential source of performance degradation, their event numbers are fixed at compilation time to avoid table lookups. Their symbolic name is a concatenation of the command name and *Entry* or *Exit*. They can be used within the MIL scripts.

Time events are generated within the kernel using the clock interrupt mechanism of the underlying operating system. An event can be raised relative to the current time with a granularity of about 10 ms. Absolute timer events go off at a specific date and time provided the system is running at that moment.

Abstract events are introduced by the user using an **event** <*id*> command; they can be subsequently used like any other event. The abstract event should be uniquely identifiable by its name within the context of the user session. Unlike primitive and time events, an abstract event is explicitly raised using the **notify** command in user code.

An event record carries as little data as possible. It merely designates a state change. The event record contains the internal event number and possibly a single atomic value. For example, upon deletion of an entry from a table C the MIL interpreter executes the command **notify***(C, deleteExit)*. What has been deleted is not part of the event record, but part of the database state itself. The trigger body can use the command C.**delta** to extract this information from the database.

The events of interest are collected in a single global event pool, called *tem_pool*, represented by a user-readable table. An event is added if-and-only-if it passes a hash-based subscription filter, i.e. there is at least on trigger instance with expressed interest. Otherwise the event is considered useless and ignored. The event pool is discarded when the server is stopped.

4 Monet Trigger Model

A trigger is processed in three staged. First, it is defined and administered in a user-specific trigger table. Second, an instantiation is created to watch for the events to occur. Third, the body is scheduled for execution when the events appear in the event pool. These phases are described in more detail below.

Trigger definitions The trigger definition is aligned with the procedural abstraction mechanism of the Monet interface language to provide for templates of active behavior. A trigger definition consists of a header (**trigger** $name(arg....)$), an event expression (**on** $term, ...$) , and an action. The header is a list of formal arguments to specialize event expression and their action. The action is a MIL statement block.

The event expression is a conjunctive boolean expression over event terms. A term obeys the format N or $O.N$ where O denotes a variable and N an event name. Both O and N may be a formal parameter. A term is optionally negated with $\sim$ to require absence of the designated event in the pool. The conjunction and negation operators provides the computational power to model first-order formulas over event pools. Disjunctions merely require expression normalization and replication of the trigger action to obtain a trigger family.

A rationale for this approach is that we expect most triggers to use the objects mentioned in the event expression. Then a disjunctive expression would often imply further analysis within the body to determine what action to be taken. The user then better separates the trigger bodies and encapsulate the common part in a separate routine. Such transformations can be hidden from the user with an ECA or rule compiler.

Trigger definitions are stored in the table *<usr>_triggers*. It is initialized with the triggers defined by the database administrator in the database prelude file.

For example, the trigger below defines a cascading insert from any table S into D. The term $S.insertExit$ becomes true when the command **notify**($insertExit$,S) has been executed by the Monet interpreter. The second term illustrates event negation; it prohibits execution of the trigger body when an error has also occurred. It is raised by the command **notify**($errorExit$). Finally, an instance is created to propagate updates on the employee table to a back-up table.

```
trigger cascade(D, S)
on S.insertExit, ~errorExit
{
  T:= S.alpha();
  if(T.count > 0) D.insert(T);
}
cascade(emp,empBackup);
```

Trigger enabling A trigger becomes enabled by 'calling' it using actual arguments to look after specific event combinations. This 'call' is handled by the Trigger Event Monitor, which enters it into a table of active trigger instances *tem_active*. The instance remains there until it becomes disabled using the **disable** command.

A novelty is to use an admission policy, which determines for each 'call' whether the state of the event pool on which it is to fire can be reached at all. Otherwise, the 'call' is rejected as being not satisfiable. Likewise, a 'call' is refused if it implicitly disables existing triggers. The admission policy routine can be refined by the user.

For example, consider the event expressions "**on** A,∼B" and "**on** B,∼A", i.e. a trigger instance fires whenever the A or B event appears exclusively. Furthermore, assume that the event pool is analysed after each event occurrence. Then extension with the rule "**on** A,B" becomes meaningless, because this state can not be reached.

Trigger firing Each trigger instance behaves like a procedure call (with its own scope of control) whose body is (re-)scheduled for execution when its event expression is satisfied by the event pool. The fireable instances are selected upon arrival of the each event and all their actions are scheduled for execution. This leads to an immediate E-A coupling mode. The user can subsequently switch to decoupling mode explicit scheduling the main part of the action separately. using the MIL request queue management primitives.

In line with all Monet operations, the trigger body emits the signals <trigger>*Entry* and <trigger>*Exit*. The object associated with the event is the first argument of the trigger call. They can be used to serialize execution of different triggers and to differentiate among events.

For example, assume that after the cascade operation a statistics table should also be updated. This scheduling order is achieved by the event term that the trigger instance *cascade* has finished. The parameter D binds with the object of interest. Note that the original events causing *cascade* to fire have already been removed from the pool.

```
trigger statistics(D)
on D.cascadeExit
{
  statcnt.replace(D,D.count);
  statavg.replace(D,D.average);
}
statistics(empBackup);
```

5 Higher Order Semantics.

The trigger mechanism described is the target language for compiling more complex ECA-rules. In this section we illustrate how enriched models can be compiled into these primitives using three prototypical examples: incremental query evaluation for rule processing, history information to control trigger activation, and transaction management.

Incremental conditions Active models permit arbitrary (existential) queries to control their execution where the query has access to a) the current state; b) the current state and transition information; and c) the current state, transition information, and transaction parameters. The Monet active component supports a) only, because limited information is retained about the context in an event record. Therefore, the ECA-rule compiler should generate code to support the other dimensions.

Simple state transition information is already maintained by the underlying BAT implementation. The proposed additions and deletions since the transaction start can be obtained using the commands **alpha** and **delta**, respectively. This feature can be used to maintain a discrimination network [7].

This discrimination network can be produced by an ECA compiler front-end, which produces triggers to collect and propagate information through the network based on the update events. An illustrative and complete algorithm is described in [4], which optimizes incremental query processing by balancing storage and re-construction cost for TREAT and A-TREAT networks.

Here we focus on their mapping to trigger definitions and instances. To illustrate we derive the triggers for the simplified rule A(x,a),B(x,b) $\rightarrow$ C(x,c). This rule requires two inserted-memory nodes (n1, n2) and a single beta-memory node (p1). The former is captured by the *amemory* trigger skeleton below. It reacts to an insertion event. The body requests the elements inserted, selects those of interest (>=C), and inserts the result into N. Similar, *bmemory* reacts to insertions on precisely one operand. It determines the x-values to be propagated to N using a semi-join over the delta of C1. The BAT-loop operation finally updates the container N. Two instantiations are needed to cope with all possible update combinations. Note that the TREAT network components merely require two trigger abstraction, while the actual network can be built out of their instantiation.

```
trigger amemory(C, Container, Value)
on Container.insertExit
{ C.insert(Container.alpha.select(Value));}

trigger bmemory(N,C1,C2)
on C1.insertExit
{
  Z := semijoin(C2,C1.alpha));
  Z @ batloop() { N.insert($1,"c");}
}
```

```
n1 := new(int,str); # create the place holders
n2 := new(int,str);
t1 := amemory(n1, A,"a"); # propagation triggers
t2 := amemory(n2, B,"b");
p1:= new(int,str); # join place holder
bmemory(p1, n1, n2); # propagation triggers
bmemory(p1, n2, n1);
```

Delayed notification The second enrichment considered here is an event history mechanism. The Monet kernel does not maintain an event history, because its semantic is highly dependent on the envisioned application domain. Instead, these policies are better compiled into trigger families using database objects for state administration. In particular, it supports triggers that fire after a specific number of events have been received. The Monet trigger abstraction capturing this semantics is shown below. It counts the events and generates a new event when the high- water mark is reached. The counter is a variable local to the trigger instance and reset immediately.

```
trigger count(E, C, N, EventNew)
on E
{ if (C >= N) { notify(EventNew); C := 0; }
else C := C +1;
}
countExit(errorExit, 0, 3,fatal);
```

Transaction coupling modes Many systems couple trigger activation and their scope of control to the transaction responsible for satisfying their event and query condition. The basis for transaction-based triggers is to support the concept at the user interface, i.e. clients indicate the transaction **begin**, **abort**, and **commit** explicitly. A built-in event makes them visible to the trigger monitor, but it also requires the transactions primitives to cooperate with the TEM. For example, the commit operation in the kernel should wait for the last transaction event to be handled.

The Monet solution is based on two properties. First, postponement of the trigger firing to transaction commit simply requires a test for the *commitEntry* or *commitExit* event to appear in the pool. This effectively means that they occur as terms in (all) the event expressions. Second, the event pool is a user-readable structure and applications can postpone continuation unto a **wait** *event expression* over it becomes satisfied. This way synchronization of parallel actions can be realized.

6 A Performance Metric

The performance of the active component has been measured to isolate the bottlenecks in our architecture as early as possible. Performance depends on the following factors: raw database processing, event detection, event analysis, and trigger instantiation and activation [6]. Although each issue can be analysed in isolation by simulation and analytical modelling, we have implemented a fully functional trigger system and measured the combined effect. This way, we avoid early bias by the perceived performance gains of sophisticated algorithms that do not significantly contribute to the total system responsiveness.

Our experiments have been chosen such that re-implementation on other active database system is feasible. Yet, the implementation makes heavy use of the database structures and kernel operations. A low-level profiler has been used to squeeze the last cpu cycles.

The evaluation platform consists of a Silicon Graphics Indigo 2 workstation with R4400 processor running at 200 Mhz, 1 Mbyte secondary cache and 256 Mb of memory. The performance experiments have been conducted with the Software Testpilot[9], a performance assessment tool developed at CWI.

The Countdown experiment The first experiment determines the baseline for active behavior, namely handling a single (abstract) event and subsequent firing of a single trigger. To deal with the granularity of the system clock, the experiment is modelled as a trigger loop where a variable is decremented until it becomes zero. The condensed Monet code for the abstraction and creation of an instance of 100 cycles is shown below. The loop is started using an explicit notification of the abstract event *down*.

The performance results are shown in Figure 3 with all times measured in milliseconds wall-clock time. The definition and instantiation consumes about 3 ms., which is spent on parsing and compilation to an internal format. The cycle time is largely determined by the cost to schedule the trigger body and its subsequent interpretation. Each cycle takes less than 0.2 ms. Detailed analysis showed that the actual cost to be attributed to active behavior is less than 0.05ms. Initial runs helped us in detecting bad resource management, which lead to non-linear behavior.

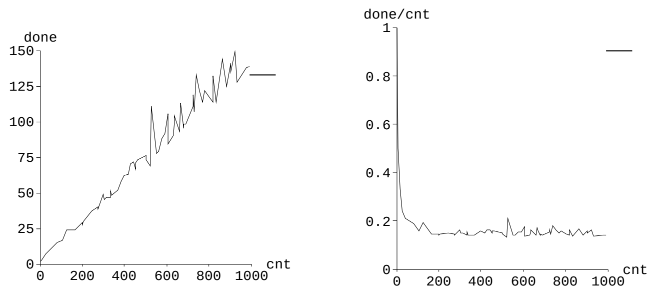

Fig. 3. The Countdown experiment

```
trigger countdown(N)
on down
{   if ( cnt < N ) {
    cnt := cnt+1;
    notify(down);
} }
cnt := 0;
countdown(100);
notify(down);
```

The Dominoes Experiment The second experiment is an analogy of a domino game. The game consists of two phases: setting up the dominoes and pushing the first such that one after the other they fall. The simulated dominoes are trigger instances whose sole action involves raising an event for the next stone.

The purpose of this experiment is to determine whether the implementation can quickly isolate a firable instance. It has been used to assess the effectiveness of the hash-filter against our preliminary linear event expression evaluator.

The Monet code for both phases is shown below. The trigger *dominoes* enables a trigger for each stone which takes a constant time of about 0.4 ms /stone. As shown in Figure 4 the cycle time is also constant (ca. 0.22 ms), because there is exactly one event for each trigger instance. It proved that our hashfilter implementation was effective over the range studied.

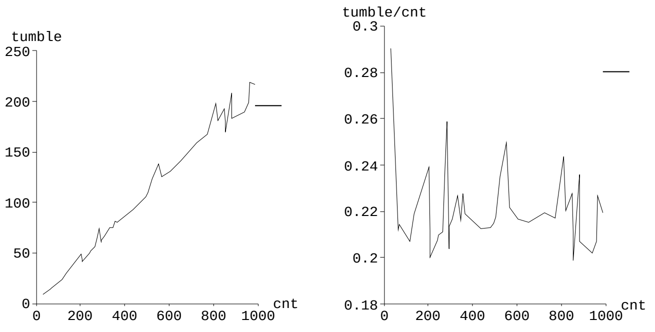

Fig. 4. The Dominoes experiment

```
trigger stone(N)
on N.tumble
{    notify(N+1, tumble); }
trigger dominoes(N)
on setupDomino
{   if ( cnt < N ) {
   cnt := cnt + 1;
   stone(cnt);
   notify(setupDomino);
} }
dominoes(100);
notify(0, tumble);
```

The Pyramid Experiment To increase both the number of trigger instances and the number of events awaiting in the pool for consumption we designed the pyramid experiment. In the construction phase a simple binary tree of trigger instances is constructed, such that each element awaits for a private de-blocking event and an event generated by its parent. The private event is immediately raised, such that after pyramid construction there are as many events in the pool as there are active triggers. The second event encodes the level of the trigger in the pyramid. The destruction phase then merely involves tumbles the root, which subsequently fires all triggers in the next layer. This leads to a quickly increasing workload of trigger actions.

This experiment can be used to demonstrate degradation due to excessive loads on the system kernel and its ability to handle them in parallel. Furthermore, the effectiveness of the accelerator constructs are tested. The Monet trigger abstractions used to build a Pyramid are shown below. The results of this experiment - without optimizations- is given in Figure 5.

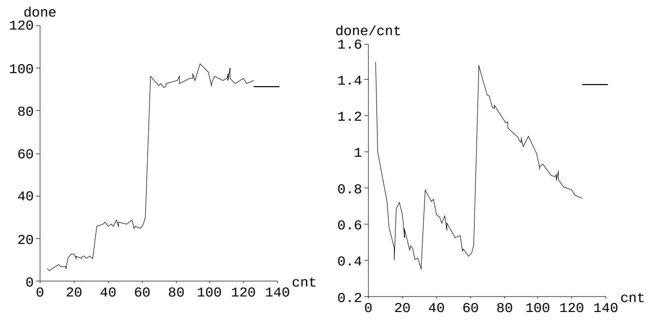

Fig. 5. The Pyramid experiment

```
event destruct, fall;
trigger stone(N,D) on D.fall, N.destruct{
  notify(D/2,destruct);
}
proc pyramid;
proc pyramid(N,D) := {
  stone(N,D);
  if( D> 0) {
   pyramid(N+D,D/2);
   pyramid(N-D,D/2);
   notify(D, destruct);
  }
}
pyramid(4,2);
notify(4,fall);
```

7 Summary

We have described the core-implementation for active behavior in Monet, a parallel DBMS kernel. The key concept exploited is to align active behavior with procedural abstraction, i.e. trigger abstractions. Trigger enabling then aligns with 'calling' the abstraction. Thereafter the trigger action becomes scheduled for (parallel) execution whenever the event expression can be satisfied. This conceptual coupling of trigger instances with procedure bodies and its 'indirect invocation' through events proved a simple and effective means to model a wide-range of examples.

A metric has been defined to test progress of the implementation and to provide a guidance to the ECA compiler writers. The performance figures show that Monet can efficiently support coarse grain active behavior. We intend to further study the impact of parallel triggers. In particular would we like to know the granularity of the trigger actions to exploit the parallelism offered by the kernel implementation.

References

1. C.A. van den Berg and M.L. Kersten. *An analysis of a dynamic query optimisation scheme for different data distributions.* In J.Freytag, D. Maier, and G.Vossen, editors, *Advances in Query Processing*, pp. 449-470. Morgan-Kaufmann, 1994.
2. P. Boncz, M.L Kersten *A Impressionist Sketch of the Monet Database System* Basque workshop in Database Systems, June 1995.
3. C. Collet, T. Coupaye, T. Svensen, *NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System*, Proc. 20th Int. Conf. on Very Large Databases, Chile, September 1994.
4. F. Fabret, M. Regnier, E. Simon, *An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases*, Proc. VLDB 19, Dublin, Ireland, Aug. 1993, pp.455-466.
5. S. Gatziu, K.R. Dittrich, *Detecting Composite Events in an Active Database System Using Petri Nets*, Proc. Workshop on Rules in Database Systems, Edinburgh, UK, September 1993, Springer-Verlag.
6. A. Geppert, S. Gatziu, and K.R. Dittrich, *Performance Evaluation of an Active Database Management System: OO7 Meets the BEAST*, Institut für Informatik der Universität Zürich, Nov. 1994
7. E. Hanson, *Rule Condition Testing and Action Execution in Ariel*, Proc. ACM SIGMOD, San Diego, June 1992, pp. 49-58.
8. M. Holsheimer, M.L. Kersten, A. Siebes, *Data Surveyor: Searching the Nuggets in Parallel*, Chapter 4 in Knowledge Discovery in Databases 2 editor Piatetsky-Shapiro and Frawley, MIT Press, Menlo Park, California,1995.
9. M.L. Kersten, F. Kwakkel, *Design and Implementation of a DBMS Performance Assessment Tool* Proceedings DEXA'93, Praag, Sept 1993, pp. 265-276.
10. S. Khoshafian, G. Copeland, T. Jadodits, H. Boral, P. Valduriez (1987) *A Query Processing Strategy for the Decomposed Storage Model,* In Proceedings of the IEEE Data Engineering Conference, pages 636-643.
11. N.W. Paton, O. Diaz, M.H. Willims, J. Campin, A. Din and A. Jaime, *Dimensions of Active behaviour*, Proc. Workshop on Rules in Database Systems, Edinburgh, UK, September 1993, Springer-Verlag.
12. A. Shatdal, C. Kant, J.N. Naughton, *Cache Conscious Algorithms for Relational Query Processing*, Proc. 20th VLDB, Santiago, Chili, 1994, pp. 510-521.
13. J. Widom, *Deductive and Active Databases: Two paradigms or Ends of a Spectrum*, Proc. Workshop on Rules in Database Systems, Edingburgh, UK, September 1993, Springer-Verlag.
14. J. Widom, R,J, Cochrance, B.G. Lindsay, *Implementing Set-Oriented Production Rules as an Extension to Starburst*, Proc. 17th Int Conf. on Very Large Databases, Barcelona, Spain, September 1991.
15. J. Widom, S.Chakravarthy, *Research issues in data Engineering*, proc. RIDE'94, Houston, Texas, Feb 14-15, 1994.

This article was processed using the LaTeX macro package with LLNCS style