Model analysis for business event processing

L. Zeng
H. Lei
T. Koyanagi
H. Ohsaki
H. Chang

Business event processing requires efficiently processing live events, computing business performance metrics, detecting business situations, and providing real-time visibility of key performance indicators. Given the high volume of events and significant complexity of computation, system performance—event throughput—is critical. In this paper, we advocate model-analysis techniques to improve event throughput. In the build time, a series of model analyses of the application logic are conducted to understand such factors as runtime data-access path, data flow, and control flow. Such analyses can be used to improve throughput three ways: at build time it can be used to facilitate the generation of customized code to optimize I/O and CPU usage; information about the control flow and data flow can be used to ensure that CPU resources are used effectively by distributing event-processing computation logic evenly over time; and at runtime, knowledge gained from the model can be used to plan multithreaded parallel event-processing execution to reduce wait states by maximizing parallelization and reducing the planning overhead. This paper presents a series of model-analysis techniques and the results of experiments that demonstrate their effectiveness.

INTRODUCTION

For an organization to function effectively in today's environment and to stay competitive and profitable, business activities and operation performance must be continuously visible. Business event processing[1–5] enables processing continuous live events, computing metric values, and detecting situations in real time, thereby supporting applications such as program trading, fraud management, and location-based services. It represents a new generation of enterprise data management and is gaining considerable momentum in both academia and industry.

A user-friendly language is needed to support business event processing. In our design, we use the popular ECA (event condition action) rule-based programming mode with substantial extensions to support the computation of the active metric network. The rule-based programming model allows application developers to realize business rules in event-processing applications, thus freeing them from transforming declarative logic into details of procedural logic.

Improving event-processing throughput becomes imperative, as there are emerging business needs that require huge amounts of event data to be processed on a just-in-time basis. One way to improve throughput is to reduce the I/O and CPU cost of individual event processing at runtime in order to reduce processing time for each event. In this paper, we propose a model-analysis approach to generate executable code that processes events with I/O and CPU efficiency. For example, by analyzing the model, knowledge about the runtime object access paths can be gained and used to generate a customized cache component to minimize the I/O in the runtime.

Another approach to improve throughput is to enable concurrent event processing. Actually, conventional event-processing systems adopt a multithread approach to process events simultaneously (one thread per event), where concurrent control is based on an object-locking mechanism. Such an approach is intuitive. However, it has some shortcomings. For example, during the lifespan of an event-processing thread, a collection of objects (e.g., metrics) may be updated. Accordingly, a collection of locks are placed on these objects, and lock contentions can occur among those concurrent threads if they update the same objects. These lock contentions may force some threads to be idle. For example, when two concurrent threads update the same set of metrics, the execution of these two threads needs to be serialized. The threads in idle status need to release system resources and then reacquire system resources whenever they are activated. Such a situation could happen several times during the lifespan of a thread. Such thread context switching introduces extra system overhead, slowing the event processing.

Multithread parallel processing is necessary to improve system throughput. The key is how to minimize the overhead introduced by concurrent control. We advocate model-analysis techniques to plan concurrent wait-free threads, which can eliminate overhead on concurrent control and thread context switching. By analyzing the model, the knowledge about constraints among the data flow and control flow in the model can be obtained. With this knowledge, wait-free, concurrent event-processing threads can be planned by selecting the appropriate work items to be processed. To summarize, the model-analysis techniques that improve event-processing performance are the following:

Customized code generation—We implemented a model compiler that generates executable code for the event processing logic (ECA rules), wherein a Java** Virtual Machine (JVM**) is adopted as the execution platform. This side-stepped the need to develop a home-grown evaluation engine. Also, it exploits the performance improvement features in the modern JVM, in particular, the just-in-time compiler that optimizes code at runtime based on execution patterns. Further, all the executable code is generated specifically for the given application logic; code for dealing with the general case is excluded. Therefore, the executable code not only has a small footprint, but also runs more efficiently than the code for the general case. Finally, the cache module can be generated based on the runtime object access paths, which can greatly reduce the I/O access in the runtime. Therefore, in runtime, the system benefits from the generated code in reducing both I/O-access and CPU-cycle consumption.

Model-driven multimediators—A state chart can be constructed from the collection of event-processing rules used to process a business-event type. Instead of compiling each state chart into an independent Java class, we compile each rule into an independent Java class. We design multimediators to orchestrate the generated Java classes. Each mediator processes both a queue that buffers the available work items and a collection of threads to process the work items. These mediators construct a queuing network wherein the topology construction is driven by the model. The number of threads in each mediator can be adjusted to tune the performance. By doing this, CPU resources can be allocated and reallocated to avoid the occurrence of bottlenecks in the queuing network, thereby dynamically improving throughput.

Scheduling of model-driven event processing—For each mediator, a thread pool is used to enable the parallel processing of events. However, when there is a data-flow dependence among the concurrent threads in different mediators, parallelization is not necessarily able to improve the throughput. For example, for a mediator that executes metric computation and situation detection, a deadlock may occur among concurrent threads if there are read and write conflicts on a collection of metrics

and situations. In such a case, the parallel threads cannot make any progress on event processing. When a high volume of events need to be processed (e.g., a few hundred events per second), the number of waiting work items is always much larger than that of threads supported by the system. Motivated by selecting work items that enable wait-free concurrent threads, a model-driven scheduler is proposed for the mediators. In the build time, it analyzes the possible runtime data flow among the event-processing threads to determine whether any two types of work items may cause lock contentions. Thus, by using this precomputed knowledge, the scheduler can select work items for wait-free concurrent threads with little runtime overhead. Such an approach greatly improves event throughput.

SYSTEM DESIGN RATIONALE

In this section, we first briefly describe the programming model for business event processing and then present the overall system design rationales. We adopt a rule-based programming model, `Event(eventPattern)[condition]|expression`, wherein ECA rules are used to describe the event-processing logic.

In an event-processing ECA rule, the event pattern can be an occurrence of business events, a change in a metric value, or the occurrence of a situation. The condition is a Boolean expression specifying the circumstances to trigger the action. It examines the properties of events and states (i.e., metrics and situations). The action is an expression that computes or updates the states. In order to support the sliding windows (for example, the last 100 tuples), the timer and counter are considered as metrics. For each type of business event, application developers can create a collection of rules to represent the event-processing logic. The metric and situation values are contained by context instances, which can form parent-child relations. Event-processing results (i.e., context instances) need to be persisted. The programming model is described in more detail in Reference 3.

Given a suitable programming model, it is critical that application logic can be efficiently executed. The first design choice concerned the runtime data store used to persist event processing states. In our initial design,[3] we generated Structured Query Language (SQL) statements to execute the computation logic in rules. By using the SQL query engine to perform the metric computation, situation detection, and state persistence, the event-processing engine became a lightweight component. However, most modern SQL query engines are designed for optimizing queries on the data in persistent storage, where rules are executed after the event data is saved to disk. Therefore, such an approach is I/O intensive, as every SQL statement requires I/O access to the data in the persistent data store. Such a design requires a high-performance I/O hardware platform. By observing this limitation, we chose to execute the computation logic in main memory first and then save the computation results to the data store. Further, to reduce the I/O demand in runtime, some cache components were generated based on object access paths.

The second design choice was selecting the appropriate rule-execution framework. There are two approaches to executing event-processing rules: interpreting and compiling. Both approaches have advantages and drawbacks. We considered the interpreting approach first. The advantage of interpreting is that it has more controls on execution progress because the interpreter maintains all the model information. This facilitates more fine-grained multithreaded scheduling that uses model information to achieve better system resource utilization. However, when executing ECA rules, the interpreter consumes more CPU cycles than directly compiled code, given that there are many types of operators, such as relational, set, vector, and scalar, used to construct expressions.[6] Further, the metrics referenced in expressions are not limited to the same context instance. To locate the associated instances of a metric, the interpreter needs to navigate through the hierarchy of context instances at runtime, which may incur performance penalties.

An executable code is generated by adopting a compiling approach for the set of rules that are associated with one type of business event. The collection of rules associated with one type of event compose a state chart; thus, the executable code essentially implements a state chart. As customized code can be generated for the execution of the state chart, it can reduce CPU cycle demand. However, the event-processing logic is embedded into generated code at compilation time. This implicates a potential performance issue. When adopting a multithread approach to process events without

model information, the thread scheduling needs to rely on the locking-based scheduling features provided by either the operating system or the programming language (e.g., JVM). The locking-based scheduling usually results in high system overhead,[7] especially in multiple-CPU systems.

To take advantage of both approaches, we propose a hybrid approach. At build time, the triggering relation among the rules that are associated with one type of business event is transformed into a state chart. By limiting the search space, the state chart expedites locating the rules to be triggered at runtime. Each rule is compiled into an independent executable code. When a business event occurs in runtime, the state chart is interpreted in order to orchestrate the executable code that implements the associated rules. The advantages of such a compilation-interpreting approach are twofold. On the one hand, by interpreting, the model information can be used to optimize the resource allocation when executing the rules. For example, information about the data flow among the rules can be used to plan the wait-free execution of the rules. On the other hand, execution of an individual rule is done by executing precompiled code, which offers the efficiency of the compilation approach.

MODEL TRANSFORMATION

To facilitate interpretation and compilation, two types of model transformation are conducted: control-flow and data-flow analysis, and expression preprocessing.

Control-flow and data-flow analysis

The rule-based programming model allows application developers to focus on the business logic of event processing, which frees them from understanding extraneous details of control flow and data flow. However, when model checking is performed, for example loop detection, explicit control-flow and data-flow information is required. In this subsection, we present how to extract explicit control-flow and data-flow information from rules.

We discuss the control-flow extraction first. As each type of incoming business event triggers a consequence of rules to be executed, each event can, therefore, initiate a control flow (represented as a state chart) constructed by the rules. In the state chart, states are events, metrics, or situations, while each transition is associated with a rule that is triggered by the state. The transition starts from and updates the value of a metric or situation that the transition points to. If there is a loop in any control-flow state chart, the system notifies the model designer to modify the rules. The three state charts shown in ***Figure 1*** can be generated, given the following set of rules:

$$
\begin{aligned}
&R_1 : Event(\gamma_1)[\gamma_1.a_1 > 5] | \zeta_1 := \chi_1(\gamma_1)\\
&R_2 : Event(\gamma_1)[\gamma_1.a_1 > 5] | \zeta_3 := \chi_2(\gamma_1)\\
&R_3 : Event(ValueChange(\zeta_1))[\zeta_1 > 5] | \zeta_3 := \chi_3(\zeta_1, \zeta_2)\\
&R_4 : Event(ValueChange(\zeta_3))[\zeta_3 < 15] | \zeta_5 := \chi_4(\zeta_3)\\
&R_5 : Event(ValueChange(\zeta_3))[\zeta_3 < 15] | \zeta_6 := \chi_5(\zeta_3)\\
&R_6 : Event(\gamma_2)[\gamma_2.a_4 < 90] | \zeta_6 := \chi_6(\zeta_6, \zeta_2)\\
&R_7 : Event(\gamma_2)[\gamma_2.a_8 > 25 \wedge \zeta_1 > 5] | \zeta_1 := \chi_7(\gamma_2, \zeta_3)\\
&R_8 : Event(ValueChange(\zeta_4))[\zeta_4.a_5 > 100] | \zeta_3 := \chi_8(\zeta_4, \zeta_5)\\
&R_9 : Event(ValueChange(\zeta_6))[\zeta_6 > 5] | \zeta_3 := \chi_9(\zeta_6, \zeta_7)\\
&R_{10} : Event(ValueChange(\zeta_7))[\zeta_7 > 15] | \zeta_5 := \chi_{10}(\zeta_6)\\
&R_{11} : Event(ValueChange(\zeta_2))[\zeta_2 < 25] | \zeta_3 := \chi_{11}(\zeta_2, \zeta_5)\\
&R_{12} : Event(\gamma_3)[\gamma_3.a_5 > 25] | \zeta_1 := \chi_{12}(\gamma_3, \zeta_1)
\end{aligned}
$$

Based on the generated control-flow state charts (see example in Figure 1), we can generate data-flow transitions. In the control flow, if the context element (i.e., event, metric, or situation) represented in state α is an operand to compute the metric or situation indicated in state β, then there is a data-flow transition from state α to β. If there are multiple operands, then a join state is added. In control-flow state charts, there are two types of data-flow transitions: a data-flow transition that is inside a control-flow state chart and a data-flow transition that is between the control-flow state charts. If data-flow transitions construct any loops, then the system must notify the model designer to modify the rules, because the presence of loops indicates that when some rules are executed, they may always be terminated, as some operands are not available.

Expression preprocessing

Uniformly, the expressions in rules can be denoted as

$$
\begin{aligned}
f = \chi(&c_1.\zeta_1, c_2.\zeta_2, \ldots, c_n.\zeta_n, c_1.\varsigma_1, c_2.\varsigma_2, \ldots,\\
&c_m.\varsigma_m, c_1.\gamma_1, c_2.\gamma_2, \ldots, c_l.\gamma_l),
\end{aligned}
\tag{1}
$$

where χ is the operator, and there are three kinds of operands: metrics (ζ_i), situations (ς_i), and events (γ_i). If the output of an expression is a metric, f represents $c.\zeta$ and c is the context that ζ belongs to. In order to facilitate the evaluation of expressions, preprocessing is performed by the model transformer at build time. At runtime, the created context instances form a tree structure based on the parent-child relationship. In an expression, the output and

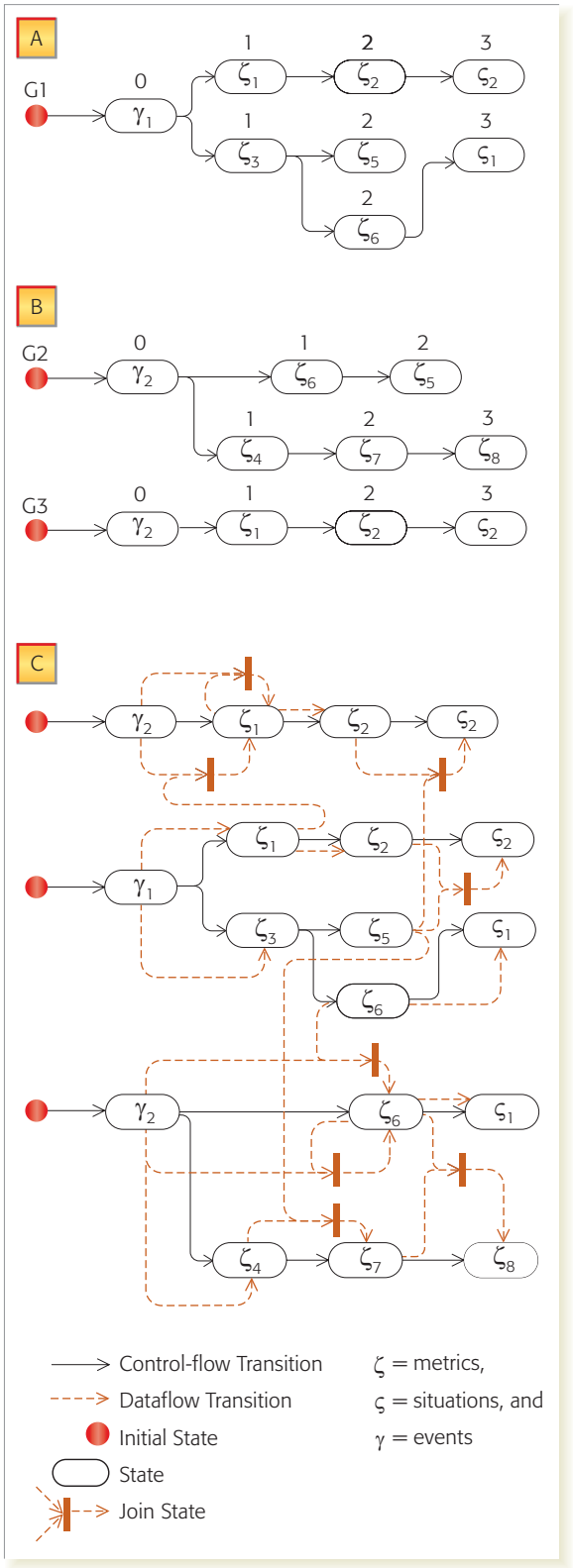

Figure 1
State charts of control flow (A and B) and of data flow (C)

operands (e.g., metrics) may belong to different context instances, wherein a path exists between the output and each operand. Based on the original context instance c that the expression belongs to and destination context instance c_i that the operand or output belongs to, the navigation path can be computed as

$$< /\mu_1/\mu_2/ \ldots\ldots /\mu_k >,$$

where μ_i in the path represents a step. Referring to the tree structure of context instances, there are two possible directions for a step: from a child-to-parent context instance, where μ_i is denoted as "..", or from a parent-to-child context instance, where μ_i is denoted as $C_i(p)$. The context type is denoted as C_i, and p is an association predicate on any metric ζ_k in context C_i that is used to identify which context instances are in the path. An example of p can be $\zeta_2 = 5 \wedge \zeta_4 = 4$. When p is null, all context instances of context type C_i are matched. To accelerate the searching of context instances, some cache components are generated based on the association predicate p. Details about cache component generation are given in the next section.

CUSTOMIZED CODE GENERATION

Generating Java code for expression evaluation consists of two steps:

1. Generating code to retrieve the value of each operand
2. Generating code to execute the operator

In the first step, the model compiler generates code to retrieve the operand value in the cache. Here, we use the metric attribute as an example of operands to illustrate our code-generation solution. By specifying attributes of the metric and the context navigation path, the metric operand in expression (1) can be further refined as

$$\zeta_i < /\mu_1/\mu_2/ \ldots\ldots /\mu_k > .a_i[d_1, d_2, \ldots, d_l].$$

In this step, the code generation for accessing the operand consists of two phases: generating code to retrieve the context instance object and generating code to retrieve the content of the metric attribute. We first discuss how to generate code to retrieve the context instance that the metric ζ_i belongs to.

In our design, a cache component is adopted to accelerate the retrieval of context instance objects. The cache manager provides an interface

Table 1 Code generated to retrieve the context instance

Code Generation for Nongroup Operators	
Step Type	**Generated Code**
Child step, *p* is null	`ContextInstanceC contextInstanceC = CacheManager.getContextCache(c).getContextInstanceByID(iID);`
Parent step	`getParentContext()`
Child step	`ContextInstanceCi contextInstanceCi = contextInstanceC.getChildContexInstanceByKey(ci,k1, k2, ..., kn0);`
Code Generation for Group Operators	
Step Type	**Generated Code**
Child step, *p* is null	`ContextInstanceCi contextInstanceCis = contextInstanceC.getChildContextInstanceByType(ci);`
Child step, *p* is not null	`ContextInstanceCi contextInstanceCis = contextInstanceC.getChildContextInstanceByKey(ci,k1, k2, ..., kn);`
Step after child step	`for (ContextInstanceCi contextInstanceCi: contextInstanceCis) {...}`

```
(ContextInstance
getContextByInstanceID(Double iID))
```

that enables retrieval of context instance objects by context instance id (iID). The cache manager possesses a table of context types that is generated from the observation model. Each context type entry is associated with a hash table that caches the context instances of the particular context type. To implement the interface, the hash table uses the iID as the key and the context instance object as the value. Further, to expedite the evaluation of the association predicate and avoid a full table scan, the context manager maintains a collection of indexes based on equality queries and range queries in the association predicate. In our implementation, the index manager creates two types of indexes, namely hash table indexes created for equality queries and B+ tree indexes created for range queries.

The generated code for retrieving the context instance is shown in ***Table 1***, where we assume that the expression execution is triggered in the context instance and that the context type is *C* and the instance ID is iID. In the code generation, there are three cases, self step, parent step, and child step. In the child step, the predicate *p* cannot be null and should not contain any range queries, as the target is one context instance.

In the above method, `getContextInstanceByKey()` is implemented based on the hash-based index that was created by the index manager. By applying the above two cases on steps in the context path, we can obtain the target context instance `contextInstanceCn`. As we generate the getter and setter methods for each metric, we can use the getter `contextInstanceCn.getMetricMi()` to retrieve the metric value.

In the case where χ is a group operator, the target context instances can be multiple. For the parent step, the code generation is the same as for nongroup operators. However, in the case of the child step, instead of a single context instance, an array of context instances are retrieved. Here, we differentiate two cases: either *p* is null or not null. Further, once there is a child step, the next retrieval step needs to be done in the loop. At the last step, all the operand metrics are added into a collection as the operand for group operators.

MULTIMEDIATOR ORCHESTRATION

A queuing network that consists of a multimediator is proposed to allocate CPU resources by dynamically sizing the thread pool to optimize event throughput. Each mediator in the network processes a work item queue, an interpreter, and a thread pool. The queue buffers the available work items.

The interpreter identifies related executable code according to the stage of event-processing logic. The thread pool enables multithread concurrent processing on work items. The threads in different thread pools have some level of priority. The CPU resource allocation of a mediator is determined by the size of its thread pool. By dynamically configuring the size of the thread pool, CPU resources can be dynamically allocated.

An approach to constructing the topology of a queuing network is to have each mediator execute only one type of rule in control-flow state charts. However, in such an approach, the number of mediators can be much larger than the number of threads that the system can run concurrently. Therefore, some mediators are idle and waiting for a CPU cycle to be allocated. Also, some system overhead is introduced when a large number of queues is managed. For these reasons, such a design may degrade the event throughput.

In order to maximize the thread parallelization of computation and, at the same time, avoid thread suspension, we propose a solution that distributes the rules into a collection of mediators, wherein the number of mediators is determined by the topology of state charts. Our strategy is twofold. First, the order of rule execution is preserved by network topologies constructed by the mediators. This can be done in two steps: sorting the rules based on the execution sequence of each control-flow state chart and distributing rules to a collection of ordered mediators based on their place in the execution order. The second part of the strategy minimizes the communication cost among the mediators and eliminates the data-flow conflicts among the threads in different mediators. This can be done by taking rules that are associated with an interstate-chart data-flow transition and distributing them into the same mediator. Therefore, by planning the execution order of the rules inside a mediator, wait-free threads can be enabled (details about the execution planning are given in the next section). The detailed procedure of topology construction is as follows:

Step 1—First, in generated control-flow state charts, the states are marked by using the control-flow distance from the event states. If there are multiple paths from an event state, then the maximum distance is used. For example, in state chart G1 (Figure 1A) the state γ_1 is marked as 0, and the state ζ_1 is marked as 1.

Step 2—The states in state charts are aggregated based on their distance from the event state. For state chart G*i*, the states with same distance mark are put into a sequence of ordered baskets $B_{i,j}$, ordered by *j*. The order of baskets is used to reserve the execution order of the rules that were triggered by the elements in the baskets. For example, for state chart G1 (Figure 1A), the ordered baskets are generated as

$$\mathbf{B1}[B_{1,0}\{\gamma_1\}, B_{1,1}\{\zeta_1, \zeta_3\}, B_{1,2}\{\zeta_2, \zeta_5, \zeta_6\}, B_{1,3}\{\varsigma_1, \varsigma_2\}].$$

This indicates, for example, that the rule triggered by γ_1 is always executed before the rules triggered by ζ_1 and ζ_3. For another example, state chart G2 (Figure 1B) generates the order of baskets as

$$\mathbf{B2}[B_{2,0}\{\gamma_2\}, B_{2,1}\{\zeta_4, \zeta_6\}, B_{2,2}\{\zeta_7, \varsigma_1\}, B_{2,3}\{\zeta_8\}].$$

Step 3—In the above steps, each control-flow state chart generates an order of baskets. In this step, the baskets that belong to different state charts are merged and binary sorted.

We assume that **B1** has the biggest number of baskets among all the **B*i***. The **B*i*** ($i \geq 2$) are merged to B_1 one by one. For each $B_{i,y}$ in **B*i***, starting from $y = 0$, where *e* is an element in $B_{i,y}$, *e* can be merged to a basket in **B1** according to the following cases:

Case 1: There exists either one or more than one basket in **B1** that has a data-flow relationship with the elements in $B_{i,y}$. If there exists one basket $B_{1,x}$ in **B1** that contains elements which trigger the rule and if the output of the rule provides operands for the rules triggered by element *e*, then *e* is merged into $B_{1,x}$. For example, in **B1**, there is a basket $B_{1,1}$. The elements ζ_3 in $B_{1,1}$ trigger rule R_5, and the output of R_5 is ζ_6, which is an operand for rule R_6 that is triggered by γ_2. Therefore, ζ_3 is merged into $B_{1,1}$, and $B_{1,1}$ becomes $\{\zeta_1,\zeta_3,\gamma_2\}$. After this merge, **B1** becomes

$$\mathbf{B1}[B_{1,0}\{\gamma_1\}, B_{1,1}\{\zeta_1, \zeta_3, \gamma_2\}, B_{1,2}\{\zeta_2, \zeta_5, \zeta_6\}, B_{1,3}\{\varsigma_1, \varsigma_2\}].$$

If there is more than one basket in **B1** that has a data-flow transition with elements in $B_{i,y}$, then the baskets in **B1**, the baskets in between, and the element *e* are merged into a single basket. Such a merge method guarantees that all the rules associ-

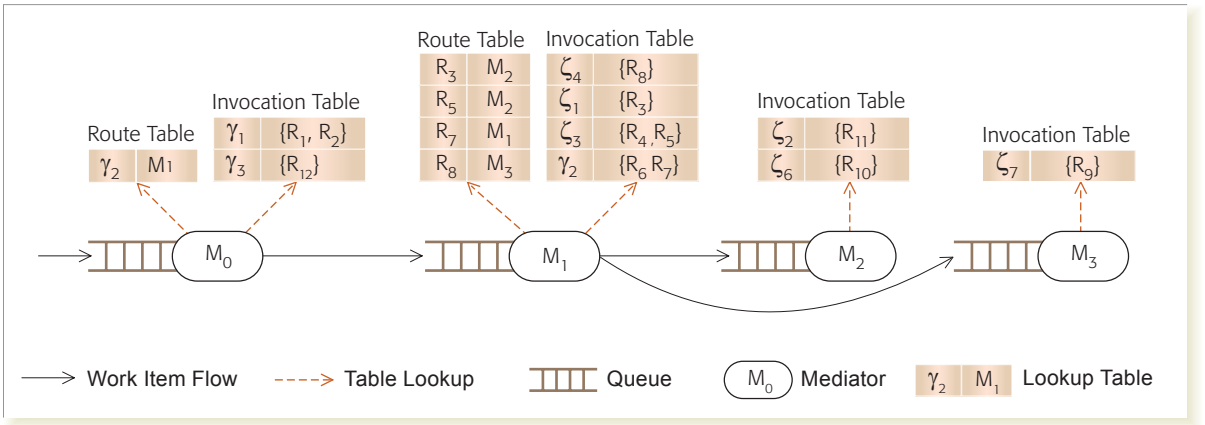

Figure 2
An example of a queuing network

ated with an interstate-chart data flow are distributed into the same basket. Thus, there is no interstate-chart data-flow transition among the mediators. In this way, the communication cost among the mediators is reduced.

Case 2: There is one basket $B_{1,x}$ in **B1** that contains element *e*. Element *e* is merged into $B_{1,x}$ in this case. For example, ζ_6 ($\zeta_6 \in B_{1,2}$) is merged into $B_{1,2}$, and $B_{1,2}$ becomes $\{\zeta_2,\zeta_4,\zeta_5,\zeta_6\}$. By performing this merge, the rules that update the same metrics or situations are distributed into the same basket.

Case 3: There is no basket $B_{1,x}$ in **B1** that has a data-flow transition with *e* or contains *e*. If $j = 0$, then basket $B\{e\}$ is inserted into **B1** as $B_{1,0}$; if $j \neq 0$, then *e* is merged into $B_{1,x}$, where $B_{1,x}$ is the next basket to which elements in $B_{i,y-1}$ are merged or the basket to which other elements in $B_{i,y}$ are merged. If the elements in $B_{i,y-1}$ or other elements in basket $B_{i,y}$ are merged to the last basket of **B1**, then $B\{e\}$ is inserted into **B1** as the last basket. For example, the ζ_8 in basket $B_{2,3}$ is inserted into **B1** as $B_{1,4}$, because $B_{1,3}$ is the last basket to which ς_1 (an element in the same basket as ζ_8) is merged. Such a merge operation preserves the invocation order of rules in all the control-flow state charts. After **B1** and **B*i*** are merged, if the number of baskets in **B1** becomes smaller, it swaps with the basket **B*i*** that currently has the largest number of baskets.

Step 4—After the above steps, all baskets are merged into a sequence of baskets. Using the above example, the baskets of three state charts are merged into

$$\mathbf{B}[B_{1,0}\{\gamma_1,\gamma_3\}, B_{1,1}\{\zeta_1,\zeta_3,\gamma_2\}, B_{1,2}\{\zeta_2,\zeta_4,\zeta_5,\zeta_6\}, B_{1,3}\{\varsigma_1,\varsigma_2,\zeta_7\}, B_{1,4}\{\zeta_8\}].$$

Figure 2 shows an example of a queuing network. Such a network is constructed based on these merged baskets, one mediator per basket, except for the last basket. Each mediator possesses an invocation table. The table maps the trigger elements to the rules being triggered. For example, an entry in the invocation table for M_0 is $<\gamma_1, \{R_1, R_2\}>$, which indicates that when M_0 receives business event γ_1, rule R_1 and rule R_2 will be executed. The mediator also possesses a route table if it has more than one outgoing arrow or it will have to forward work items to other mediators. The route table defines two types of mapping: mapping between the rules and the next mediator to which a work item will be dispatched when the execution of the rule is completed and mapping between a work item and the mediator to which that work item should be forwarded. For example, an entry in the route table of M_0 is $<\gamma_2, M_1>$, which indicates that business event γ_2 is bypassed by M_0 and forwarded to M_1.

DATA-FLOW-DRIVEN EVENT-PROCESSING SCHEDULING

In this section, we present a novel data-flow-driven scheduler for mediators. The scheduler enables wait-free concurrent threads by planning the execution order of work items. There are two issues in realizing such an approach. The first issue is to determine whether there are enough available work items to be planned for execution. In most applica-

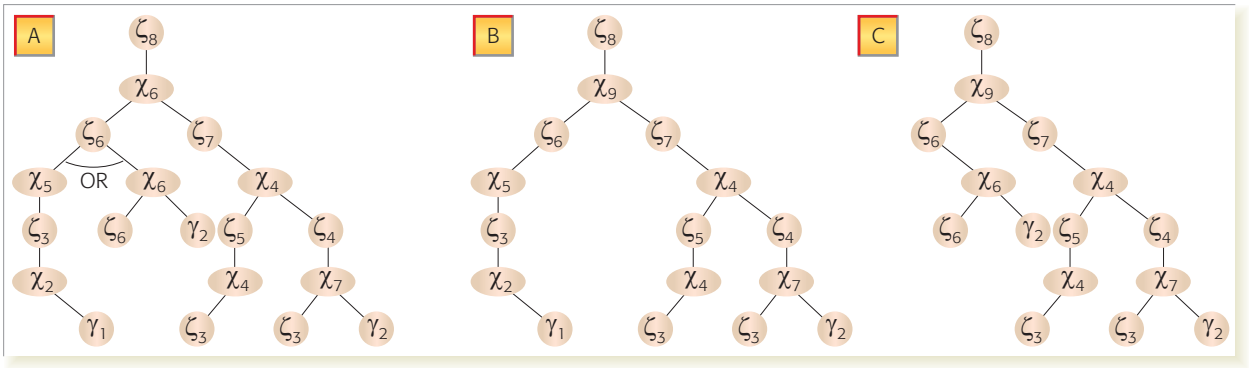

Figure 3
Dependence trees: (A) with OR, (B) without OR, and (C) without OR

tions, the number of work items buffered is much larger than the number of concurrent threads that the system can support. Such a design principle complies with the ratio between the size of main memory and the number of CPUs (or CPU cores) in the model computer architecture. The second issue is the runtime overhead introduced by the scheduler. This issue is resolved by performing a data-flow analysis in build time. In our approach, the possible runtime data flow among the rules is analyzed in build time to determine whether any two types of work items may cause lock contentions. Based on the data flow and lock contentions, an execution policy is created for each rule. Using this precomputed knowledge, the scheduler can decide when work items are ready for execution in wait-free concurrent threads with little runtime overhead. Such an approach greatly improves the throughput of event processing. In the remainder of this section, we present the build time data-flow-analysis techniques and then the scheduling algorithm.

Build-time model analysis

In this section, we present the model analysis techniques that derive from the execution policy for each rule. We start with an analysis of how the metric or situation is computed. For each metric or situation, a dependence tree is generated based on the expressions in the rules that compute its value. There are three kinds of nodes in a dependence tree: the *element node* (represents the output of an expression), the *operator node* (child nodes of the element node that represent operators of the expression), and the *dependent element node* (child nodes of an operator node that represent operands of the expression). A dependent element node can also be considered as an element node if it is not an event node. For example, the metric ζ_8 associates an expression $\zeta_8 = \chi_9\ (\zeta_6, \zeta_7)$ in rule R_8. Therefore, the node ζ_8 has a child operator node χ_9, and the operator has two child operator nodes, ζ_6 and ζ_7. The tree can continue to grow until the dependent element is an event element (e.g., γ_1) or the node has already appeared in the tree (e.g., the node ζ_6 and ζ_3 in ***Figure 3A***). In the case where multiple expressions are associated with an element node, the operator nodes form an OR relationship. In the example shown in Figure 3A, the node ζ_6 has two expressions associated with it: $\zeta_6 = \chi_2(\zeta_3)$ and $\zeta_6 = \chi_3(\zeta_6, \gamma_2)$. Therefore, the nodes χ_2 and χ_3 form an OR relationship. When the construction is completed, the tree with OR nodes can be split into multiple trees with no OR nodes, as shown in ***Figures 3B*** and ***3C***.

After generating the dependence tree for each metric or situation, we can use them to study the correct sequence in which to execute rules. Assume that rule R_i and R_j are executed by the same mediator. When they are triggered in same context instance, identifying the correct sequence in which to execute them is the key to enable wait-free concurrent threads. We consider the following cases:

Case 1: R_i and R_j have the same output. In this case, these two rules cannot be executed concurrently in the same context instance. Further, if the expression in a rule realizes a self-data-flow transition (i.e., an output is one of the operands), then the rule needs to be executed later. For example, both R_1 and R_{12} are processed by mediator M_0, and the output is ζ_1. However, the R_{12} expression has a self-data-flow

transition for ζ_1. Therefore, R_{12} needs to be executed after R_1, when ζ_1 is not available.

Case 2: R_i and R_j have different outputs. This case can be further classified into the following two situations. First, R_i appears in any of the R_j expression trees, which indicates that R_i may provide operands for R_j. Therefore, R_i needs to be executed before R_j if the R_j operands are not available. Second, both R_i and R_j do not appear in the expression tree of the other, which indicates that R_i and R_j have no data-flow associations. Therefore, R_i and R_j can be executed concurrently.

By accumulating the execution order for each pair of rules, an execution policy is derived for each rule. There are two types of execution policies:

1. The rule is initiated to be executed as long as no other rule of the same type is being executed in the same context instance. Such a policy is applied to a rule that has no other prerequisite rule in the execution order.
2. The rule is initiated to be executed only when the execution of prerequisite rules in the execution order is completed in the same context instance and there is no other instance of the same type of rule executed in the same context instance. Such a policy is applied to a rule that has some prerequisite rules in the execution order.

For example, the mediator M_0 possesses three rules: R_1, R_2, and R_{12}. Both R_1 and R_{12} can be executed concurrently with R_2, while R_{12} has to be executed after R_1. Therefore, R_1 and R_2 belong to first case. R_{12} belongs to second case, as it has a prerequisite rule R_2.

Scheduling algorithm

In runtime, based on the execution policies, the mediator adopts scheduling algorithms to determinate whether a triggered rule can be executed immediately or if it must be initialized and queued as a work item. For a rule that can be executed immediately, the following algorithm is used, where r_i is the rule instance that is triggered, *c* is the context instance that r_i belongs to, and `c.exQueues[i]` is the queue of rule instances for rule type R_i of r_i:

```
begin
  if a rule instance of r_i is being executed in c by
      thread t then
    add r_i instance into the t execution queue;
  else if there is a free thread t in the thread
      pool then
    add r_i instance into c.exQueues[i];
    move all the items from c.exQueues[i] to the t
        queue;
    start thread t execution;
  else
    add r_i instance into c.exQueues[i];
end
```

In the algorithm, for each context instance *c*, `exQueues` is an array of queues with one queue for each rule type in the context type. The queue array buffers the work items that are ready for execution. Each thread in the thread pool also maintains a queue that buffers the work items that need to be executed in the lifespan of the thread.

For example, when the mediator M_0 receives an event γ_1 in the context instance *c*, the execution of r_2 (instance of R_2) is initiated. Therefore, if there is a thread *t* executing another instance of R_2 in the same context instance *c*, the scheduler adds the r_2 into the queue of thread *t*. Therefore, the execution of r_2 is started when it comes to the front of the queue for *t*. If, in the context instance *c*, there is no other R_{12} instance being executed, the scheduler allocates a free thread to execute r_2 or buffers r_2 in the context *c* queue if there is no free thread available in the thread pool.

On the other hand, for a rule that must be initialized and queued as a work item, the following algorithm is applied:

```
begin
  if another rule instance of R_i is being executed
      in c by thread t
  then
    add r_i into the t execution queue;
  else
    add r_i instance into c.exQueues[i];
    if c.executable[i] and there is a free thread
        t in the thread pool
    then
      add r_i into the t queue;
      move all the items from c.exQueues[i] to the
          t queue;
      start thread t execution;
end
```

In particular, in each context instance *c*, a Boolean array `c.executable` is used (one entry for a rule) to indicate whether that rule type is ready to be executed. For those rules that have a previous rule in the execution order, the initial value for the entry is false. If there is already a thread that is currently executing the same rule in the same context instance, the new instance is added to the thread execution queue. Otherwise, a check is made to determine whether the rule is ready to be executed, and then the thread is allocated to start the execution of the rule instances.

When a thread has completed the execution of one rule instance, it checks whether the completion of the rule enables other types of rules to be ready for execution in the same context instance, and it sets the value of `c.executable[i]`. For example, when an execution of rule instance R_2 is completed in context instance *c*, the rule R_{12} becomes ready to be executed. Further, when a thread has completed all the work items in its queue, it starts to execute work items in other rule queues in the context instance, if there are any types of rules ready to be executed.

Queuing-network performance tuning

The queuing network constructed by mediators is self-tuning; the number of threads in each mediator is automatically adjusted. The runtime engine monitors the incoming event rate, queuing lengths, and processing rates in each mediator. This data is used to adjust the number of threads in each mediator. By doing this, the CPU resource can be allocated and reallocated to avoid bottlenecks occurring in the queuing network, thus improving the dynamic throughput. We employed both queuing theory and feedback control theory to tune the performance. We also studied the relationship between the types of observation models and the most effective tuning approach.

EXPERIMENT AND PERFORMANCE EVALUATION

This section describes a series of experiments to demonstrate the effectiveness of model analysis techniques.

Experimental setup

To conduct the experiments, we adopted an observation model that monitors customers' use of credit. The types of business events to be processed included *customer event, credit transaction event, credit transaction close event, dispute identified event*, and *dispute resolved event*. About 300 rules were used to process these business events, and approximately 300 metrics—such as, the number of open transactions, total open-transaction amount, total due amount, overdue amount at 31, 61, 91, and 181 days, total amount of dispute, and average number of days to resolve a dispute—were computed. About ten situations—such as the total amount due exceeding a threshold, ongoing disputes, and average number of days to resolve a dispute exceeding a threshold—were detected.

To test the system, we designed an event emitter that sends business events with a given sending rate (i.e., number of events per second) and deployed it on JVM 1.4.2. The observation manager is deployed on IBM WebSphere* Enterprise Service Bus 6.0.2 as a message-driven bean (a reusable software component). It consists of an event-processing engine and a model-driven cache. The persistent data store is deployed on an IBM DB2* Enterprise Server 8.2, and Java Data Base Connectivity** is used to connect it with the observation manager. In each experiment, we change the event sending rate to test the maximum event throughput supported by the observation manager. The event throughput is computed as the average of 10 runs.

Experiment results

First, we compared the solutions with and without the cache component. Without the cache, the rule execution was encoded as an SQL statement.[3] By deploying the event-processing engine on the same hardware-configured host and varying the hardware configurations on the database server, we ran two sets of tests. The performance comparison results are shown in ***Figure 4A***. In both sets of tests, the observation manager with the cache greatly outperformed the one without the cache. The low-end solution consisted of one Intel Xeon** CPU with 2 GB of random access memory (RAM); the high-end solution consisted of two Intel Xeon CPUs with 4 GB of RAM. With the high-end configuration, the event throughput was more than double that of the low-end configuration, verifying our design rationale; that is, if a cache is not adopted, event throughput greatly depends on the hardware configuration of the database server. With a cache, events were processed at the rate of 82.1 events/sec in the low-end database server, and 105.3 events/sec in the high-end database server. This result verifies the effectiveness of the cache and confirms

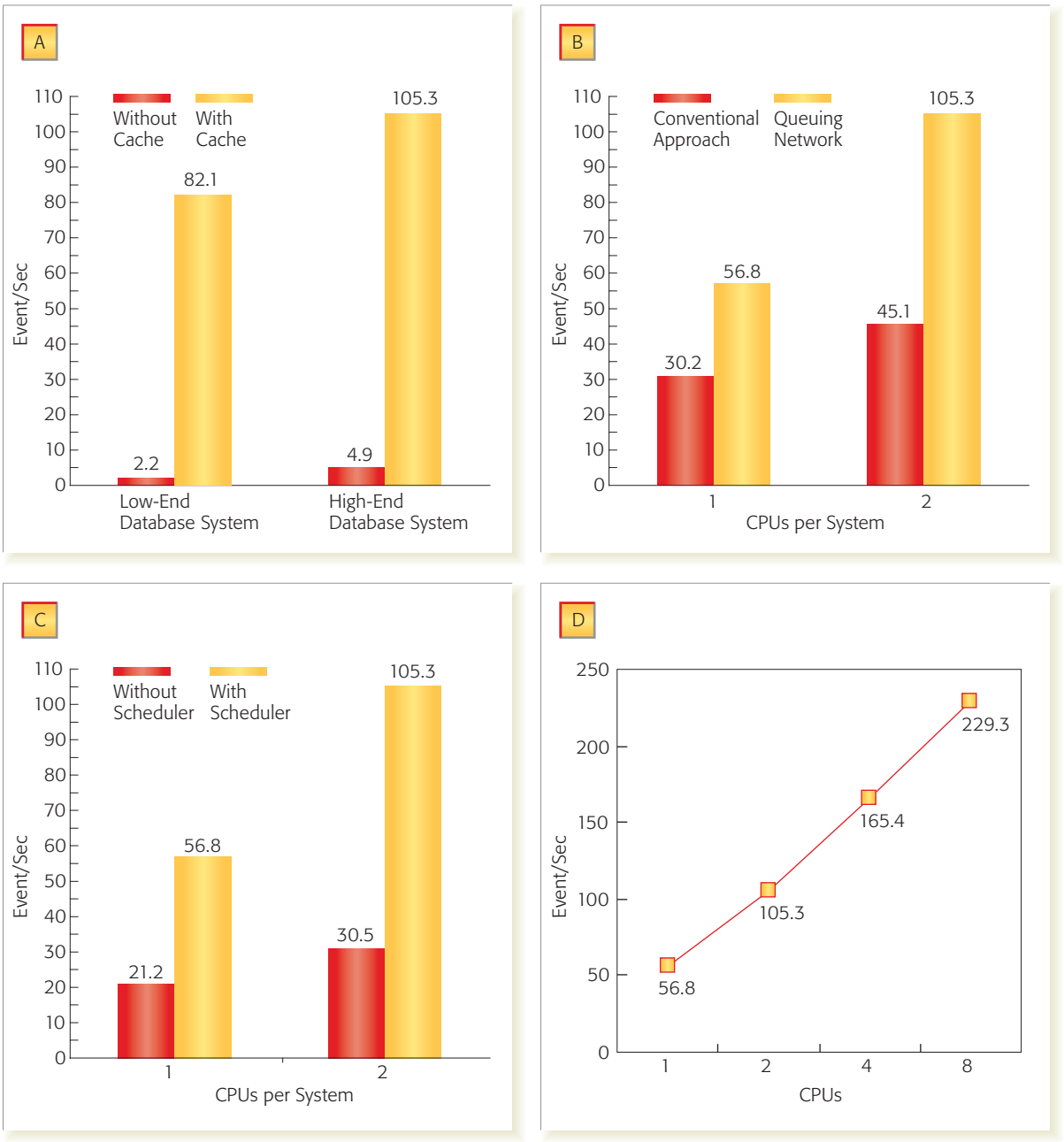

Figure 4
Experimental results: (A) model-driven cache, (B) queuing network, (C) effectiveness of scheduler, and (D) scalability

that the event throughput of the system with the cache does not rely on the performance of the persistent data store.

Next, we evaluated the multimediator framework by comparing the conventional approach that generates an independent Java class for all the rules that process a particular type of business event. In our multimediator framework, about 300 independent Java classes were generated: one class for each rule. Based on the algorithm in the section "Multimediator orchestration," seven mediators are deployed to invoke these Java classes. In the conventional approach, five independent Java clas-

ses are generated, as there are five types of business events. We designed a thread pool to invoke these five Java classes so that multiple business events could be processed concurrently. In both approaches, we tuned the number of threads in the pool to optimize event throughput. The results (*Figure 4B*) indicate that the multimediator queuing framework significantly outperforms the conventional approach.

We then evaluated the effectiveness of the data-flow-driven scheduler. First, we turned on the scheduler and, as it was not necessary, turned off the lock-based concurrency control in the cache, and ran the performance test. We then turned off the scheduler and turned on the lock-based concurrency control in the cache and ran the performance test. Comparing the results (*Figure 4C*) shows that the event-processing engine equipped with the scheduler outperformed the one without the scheduler. For example, in a four-CPU system, the scheduling approach resulted in event throughput being about three times greater than the concurrency control approach (105.3 event/sec compared with 30.5 event/sec).

Lastly, we deployed the observation manager on systems with different numbers of CPUs (one, two, four, and eight) to study the scalability. The results (*Figure 4D*) show that event throughput scaled nearly linearly to the number of CPUs in the system.

RELATED WORK

The concept of composite event processing was first introduced in active databases.[8–11] The semantics of event operators and time windows were introduced to define the composite event. However, database systems were not optimized for complex event processing because of their "persist and then process" schema. Middleware solutions like that of Adi and Etzion[12] extend the concept of composite events and aim for situation detection. By maintaining the state information and performing composite event operators in main memory, the system can achieve high event throughput. However, state persistence is not supported. Further, metric computation is not supported, which is the key feature of business event processing.

The continual query system[13] is able to monitor updates in areas of interest and return results whenever the updates reach specified thresholds. It provides push-enabled, event-driven, content-sensitive information delivery capabilities, which can be considered as the enabling technology for implementing situation detection. However, there is no discussion of how to efficiently perform the state computation and persist the state information.

Publish-and-subscribe systems[14–17] focus on event filtering. The main design challenge in these systems is to process the large number of event filters on a high volume of events. However, these systems do not support either state computation (e.g., metric value computation) or state persistence. The complexity of business event processing differs from the publish/subscribe system in the amount of data produced. When processing business events, a large number of metric computation expressions on a high volume of events need to be computed in real time.

Data-stream management systems[15,18–22] consider the collections of events in certain time windows as streams. These systems enable some relational operators to process streams and create output streams. They process live event data in real time. However, the technique is not sufficient to support efficient metric computation. First, the focus is on an approximate query, whereas business event processing requires precise data processing and computation of exact metric values. Second, most data-stream management systems do not consider the persistence of process results.

The stage-based architecture was first proposed in Staged Event-Driven Architecture[23] (SEDA) for deploying highly concurrent Internet services. SEDA decomposes an event-driven application into a collection of stages connected by queues in order to prevent over-committing resources when demand exceeds service capacity. SEDA does not consider the data-flow constraints among the threads, which is the primary bottleneck for an event-processing system. The stage-based architecture is also proposed for constructing a relational query engine,[24] which aims at optimizing the performance of the memory hierarchy, which is the primary bottleneck for data-intensive applications. The stage topology in this paper is different from these approaches; it is generated based on application logic in order to optimize communication costs among the stages. Further, by scheduling work items inside each stage,

wait-free concurrent threads are enabled, which is the key to achieving high event throughput.

CONCLUSION

In this paper, we proposed a series of model-analysis techniques to improve the event throughput of business event processing. A novel hybrid compilation and interpretation framework was proposed to execute event-processing rules. At build time, after transformations, the model information is rearranged, and a collection of executable code and cache modules are generated. At runtime, the model-driven multimediators interpret transformed model information to orchestrate these generated codes. Also, a model-driven plan was adopted to enable wait-free concurrent threads for event processing. Our experiments illustrated that the model-driven cache modules in the data store play a key role in event throughput improvement. Further, the experiments showed that integration with a model-driven scheduler enabled our model-driven multimediator to outperform the conventional concurrent thread approach. The experiments demonstrated that the multimediator architecture scaled up according to the number of CPUs in the system. Currently, there are several ongoing customer engagement efforts to further verify our solution. Our plans for future work include supporting metric networks (i.e., probabilistic system dynamics and extensible user-defined dependency) and performing reliability studies.

CITED REFERENCES

1. P. Chowdhary, K. Bhaskaran, N. S. Caswell, H. Chang, T. Chao, S.-K. Chen, M. Dikun, et al., "Model Driven Development for Business Performance Management," *IBM Systems Journal* **45**, No. 3, 587–605 (2006).
2. L. Zeng, H. Lei, M. Dikun, H. Chang, and C. Shu, "Dynamic Evolution of Business Performance Management," *Proceedings of the IEEE International Conference on e-Business Engineering*, Shanghai, China (2006), pp. 415–424.
3. L. Zeng, H. Lei, M. Dikun, H. Chang, and K. Bhaskaran, "Model-Driven Business Performance Management," *Proceedings of the IEEE International Conference on e-Business Engineering*, Beijing, China (2005), pp. 295–304.
4. Complex Event Processing, http://en.wikipedia.org/wiki/Complex_event_processing.
5. D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Professional, Boston, MA (2002).
6. J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman, "Compiled Query Execution Engine Using JVM," *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, Atlanta, GA (2006), p. 23.
7. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*, Addison-Wesley Professional, Boston, MA (2006).
8. N. H. Gehani, H. V. Jagadish, and O. Shmueli, "Composite Event Specification in Active Databases: Model & Implementation," *Proceedings of the 18th International Conference on Very Large Databases*, Vancouver, Canada (1992), pp. 327–338.
9. S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *Data & Knowledge Engineering* **14**, No. 1, 1–26 (1994).
10. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts and Detection," *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile (1994), pp. 606–617.
11. D. Zimmer and R. Unland, "On the Semantics of Complex Events in Active Database Management Systems," *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia (1999), pp. 392–399.
12. A. Adi and O. Etzion, "Amit—The Situation Manager," *The International Journal on Very Large Data Bases* **13**, No. 2, 177–203 (2004).
13. L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery," *IEEE Transactions on Knowledge and Data Engineering* **11**, No. 4, 610–628 (1999).
14. L. Zeng and H. Lei, "A Semantic Publish/Subscribe System," *Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business*, Beijing, China (2004), pp. 32–39.
15. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching Events in a Content-Based Subscription System," *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, Atlanta, GA (1999), pp. 53–61.
16. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA (2001), pp. 115–126.
17. A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards Expressive Publish/Subscribe Systems," *Proceedings of the International Conference on Extending Database Technology*, Munich, Germany (2006), pp. 627–644.
18. J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek, "Beyond Average: Toward Sophisticated Sensing with Queries," *Proceedings of the Workshop on Information Processing in Sensor Networks (IPSN)*, Palo Alto, CA (2003), http://db.cs.berkeley.edu/papers/ipsn03-beyondavg.pdf.

19. Borealis: Distributed Stream Processing Engine, Brandeis University, Brown University, and the Massachusetts Institute of Technology, http://www.cs.brown.edu/research/borealis/public/.
20. TelegraphCQ, University of California–Berkeley, Computer Science Division, http://telegraph.cs.berkeley.edu/index.html.
21. STREAM, Stanford Stream Data Manager, Stanford University, http://infolab.stanford.edu/stream/.
22. E. Wu, Y. Diao, and S. Rizvi, "High-Performance Complex Event Processing Over Streams," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL (2006), pp. 407–418.
23. M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Canada (2001), pp. 230–243.
24. S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: A Simultaneously Pipelined Relational Query Engine," *Proceedings of the SIGMOD International Conference on Management of Data*, Baltimore, MD (2005), pp. 383–394.

Accepted for publication May 21, 2007.
Published online September 26, 2007.

Liangzhao Zeng
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (lzeng@us.ibm.com). Dr. Zeng is a researcher in the Business Informatics department. He received a Ph.D. degree in computer science from the University of New South Wales, Australia. He works in the areas of enterprise data management, event processing, and services engineering, with a focus on system infrastructure and data management issues. His recent research interests include complex event processing, semantic data management, and service composition.

Hui Lei
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (hlei@us.ibm.com). Dr. Lei is a research staff member. He received a Ph.D. degree in computer science from Columbia University. He works in the areas of enterprise computing, mobile computing, and services engineering, with a focus on software infrastructure and data management issues. He is a past chair of the Mobile Computing Professional Interest Community at IBM Research. Dr. Lei is a senior member of IEEE.

Teruo Koyanagi
Tokyo Research Laboratory, IBM Japan, Ltd., 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan (teruok@jp.ibm.com). Mr. Koyanagi is a research staff member. He received B.S. and M.S. degrees in science engineering from the University of Tsukuba, Japan. His recent research interests include cache technologies, application optimization, and business event processing.

Hiroyasu Ohsaki
IBM Yamato Software Laboratory, IBM Japan Ltd., 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan (ohsaki@jp.ibm.com). Mr. Ohsaki is an advisory software engineer. He received a B.S. degree in mechanical engineering from the University of Electro-Communications, Japan. His current interest is in service-oriented architecture solutions with the WebSphere Process Server and Business Monitor.

Henry Chang
IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (hychang@us.ibm.com). Dr. Chang is a Senior Technical Staff Member and a research manager in the Business Informatics department. He received a B.S. degree in electrical engineering from the National Taiwan University, and M.S. and Ph.D. degrees in computer science from the University of Wisconsin–Madison. He leads the research effort in business performance monitoring and management framework with technical impacts on WebSphere business process management suites and IBM internal supply-chain-visibility initiatives. He received an IBM Innovate Award for his work on business-to-business collaboration solutions. Dr. Chang is a longtime member of ACM and IEEE. ■