Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Matthias Boehm, Dirk Habich, Steffen Preissler, Wolfgang Lehner, Uwe Wloka

Vectorizing Instance-Based Integration Processes

Erstveröffentlichung in / First published in:

Enterprise Information Systems. 11th International Conference. Milan, 06.-10.05.2009. Springer, S. 40-52. ISBN 978-3-642-01347-8.

DOI: https://doi.org/10.1007/978-3-642-01347-8_4

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-822777

SLUB Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA Quality Content of Saxony

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

Vectorizing Instance-Based Integration Processes

Matthias Boehm[1], Dirk Habich[2], Steffen Preissler[2], Wolfgang Lehner[2], and Uwe Wloka[1]

[1] Dresden University of Applied Sciences, Database Group
mboehm@informatik.htw-dresden.de,
wloka@informatik.htw-dresden.de
[2] Dresden University of Technology, Database Technology Group
dirk.habich@tu-dresden.de,
steffen.preissler@tu-dresden.de,
wolfgang.lehner@tu-dresden.de

Abstract. The inefficiency of integration processes—as an abstraction of workflow-based integration tasks—is often reasoned by low resource utilization and significant waiting times for external systems. Due to the increasing use of integration processes within IT infrastructures, the throughput optimization has high influence on the overall performance of such an infrastructure. In the area of computational engineering, low resource utilization is addressed with vectorization techniques. In this paper, we introduce the concept of vectorization in the context of integration processes in order to achieve a higher degree of parallelism. Here, transactional behavior and serialized execution must be ensured. In conclusion of our evaluation, the message throughput can be significantly increased.

Keywords: Vectorization, Integration processes, Throughput optimization, Pipes and filters, Instance-based.

1 Introduction

Integration processes—as an abstraction of workflow-based integration tasks—are typically executed with the *instance-based execution model*. This implies that incoming messages are serialized in incoming order, and this order is then used to execute single-threaded instances of process plans. Example system categories for that execution model are EAI (Enterprise Application Integration) servers, WfMS (Workflow Management Systems) and WSMS (Web Service Management Systems). Workflow-based integration platforms usually do not reach high resource utilization because of (1) the existence of single-threaded process instances in parallel processor architectures, (2) significant waiting times for external systems, and (3) IO bottlenecks (message persistence for recovery processing). Hence, the throughput—in the sense of processed integration process plan instances per time period—is not optimal and can be significantly optimized using a higher degree of parallelism. The opposite to the *instance-based execution model* is the *pipes and filters execution model*. Here, each operator is conceptually a single thread, and each edge between two operators contains a message queue. Hence, a high degree of parallelism is reached. This is typical for DSMS (Data Stream Management Systems) and ETL (Extraction Transformation Loading) tools.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009",
S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

Our approach is to introduce the vectorization of integration processes as an internal optimization concept in order to increase the throughput of integration platforms. We use the term vectorization in the sense of a transformation from the instance-based to the pipes-and-filters execution model. Note that this is an analogy to computational engineering, where vectorization is classified (according to Flynn) as SIMD (single instruction, multiple data) or in special cases as MIMD (multiple instruction, multiple data). We use this analogy because in the pipes-and-filters execution model, sequences (vectors) of messages are executed by a single operator. Here, specific constraints like the serialization of external behavior and the transactional behavior (recoverability) must be ensured. Finally, there is the need for execution model transparency. Thus, the user should think of an *instance-based execution model* as the used logical model.

In order to overcome the problem of low message throughput (caused by low resource utilization), we make the following contributions:

- In Section 2, we explain requirements for integration processes and we formally define the integration process vectorization problem.
- Subsequently, in Section 3, we introduce our novel approach for process plan rewriting in order to apply the vectorization of process instances.
- Based on those details, we present selected results of our exhaustive experimental evaluation in Section 4.

Finally, we analyze related work in Section 5 and conclude in Section 6.

2 Problem Description

In this section, we emphasize the assumptions and requirements that lead to our idea of throughput optimization. Here, we formally define the integration process vectorization problem, survey possible application areas, and finally give a solution overview.

2.1 Assumptions and Requirements

Figure 1 illustrates a generalized integration platform architecture for instance-based integration processes. Here, the key characteristics are a set of inbound adapters (passive listeners), several message queues, a central process engine, and a set of outbound adapters (active services). The message queues are used as logical serialization elements within the asynchronous execution model. However, the synchronous as well as the asynchronous execution of process plans is supported. Further, the process engine

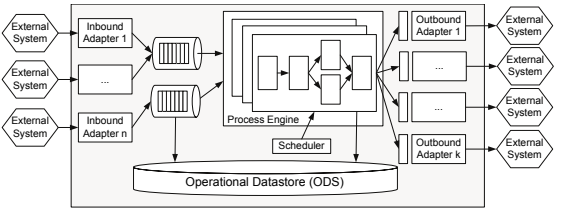


Fig. 1. Integration Platform Architecture

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009",
S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

is instance-based, which means that for each subsequent message in a queue, a new instance (one thread) of the specified process plan is created and executed serially.

In the context of integration processes, the throughput maximization rather than the execution time minimization is the major optimization objective. Further, we assume that integration platforms typically do not have a 100-percent resource utilization. This is mainly caused by (1) significant waiting times for external system invocations, (2) the trend towards multi-processor architectures, and (3) the IO bottleneck due to the need for message persistence for recoverability issues. Hence, by increasing the degree of parallelism, the message throughput can be significantly improved.

Due to the need for logical serialization of process plan instances, simple multi-threading of single instances is not applicable. As presented in [1], we must ensure that messages do not outrun other messages; for this purpose, we use logical serialization concepts such as message queues.

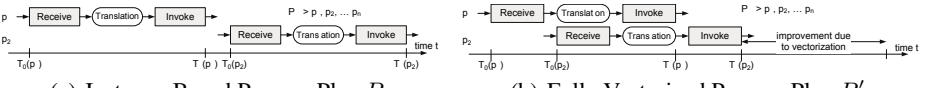
Example 1. Message Outrun Anomaly: *Assume two message types: orders, M_O, and customer, M_C. Messages of those different types are executed by different integration processes P_O and P_C with $M_O \rightarrow P_O$ and $M_C \rightarrow P_C$. Both process types comprise the receipt of a message, the schema mapping and the invocation of an external system s_1. Further, assume that the customer master data must be propagated to the external system s_1 before the customer's first order can be processed. In addition to that, the inventory is maintained during order processing. In the serialized case, messages of both types are serialized. Hence, they cannot outrun each other. In the non-serialized case, an order message can outrun the corresponding customer information. This might result in a referential integrity conflict within the target system s_1.*

However, the serialized execution of process instances is not always required. We can weaken this to serialized external behavior of process plan instances.

2.2 Optimization Problem

Now, we formally define the integration process vectorization problem. Figure 2(a) illustrates the temporal aspects of a typical instance-based integration process. Here, a message is received from a message queue (`Receive`), then a schema mapping (`Translation`) is processed and finally, the message is sent to an external system (`Invoke`). In this case, different instances of this process plan are executed in serialized order. In contrast to this, Figure 2(b) shows the temporal aspects of a vectorized integration process. Here, only the external behavior (according to the start time T_0 and the end time T_1 of instances) must be serialized. The problem is defined as follows:

Definition 1. Integration Process Vectorization Problem (IPVP): *Let P denote a process plan and p_i with $p_i = (p_1, p_2, \ldots, p_n)$ denotes the process plan instances with*



(a) Instance-Based Process Plan P (b) Fully Vectorized Process Plan P'

Fig. 2. Vectorization of Integration Processes

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

$P \Rightarrow p_i$. Further, let each process plan P comprise a graph of operators $o_i = (o_1, o_2, \ldots, o_m)$. Due to serialization, the process plan instances are executed with $T_1(p_i) \leq T_0(p_{i+1})$. Then the integration process vectorization problem describes the search for the derived process plan P' that exhibits the highest degree of parallelism for the process plan instances p'_i such that the constraint conditions $(T_1(p'_i, o_i) \leq T_0(p'_i, o_{i+1})) \wedge (T_1(p'_i, o_i) \leq T_0(p'_{i+1}, o_i))$ hold and the semantic correctness is ensured.

Based on the IPVP, we investigate the static cost analysis, where in general, cost denotes the execution time. If we assume an operator sequence o with constant operator costs $C(o_i) = 1$, we get

$$\begin{aligned} C(P) &= n \cdot m && \text{// instance-based} \\ C(P') &= n + m - 1 && \text{// fully vectorized} \\ \Delta(C(P) - C(P')) &= (n-1) \cdot (m-1) \end{aligned}$$

where n denotes the number of process plan instances and m denotes the number of operators. Clearly, this is an idealized model, while typically lower improvements are reachable. Those depend on the most time-consuming operator o'_k with $C(o'_k) = \max_{i=1}^{m} C(o'_i)$ of a vectorized process plan P', where we get

$$\begin{aligned} C(P) &= n \cdot \sum_{i=1}^{m} C(o_i) \\ C(P') &= (n + m - 1) \cdot C(o'_k) \\ \Delta(C(P) - C(P')) &= n \sum_{i=1 \wedge i \neq k}^{m} C(o_i) - (n + m - 1) \cdot C(o'_k)\ . \end{aligned}$$

Obviously, $\Delta(C(P) - C(P'))$ can be negative in case of a very small n. However, with an increasing n, the performance improvement grows linearly.

2.3 Solution Overview

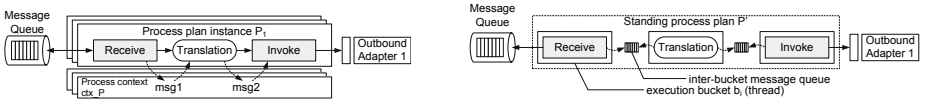
Here, we want to give a solution overview of our process plan vectorization approach. According to the generalized integration platform architecture, this exclusively addresses the process engine, while all other components can be reused without changes.

The core idea is to rewrite the instance-based process plan—where each instance is executed as a thread—to a vectorized process plan, where each operator is executed as a single *execution bucket* and hence, as a single thread. Thus, we model a standing process plan. Due to different execution times of the single operators, inter-bucket queues (with max constraints) are required for each data flow edge. Figure 3 illustrates those two execution models. Although significant performance improvement is possible, major challenges arise when rewriting P to P'.

Here, the main goal when rewriting a process plan P is the transparency of the used execution model. Hence, a user should only recognize the instance-based execution model, while internally (and transparent in the sense of being hidden to the user), the *vectorized execution model* is used. This aim poses several research challenges. This

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009",
S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4



(a) Instance-Based Process Plan P (b) Fully Vectorized Process Plan P'

Fig. 3. Different Execution Models

includes (1) ensuring *semantical correctness* of P', (2) preserving the *external behavior*, (3) ensuring *transactional behavior and recoverability*, and (4) realizing both the synchronous (simulated for P') as well as the asynchronous execution models. Finally, we must (5) handle the rewriting of different data flow concepts (from instance-based process plans, which use a variable-based data flow, to vectorized process plans that exhibit an explicit data flow (pipelining)). In order to overcome Problems 1-3, we present specific rewriting rules. Problem 4 is tackled with an extended message model. Finally, we propose operator-aware rewriting techniques in order to overcome Problem 5.

In the rest of the paper, we provide the details on how to rewrite an instance-based process plan into a vectorized process plan. Further, in Section 4, we present selected results of an exhaustive evaluation.

3 Rewriting Process Plans

In this section, we explain in detail how to rewrite instance-based process plans to fully vectorized process plans.

3.1 Message Model and Process Model

As formal foundation, we use the instance-based *Message Transformation Model (MTM)*. Hence, we have to define extensions in order to make it applicable also in the context of vectorized integration processes (then we refer to it as *VMTM*). Both consist of a message model and a process model.

We model a message m of a message type M as a quadruple with $m = (M, S, A, D)$, where M denotes the message type, S denotes the runtime state, and A denotes a map of atomic name-value attribute pairs with $a_i = (n, v)$. Further, D denotes a map of message parts, where a single message part is defined with $d_i = (n, t)$. Here, n denotes the part name and t denotes a tree of named data elements. In the VMTM, we extend it to a quintuple with $m = (M, C, S, A, D)$, where the context information C denotes an additional map of atomic name-value attribute pairs with $c_i = (n, v)$. This extension is necessary due to parallel message execution within one process plan.

A process plan P is defined with $P = (o, c, s)$ as a 3-tuple representation of a directed graph. Let o with $o = (o_1, \ldots, o_m)$ denote a sequence of operators, let c denote the context of P as a set of message variables msg_i, and let s denote a set of services $s = (s_1, \ldots, s_l)$. Then, an instance p_i of a process plan P, with $P \Rightarrow p_i$, executes the sequence of operators once. Each operator o_i has a specific type as well as an identifier NID (unique within the process plan) and is either of an *atomic* or of a *complex* type. Complex operators recursively contain sequences of operators with

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

$o_i = (o_{i,1}, \ldots, o_{i,m})$. Further, an operator can have multiple input variables $msg_i \in c$, but only one output variable $msg_j \in c$. Each service s_i contains a type, a configuration and a set of operations. Further, we define a set of interaction-oriented operators iop (`Invoke`, `Receive` and `Reply`), control-flow-oriented operators cop (`Switch`, `Fork`, `Iteration`, `Delay` and `Signal`) and data-flow-oriented operators dop (`Assign`, `Translation`, `Selection`, `Projection`, `Join`, `Setoperation`, `Split`, `Orderby`, `Groupby`, `Window`, `Validate`, `Savepoint` and `Action`). Furthermore, in the *VMTM*, the flow relations between operators o_i do not specify the control flow but the explicit data flow in the form of message streams. Additionally, the `Fork` operator is removed due to redundancy. Finally, we introduce the additional operators `AND` and `XOR` (for synchronizing the serialized external behavior) as well as the `COPY` operator (for supporting the changed data flow).

3.2 Rewriting Algorithm

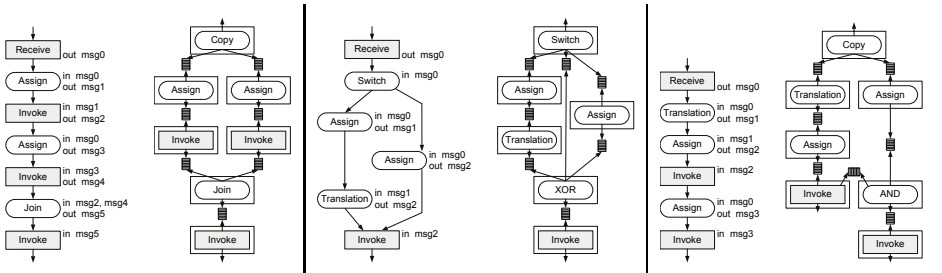
Now, let us focus on the realization of such process plan rewriting; even without considering transactional behavior and cost analysis, it is already very complex.

Algorithm 1. Process Plan Vectorization.

Require: operator sequence o
1: $B \leftarrow \oslash, D \leftarrow \oslash, Q \leftarrow \oslash$
2: **for** $i = 1$ to $|o|$ **do**
3: // $\forall$ operators
4: **for** $j = i$ to $|o|$ **do**
5: // $\forall$ following operators
6: **if** $\exists o_i \xrightarrow{\delta} o_j$ **then**
7: $Q \leftarrow Q \cup q$ with $q \leftarrow$ create queue
8: $D \leftarrow D \cup d < o_i, q, o_j >$ with $d < o_i, q, o_j > \leftarrow$ create dependency
9: **end if**
10: **end for**
11: **if** $o_i \in Switch, Iteration, Fork, Savepoint, Invoke^*$ **then**
12: // see Subsubsections 3.2.2 and 3.2.3
13: **else**
14: $b_i(o_i) \leftarrow$ create bucket over o_i
15: **for** $k = 1$ to $|D|$ **do**
16: // foreach dependency
17: $d < o_x, q, o_y > \leftarrow d_k$
18: **if** $o_i \equiv o_x$ **then**
19: connect $b_i(o_i) \rightarrow q$
20: **else if** $o_i \equiv o_y$ **then**
21: connect $q \rightarrow b_i(o_i)$
22: **end if**
23: **end for**
24: $B \leftarrow B \cup b_i(o_i)$
25: **end if**
26: **end for**
27: **return** B

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009",
S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4



(a) Plan P (b) Plan P' (c) Plan P (d) Plan P' (e) Plan P (f) Plan P'

Fig. 4. Rewriting Examples (core concept, context-specific and serialized external behavior)

Rewriting Unary and Binary Operators. When rewriting instance-based process plans to vectorized process plans, we distinguish between unary operators (one input message: `Invoke`, `Assign`, `Translation`, `Selection`, `Projection`, `Split`, `Orderby`, `Groupby`, `Window`, `Action`, and `Delay`) and binary operators (multiple input messages: `Join`, `Setoperation`, and `Assign`). Both unary and binary operators can be rewritten with the same core concept (see Algorithm 1) that contains the following four steps. First, we create a queue instance for each data dependency between two operators (the output message of operator o_i is the input message of operator o_j with $j > i$). Second, we create an execution bucket for each operator. Third, we connect each operator with the referenced input queue. Clearly, each queue is referenced by exactly one operator, but each operator can reference multiple queues. Fourth, we connect each operator with the referenced output queues. If one operator must be connected to n output queues with $n \geq 2$ (its results are used by multiple following operators), we insert a `Copy` operator (gets a message from one input queue, then copies it $n-1$ times and puts those messages into the n output queues). In order to make the rewriting concept more understandable, we illustrate it using the following example.

Example 2. Vectorization of Unary and Binary Operators: *Assume a process plan P that receives a message, prepares two queries, loads data from two external sources, joins the results, and sends the final message to a third system (Figure 4(a)). If we vectorize this to P' (Figure 4(b)), we can apply the standard vectorization concept. The `Receive` operator has been removed because all operators directly read from queues. Further, the `Copy` operator has been inserted because both `Assign` operators have the same input. Additionally, there is the binary `Join` operator that reads messages from two concurrent input queues.*

Due to dependency checking, the process plan vectorization algorithm has a cubic worst-case complexity of $O(m^3) = O(m^3 + m^2)$.

Rewriting Context-Sensitive Operators. Now, we consider the context-specific operators `Switch`, `Iteration`, `Fork`, `Validate`, `Signal`, `Savepoint`, and `Reply`.

Rewriting `Switch` operators. When rewriting `Switch` operators, we must be aware of their ordered if-elseif-else semantics. Here, message sequences are routed along different switch-paths, which will eventually be merged. Assume a message sequence of

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009",
S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

msg_1 and msg_2, where msg_1 is routed to path A, while msg_2 is routed to path B. If $C(A) \geq C(B) + C(Switch_B)$, msg_2 arrives earlier at the merging point than msg_1 does. Hence, a *message outrun* has taken place. Therefore, we have introduced the `XOR` operator that is inserted just before the single switch paths are merged. It reads from all queues (including a dummy queue for synchronization), compares the timestamps of read messages and forwards the oldest.

Example 3. Rewriting `Switch` Operators. *Assume a process plan P (Figure 4(c)). If we vectorize it to P' (Figure 4(d)), we apply the `Switch`-specific rewriting technique, where we create two pipeline branches (one for each switch-path). In order to avoid message outrun, we additionally inserted the `XOR` operator and a dummy queue.*

Rewriting `Iteration` operators. Also, when rewriting `Iteration` operators, the main problem is the *message outrun*. Here, we must ensure that all iteration loops (for a message) have been processed before the next message enters. Basically, a *for each* `Iteration` is rewritten to a sequence of (1) `Split` operator, (2) operators of the `Iteration` body and (3) `Setoperation` (union all) operator. In contrast to this, iterations with *while* semantics are not vectorized (one single execution bucket).

Rewriting `Validate` and `Signal` operators. One of the major differences between the instance-based process model and the vectorized process model is the maintenance of the process context (variables). Especially when dealing with validation, signals and error handling, this becomes crucial. Therefore, we extended the message model by context C (see Subsection 3.1). In case of an error (invalidity or explicit signal), we store the specific information in correlation to the current message that caused the signal. Then we can apply recovery processing.

In summary, when rewriting context-specific operators, we want to assure the semantic correctness during the rewriting of instance-based integration processes to the vectorized process model. This is a part of the general rewriting algorithm (Algorithm 1, lines 13-14). There are additional rewriting rules for `Fork`, `Savepoint` and `Reply` operators, which we omitted here because they are straight-forward.

Serialization and Recoverability. In order to realize the serialization of external behavior (precondition for transparency of the used execution model), we must ensure that explicitly modeled sequences of `Invoke` operators are serialized. Hence, we use the `AND` operator for synchronization purposes. If an `Invoke` operator has a temporal dependency, we insert an `AND` operator right before it as well as a dummy queue between the source of the temporal dependency and the `AND` operator. The `AND` operator reads from the dependency and the original queue and synchronizes the external behavior.

Example 4. Serialization of external behavior: *Assume a process plan P (Figure 4(e)). If we vectorize this process plan to P' (Figure 4(f)) with two pipeline branches, we need to ensure the serialized external behavior. Here, we insert an `AND` operator, where the left `Invoke` sends dummy messages to this operator. Only in the case that the right `Assign` as well as the left `Invoke` have been processed successfully, the real message of the right pipeline branch is forwarded to the second `Invoke`.*

With regard to recoverability of single integration processes, we might need to execute recovery processing with loaded queues. In general, we use the `stopped` flag of a

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

queue in order to stop it in case of a failure at operator o_i. In fact, we need to stop the input queue of this operator, while all other operators can continue working. Hence, the max queue constraint will be reached and clients are blocked.

Cost Analysis. In Subsection 2.2, we illustrated the theoretical performance of a simple sequence of operators, where each operator o_i has a single data dependency with the previous operator o_{i-1}. Now, we investigate the performance with regard to specific rewriting results (the idealized cost model is reused).

Parallel data flow branches. Here, different messages are processed by $|r|$ concurrent pipelines (branches) within P'. Examples for this are simply overlapping data dependencies and the `Switch` operator. Assume an operator sequence o of length m. In the instance-based model, the costs of n instances are $C(P) = n \cdot m$. In case the operator sequence contains a single branch with $|r| = 1$, we can improve the costs by $(n-1) \cdot (m-1)$ to $n+m-1$ using process plan vectorization. In the case of multiple branches with $|r| \geq 2$, the possible improvement is given by

$$C(P') = n + \max_{i=1}^{|r|}(|r_i|) - 1$$

$$\Delta(C(P) - C(P')) = n \cdot (m-1) - \max_{i=1}^{|r|}(|r_i|) + 1.$$

Clearly, in the case of $|r| = 1$ and $|r_1| = m$, the general cost analysis stays true. In the best case, $\max_{i=1}^{|r|}(|r_i|)$ is equal to $\frac{m}{|r|} \in \mathbb{N}$. The improvement is caused by the higher degree of parallelism. However, parallel data-flow branches may also cause overhead for splitting (`Copy`) and merging (`AND` or `XOR`).

Rolled-out `Iteration`. When rewriting `Iteration` operators with *for each* semantics, we split messages according to the *for each* condition and process the iteration body as inner pipeline without cyclic dependencies. Finally, the processed sub-messages are merged using the `Setoperation` operator (union all). In the instance-based case, $C(o) = r \cdot m$ is true, where r denotes the number of iteration loops (number of sub-messages) and m denotes the number of operators in the iteration body. Due to the sub-pipelining, we can reduce the processing time to $C(o') = r + m - 1 + 2$.

3.3 Cost-Based Vectorization

The two major weaknesses of our approach are (1) that the theoretical performance of a vectorized integration process mainly depends on the performance of the most cost-intensive operator, and (2) that the practical performance also strongly depends on the number of available threads. Thus, the optimality of vectorization strongly depends on dynamic workload characteristics. Hence, future work should investigate the generalized problem description, where we search for the optimal k execution buckets (each containing a number of operators) in a cost-based manner.

4 Experimental Evaluation

In this section, we provide selected experimental results. Basically, we can state that the vectorization of integration processes leads to a significant performance improvement for different scale factors.

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

4.1 Experimental Setup

We implemented the introduced approaches within the so-called `WFPE` (workflow process engine) using Java 1.6 as the programming language. This implementation is available upon request. In general, the `WFPE` uses compiled process plans (a java class is generated for each integration process type). Furthermore, it follows an instance-based execution model. Now, we integrated components for the static vectorization of integration processes (we call this `VWFPE`). For that, new deployment functionalities were introduced (those processes are executed in an interpreted fashion) as well as several changes in the runtime environment were realized.

We ran our experiments on a standard blade (OS Suse Linux) with two processors (each of them a Dual Core AMD Opteron Processor 270 at 1,994 MHz) and 8.9 GB RAM. Further, we executed all experiments on synthetically generated XML data (using the DIPBench toolsuite [2]). In general, we used the following five aspects as scale factors: data size d of a message, the number of operators m of a process plan, the time interval t between two messages, the number of process instances n and the maximal number of messages q in a queue. Here, we measured the performance of different combinations of those. For statistical correctness, we repeated all experiments 20 times.

As base integration process for our experiments, we used a sequence of six operators. Here, a message is received (`Receive`) and then an interaction is prepared (`Assign`) and executed with the file adapter (`Invoke`). After that, the resulting message (contains orders and orderlines) is translated using an XML transformation (`Translation`) and finally sent to a specific directory (`Assign`, `Invoke`). We refer to this as $m = 5$ because the `Receive` is removed during vectorization. When scaling m up to $m = 35$, we copy and reconfigure those operators.

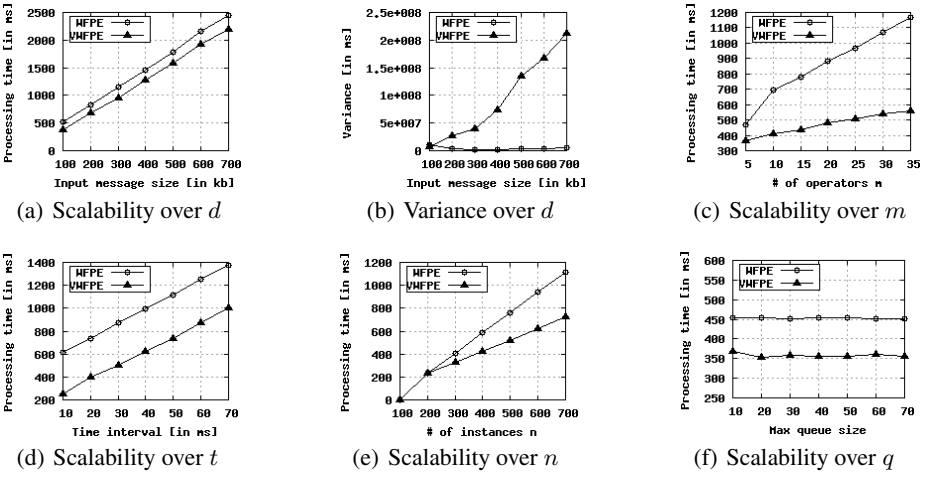
4.2 Performance and Throughput

Here we ran a series of experiments based on the already introduced scale factors. The results of these experiments are shown in Figure 5.

In Figure 5(a) we scaled the data size d of the input messages from 100kb to 700kb XML messages and measured the processing time for 250 process instances ($n = 250$) needed by the three different runtimes. There, we fixed $m = 5$, $t = 0$, $n = 250$ and $q = 50$. We can observe that both runtimes exhibit a linear scaling according to the data size and that significant improvements can be reached using vectorization. There, the absolute improvement increases with increasing data size. Further, in Figure 5(b), we illustrated the variance of this sub-experiment. The variance of the instance-based execution is minimal, while the variance of the vectorized runtime is worse because of the operator scheduling. Now, we fixed $d = 100$ (lowest absolute improvement in 5(a)), $t = 0$, $n = 250$ and $q = 50$ in order to investigate the influence of m. We varied m from 5 to 35 operators. Interestingly, not only the absolute but also the relative improvement of vectorization increases with increasing number of operators. Figure 5(d) shows the impact of the time interval t between the initiation of two process instances. For that, we fixed $d = 100$, $m = 5$, $n = 250$, $q = 50$ and varied t from 10ms to 70ms. The absolute improvement between instance-based and vectorized approaches decreases slightly with increasing t. As an explanation, the time-interval has no impact on the instance-based execution. In contrast to that, the vectorized approach depends on

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4



(a) Scalability over d (b) Variance over d (c) Scalability over m

(d) Scalability over t (e) Scalability over n (f) Scalability over q

Fig. 5. Evaluation Results for Experimental Performance

t due to the resource scheduling whenever not all of the execution buckets need CPU time. Further, we analyze the influence of the number of instances n as illustrated in Figure 5(e). Here, we fixed $d = 100$, $m = 5$, $t = 0$, $q = 50$ and varied n from 100 to 700. Basically, we can observe that the relative improvement between instance-based and vectorized execution increases with increasing n, due to parallelism of process instances. Figure 5(f) illustrates the influence of the maximal queue size q, which we varied from 10 to 70. Here, we fixed $d = 100$, $m = 5$, $t = 0$ and $n = 250$. In fact, q slightly affects the overall performance for a small number of concurrent instances n. However, at $n = 250$, we cannot observe any significant influence according to the performance for both approaches.

5 Related Work

Database Management Systems. In the context of DBMS, throughput optimization has been addressed with different techniques. One significant approach is data sharing across common subexpressions of query instances [3,4]. However, in [5] it was shown that sharing can also hurt performance. Another inspiring approach is given by staged DBMS [6]. Here, in the QPipe Project [7], each relational operator was executed as a micro-engine (one operator, many queries). Additional approaches exist in the context of distributed query processing [8,9].

Data Stream Management Systems. Further, in data stream management systems (DSMS) and ETL tools, the *pipes and filters* execution model is widely used. Examples for those systems are QStream [10], Demaq [11] and Borealis [12]. However, in DSMS, scheduling is not realized with multiple processes or threads but with central control strategies and thus, the problems addressed in this paper are not present.

Streaming Service and Process Execution. In service-oriented environments, throughput optimization has been addressed on different levels. Performance and resource

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

issues, when processing large volumes of XML documents, lead to message chunking on the service-invocation level. There, request documents are divided into chunks, and services are called for every single chunk [13]. An automatic chunk-size computation using the extremum-control approach was addressed in [14]. On the process level, pipeline scheduling was incorporated in [15] into a general workflow model to show the valuable benefit of pipelining in business processes. Further, [16] add pipeline semantics to classic step-by-step workflows.

Integration Process Optimization. This has not yet been explored sufficiently. There are platform-specific optimization approaches for the *pipes and filters* execution model, like the optimization of ETL processes [17]; there are also numerous optimization approaches for instance-based processes like the optimization of data-intensive decision flows [18], the static optimization of the control flow using critical path approaches [19] and SQL-supporting BPEL activities and their optimization [20]. Further, the execution time minimization of integration processes [21] was already investigated.

6 Conclusions

In order to optimize the throughput of integration platforms, in this paper, we introduced the concept of automatic vectorization of integration processes. We showed how integration processes can be rewritten in a transparent manner, where the internal execution model is hidden from the user in order to reach a higher degree of parallelism while ensuring the transactional behavior and external behavior similar to instance-based integration processes. Based on our experimental evaluation, we can state that significant throughput improvement is possible and the concept of process vectorization is applicable in practice. Future work should address the cost-based vectorization.

References

1. Boehm, M., Habich, D., Lehner, W., Wloka, U.: An advanced transaction model for recovery processing of integration processes. In: ADBIS (2008)
2. Boehm, M., Habich, D., Lehner, W., Wloka, U.: Dipbench toolsuite: A framework for benchmarking integration systems. In: ICDE (2008)
3. Dalvi, N.N., Sanghai, S.K., Roy, P., Sudarshan, S.: Pipelining in multi-query optimization. In: PODS (2001)
4. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. In: SIGMOD (2000)
5. Johnson, R., Hardavellas, N., Pandis, I., Mancheril, N., Harizopoulos, S., Sabirli, K., Ailamaki, A., Falsafi, B.: To share or not to share? In: VLDB (2007)
6. Harizopoulos, S., Ailamaki, A.: A case for staged database systems. In: CIDR (2003)
7. Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: Qpipe: A simultaneously pipelined relational query engine. In: SIGMOD (2005)
8. Ives, Z.G., Florescu, D., Friedman, M., Levy, A.Y., Weld, D.S.: An adaptive query execution system for data integration. In: SIGMOD (1999)
9. Lee, R., Zhou, M., Liao, H.: Request window: an approach to improve throughput of rdbms-based data integration system by utilizing data sharing across concurrent distributed queries. In: VLDB (2007)

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Enterprise Information Systems. 11th International Conference. Milan 2009", S. 40-52, ISBN 978-3-642-01347-8
https://doi.org/10.1007/978-3-642-01347-8_4

10. Schmidt, S., Berthold, H., Lehner, W.: Qstream: Deterministic querying of data streams. In: VLDB (2004)
11. Boehm, A., Marth, E., Kanne, C.C.: The demaq system: declarative development of distributed applications. In: SIGMOD (2008)
12. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR (2005)
13. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: VLDB (2006)
14. Gounaris, A., Yfoulis, C., Sakellariou, R., Dikaiakos, M.D.: Robust runtime optimization of data transfer in queries over web services. In: ICDE (2008)
15. Lemos, M., Casanova, M.A., Furtado, A.L.: Process pipeline scheduling. J. Syst. Softw. 81(3) (2008)
16. Biornstad, B., Pautasso, C., Alonso, G.: Control the flow: How to safely compose streaming services into business processes. In: IEEE SCC (2006)
17. Simitsis, A., Vassiliadis, P., Sellis, T.: Optimizing etl processes in data warehouses. In: ICDE (2005)
18. Hull, R., Llirbat, F., Kumar, B., Zhou, G., Dong, G., Su, J.: Optimization techniques for data-intensive decision flows. In: ICDE (2000)
19. Li, H., Zhan, D.: Workflow timed critical path optimization. Nature and Science 3(2) (2005)
20. Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., Kraft, T.: An approach to optimize data processing in business processes. In: VLDB (2007)
21. Boehm, M., Habich, D., Lehner, W., Wloka, U.: Workload-based optimization of integration processes. In: CIKM (2008)

Provided by Sächsische Landesbibliothek - Staats- und Universitätsbibliothek Dresden