Institute of Architecture of Application Systems

Performance Optimizations for Interacting Business Processes

Sebastian Wagner, Dieter Roller, Oliver Kopp, Tobias Unger, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{WagnerRKU2013,
  author    = {Wagner, S. and Roller, D. and Kopp, O. and Unger, T. and
               Leymann, F.},
  title     = {Performance Optimizations for Interacting Business Processes},
  booktitle = {2013 IEEE International Conference on}
               Cloud Engineering (IC2E),
  year      = {2013},
  pages     = {210--216},
  doi       = {10.1109/IC2E.2013.34},
  publisher = {IEEE Computer Society}
}
```


Universität Stuttgart
Germany

Performance Optimizations for Interacting Business Processes

Sebastian Wagner, Dieter Roller, Oliver Kopp, Tobias Unger, Frank Leymann
Institute of Architecture of Application Systems
Universtät Stuttgart
Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de

***Abstract*—Choreographies describe the interaction behavior of processes at design time: a choreography defines when messages have to be exchanged between the involved processes during their runtime. In the context of Web services and the de-facto workflow language BPEL, SOAP is used to encode the messages. When complex messages are exchanged between the processes, this can become costly and time consuming with respect to the overall execution time of a choreography. In this work, we suggest three different performance optimization techniques for workflow engines to reduce the number of message exchanges between the interacting processes and hence, to decrease the execution times and costs of the choreographies: intra-engine transport, service request caching, and inline execution. We describe how these techniques are implemented in a workflow engine. Performance measurements are carried out to determine the performance improvements that are achieved with each optimization technique. We further show that the optimizations also affect the energy consumption of the workflow engine.**

I. Introduction

Many companies use business processes to describe the activities that have to be performed to achieve a desired business objective such as the manufacturing of goods or services. The processes can be modeled with a process modeling language such as the Web Service Business Process Execution Language 2.0 (BPEL [1]) or the Business Process Model and Notation (BPMN [2]). We use BPEL in this paper as it is still the de-facto standard for *executable* business processes [3].

The different activities within a business process are often not realized by just one single process but, for organizational or complexity reasons, via several interacting processes. For instance, if a car has to be manufactured, the activities required to create the chassis of the car are implemented by the car manufacturer's business process while the activities to create the engine are implemented in the process of the engine manufacturer. This approach also helps to reduce the complexity of a process as the set of activities is distributed on several processes [4].

The interacting processes communicate with each other by exchanging messages between their communication activities, such as send and receive activities [5]. The collaboration can be modeled using interconnection models, where the publicly observable behavior of each participant is modeled as a process and where the communication activities are wired together by message links.

For execution the BPEL process models of a choreography have to be deployed to a BPEL workflow engine, such as the IBM WebSphere Process Server or the Apache Orchestration Director (Apache ODE[1]). The various processes of the choreography are either deployed and executed on the same engine or are distributed to different engines with each engine executing one or more process models of the choreography. In both scenarios messages must be exchanged between the different processes. To follow interoperability requirements, SOAP/HTTP is used as the default transport binding [6]. With SOAP as the encoding mechanism, message exchanges can be costly in terms of resource consumption, in particular with respect to CPU cycles: (i) at the sender's side the message has to be serialized from the BPEL engine's internal format to XML-based SOAP used for interoperable communication, (ii) the message has to be transferred, and (iii) the message has to be deserialized back to the engine's internal format at the receiver side. The net results are longer execution times and less throughput compared to the execution of a single process for the complete choreography. As typically the pay-per-use model is applied in Cloud environments longer execution times also result in higher costs for enacting a choreography on a workflow engine that is hosted in the cloud.

This paper proposes three different optimization techniques to reduce the overhead caused by message exchanges and to enable more efficient interaction between the processes in a single workflow engine environment. A single workflow engine is for instance used by different collaborating parties in a Community Cloud. The proposed optimization techniques are applied on various levels. *Intra-engine transport* and *service request caching* are handled at the workflow engine level reducing the number of message exchanges; *inline execution* merges the interacting process models into a single process at the choreography level. To compare the impact of the optimization techniques on the CPU utilization, we implemented them on a workflow engine and conducted performance tests whose results are also presented in this paper. As energy efficiency becomes more and more important ("3x20 in 2020", [7]), we also conducted energy

[1] http://ode.apache.org/

measurements to determine the influence of the optimization techniques on the energy consumption of the workflow engine.

The remainder of this paper is structured as follows: Section II provides the background information for the optimization techniques; Section III describes the architecture of the workflow engine where the optimizations are implemented; Section IV introduces the workflow engine optimization techniques; Section V discusses how the execution time of a choreography can be improved by merging the process models that reside on the same engine; Section VI presents and discusses the results of the performance measurements; Section VII provides an overview of related work; Section VIII finishes off by presenting the conclusions and providing an outlook on future work.

II. Preliminaries

This section provides background information about the language constructs of BPEL and the interconnection choreography modeling language BPEL4Chor [8] that is relevant to understand the performance optimization techniques discussed in this work.

BPEL offers different types of activities: `sequence` and `flow` activities are considered as structured activities as they define a control flow order on their child activities. The child activities of a `sequence` activity have to be carried out in the order they appear. For instance, in process *C*, depicted Fig. 1, the first activity *C1* is executed, then *C2*, and so on. All activities that are encompassed by a `flow` activity are performed concurrently as long as no explicit control links are specified between them.

The activities `invoke`, `receive` and `reply`, collectively called communication activities, are used to communicate with a partner Web service that is for instance implemented by another BPEL process. The `invoke` activity is used to send messages to a partner, either as an asynchronous or a synchronous `invoke`. In the asynchronous case, solely a message is sent to a `receive` activity of the partner. In the synchronous case, `invoke` additionally waits for a reply message sent by the partner. This message is sent by a `reply` activity following directly or indirectly the `receive` activity of the partner in the control flow. In the example choreography depicted in Fig. 1, synchronous communication takes only place between process *A* and *D*. With all other processes *A* communicates asynchronously. Technically, each receiving activity is represented by a corresponding WSDL operation in the Web service interface of a process. These operations are called from the partner to send messages to the process.

The `opaque` activity acts as place holder for other activities to indicate that some business logic happens there (e. g., *D2*).

The communication activities of the five process models in the example interconnection choreography in Fig. 1 are connected via seven message links *m1* to *m7*. The

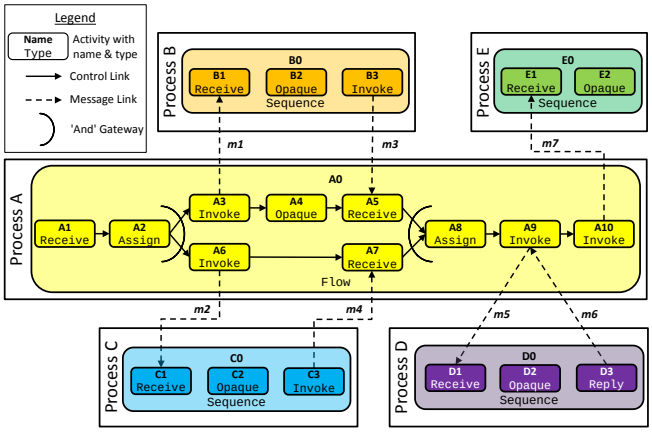

Figure 1. Example Interconnection Choreography

`receive` activity *A1* is the only communication activity that is not related to a message link as this activity is called by an external partner that triggers the execution of an instance of process model *A*. We use BPEL4Chor to describe the choreography as this language provides a means to define message links between the communication activities of BPEL processes. BPEL4Chor is solely used to model choreographies. During deployment time, each BPEL4Chor process model is transformed to an executable BPEL process model, which is deployed to the BPEL engine.

Another BPEL concept relevant for the optimization is *message correlation*. As several instances of one process model may be executed simultaneously, message correlation ensures that messages sent by a partner are routed to the correct process instance by the engine.

III. Workflow Engine

The processes models of a choreography are deployed and executed on a workflow engine, which provides the runtime environment for the instances of the process models. We implement the performance improvements on the Stuttgarter Workflow Maschine (SWoM)[2] developed at our institute.

Fig. 2 shows the architecture of the SWoM, which is based to on the architectural principles of a workflow engine introduced by Leymann and Roller [9]. This work focuses on the *Runtime* components as they are relevant for the performance optimization techniques. The *Administration* layer contains the components providing user management and process management functionalities, e. g., to monitor and repair running process instances. The *Buildtime* components are responsible for importing process models into the SWoM via the *Importer* and for the deployment of the imported process model via the *Deployer* in order to make them ready for instantiation. Deployed and imported process models are stored in the *Buildtime Database*. Each process model is associated with a *Deployment Descriptor*, that allows for

[2]http://www.iaas.uni-stuttgart.de/forschung/projects/swom/

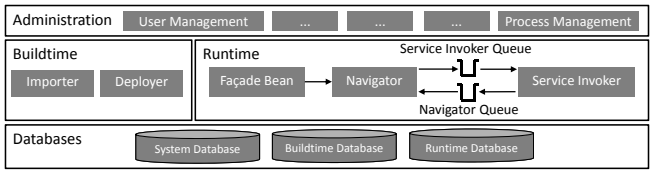

Figure 2. Architecture of SWoM

example the specification of the endpoints of the partner Web Services, or the configuration settings for the proposed engine level optimization techniques. The *Runtime Database* stores process instance data between two subsequent processing steps. The *System Database* holds administrative information, such as user authorization, authentication information, and information about exceptional states during the runtime.

The instantiation and execution of process instances is performed by the *Runtime* components that are implemented as Java Enterprise Edition stateless session beans that typically communicate with each other by exchanging messages via message queues. The *Navigator* component creates instances from process models and navigates through the process instances using the control flow information stored in the corresponding process models. If the navigator processes an `invoke` activity, it sends, via the *Service Invoker Queue*, a request message to the *Service Invoker*. The Service Invoker invokes, based on the information in the request message, the specified Web Service via an appropriate SOAP call. For a synchronous `invoke` activity the Service Invoker receives the Web service response and sends it back to the Navigator via the *Navigator Queue*. Requests from external Web services or asynchronous responses are received by the *Façade Bean*, which implements the Web Service interface that is specified for the process model.

IV. Runtime Optimizations

This section presents two runtime optimizations: intra-engine transport (Section IV-A) and service request caching (SectionIV-B).

A. Intra-Engine Transport

When BPEL processes exchange messages with partner processes, these messages have to be serialized at the sender's side from the internal engine representation, e. g., a Java object, to an agreed message format. At the receiver's side the message has to be deserialized back to the engine-internal representation. As BPEL is strongly related to WSDL, usually SOAP/HTTP is used: SOAP is the de-facto format for exchanging messages between BPEL processes and SOAP/HTTP is the binding ensuring interoperability between services [6]. As shown by Davis et al. [10] and Ng et al. [11], the serialization and deserialization of SOAP messages is time and resource intensive. Overall SOAP/HTTP implemented in Apache Axis[3] is 13 times slower than using Java RMI.

If interacting processes are deployed to the same engine this overhead can be avoided by using *intra-engine transport*. The SWoM implementation of this optimization technique bypasses the complete JAX-WS and DOM processing by having the *Invoker* calling the *Navigator* via the Navigator's local interface. Intra-engine transport can be also used if the SWoM is deployed in a clustered environment where the components of the SWoM are distributed on different nodes of a WebSphere cluster; in this case the *Invoker* calls the *Navigator* via its remote EJB interface.

Intra-engine transport can be configured in the SWoM individually for each `invoke` activity. This has the benefit that even if just a subset of processes of a choreography is deployed on one engine, intra-engine transport can be activated for this subset. For instance, if the processes *A* and *B* in Fig. 1 are deployed on the same engine whereas the other processes are distributed to different engines, intra-engine transport can be only activated between activity pairs *A3* and *B1* and *B3* and *A5*. Note that this is different from the technique in Oracle BPEL Process Manager [12], where the appropriate tuning parameter `optSOAPShortcutBPEL` seems to apply to the total process.

B. Service Request Caching

Caching is a well-established strategy for reducing the number of requests made between components and for decreasing response times. A prominent example is HTTP caching [13].

The response time of Web service calls and the engine internal processing can be also improved through appropriate caching techniques. The SWoM implements, among other caching strategies, a very attractive caching strategy: service request caching via a client side cache, the *Service Request Cache* (SRC). This cache stores response messages from Web service calls and returns the cached result if the Web service is called again. More precisely, if an instance of an activity called a Web service the response of the partner is stored in the SRC. If another instance of the same activity is executed it retrieves the response not from the partner but from the SRC, i. e., the partner is not called again.

The cached response is also available for activity instances that base on the same activity model but belong to different process instances. Assume for instance that caching is activated for the `invoke` activity `A9` in our example process model `A` depicted in Fig. 1. Only for the first activity instance *A9'* of activity `A9` a Web service call to an instance of activity model `D1` is carried out. All requests from further instances of activity `A9` that belong to other process instances *A'*, *A"* etc. are served from the SRC.

Service request caching can also reduce the number of process instances that are created if the partner Web service is a process that is instantiated by the calling process. The reason is the implicit process instance creation in BPEL

[3]http://axis.apache.org/

where an instance of a process model is created if an instance creating `receive` such as the activity `D1` of process model `D` receives a message. Hence, if the response of an instance of `D1` is cached, no further instance of process model `D` is created as no message is sent any more to activity `D1`.

When the Navigator executes the `invoke` activity, it calls the Service Invoker to perform the synchronous SOAP call of the partner Web service. The Service Invoker in turn calls its *Service Request Cache Manager* (SRCM) with the request. The SRCM checks if the request is flagged cachable. This and the invalidation time of the response message belonging to a cachable request is described in the deployment descriptor for each `invoke` activity model. If the request is marked as cachable, the SRCM checks if a request message with the same payload was already cached before and if the invalidation time of the request has not expired yet. The cache is a simple in-memory data store that persists the request messages and the corresponding response messages. In case there is a match, the cached response message belonging to the request is returned to the Navigator by the Service Invoker. In case there is no match or if the invalidation time has expired, the Service Invoker performs a Web service call. The response from the Web service is passed to the Navigator and also to the SRCM which stores the request and the returned response in the SRC.

Service request caching has been proposed among others by many authors. Schmidt et. al [14] propose such a cache for the enterprise service bus (ESB) [15]. An appropriate cache mediation pattern for service invocation is presented in [16]. Xue [17] shows how the dynamic cache of IBM WebSphere can be exploited for service request caching in IBM Process Server.

The implementation the SWoM has taken provides several advantages: First, the information is managed in the engine's internal format so that no transformation of the responses external format into the engine internal one is required; second, the implementation does not rely on any proprietary application server functions, and third by attaching the request caching information to the appropriate `invoke` activity provides for better granularity and to cater more precisely to the requirements of the business process.

V. Inline Execution

In contrast to the optimization techniques described before, here, we present a technique that is exploited during deployment time, i. e., before the choreography is deployed to the engine: all process models of a choreography that are executed on the same engine are consolidated (merged) into one process model. This provides two performance advantages: (i) no message exchanges are required anymore and (ii) the number of process instances is reduced which saves resources required for process instance creation and execution.

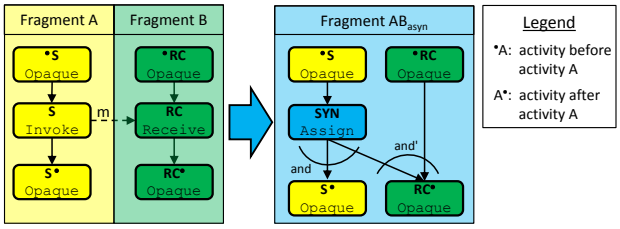

Figure 3. Consolidation of Asynchronous Interactions

The process consolidation we introduced in [18] merges two or more complementing process models that are members of the same choreography into a new single process model. The process models of the choreography in Fig. 1 are complementing each other as each process model contributes to the business objective to be achieved by the choreography. The basic idea of process consolidation is that the communication activities and their associated message links imply certain control flow relations between the activities of the interacting process models. Hence, the communicating activities such as `invoke` or `receive` are used as merge points. This means that these implicit control flow relations are "materialized" to explicit control flow relations between the activities. The aim of the consolidation is to keep the control flow relations between the different activities of the merged process models.

In a first step, each process to be merged is put into the consolidated process as a child of a `flow` activity. Thereby, it is ensured that each variable unique in each process model is still unique in the consolidated process model. Then merge operations are applied in the consolidated process model to materialize the control flow links from the message links.

An asynchronous `invoke` activity creates a message from a variable and sends this message. The message is received by a `receive` activity and stored in the designated variable. The merge operation replaces the `invoke` activity by an `assign` activity copying the value from the variable used by the `invoke` activity to the variable used by the `receive` activity (Fig. 3). The `receive` activity is deleted and the control links are modified to go directly from the predecessor to the successor of the `receive` activity. An additional control link is added from the `assign` activity to the successor of the `receive` activity. This ensures that the `assign` is executed when the former `invoke` would happen and that the successor of the former `receive` is executed after both the former `invoke` would have been finished *and* the predecessors of the former `receive` activity would have been finished.

In the synchronous case, the `invoke` activity also creates a message from an input variable and sends it to a `receive` activity. In contrast to the asynchronous case, the `invoke` activity does not complete. It has to wait until the `reply` activity created the reply message from a variable and sent it back to the `invoke` activity where it is stored in an output variable. To emulate this behavior, *two* `assign` activities are added to the consolidated process model. One for copying

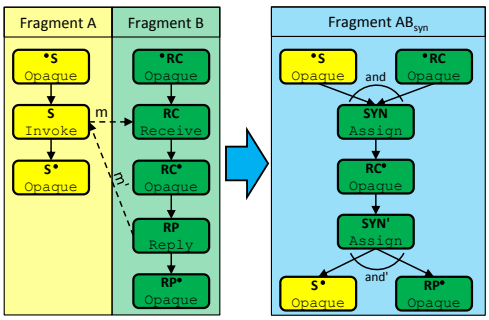

Figure 4. Consolidation of Synchronous Interactions

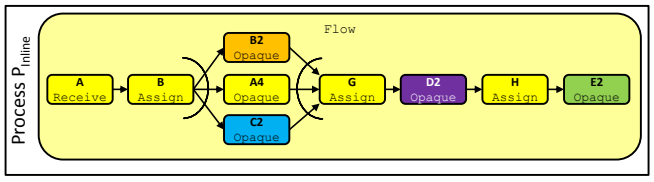

Figure 5. Consolidated Process Model P_{Inline} of Example Choreography

the input variable to the respective variable written by the `receive` before and one for copying the content of the variable where the reply message was created from to the former output variable of the `invoke` activity. Control links are modified accordingly. The consolidation for synchronous `invoke` activities is depicted in Fig. 4.

Fig. 5 shows the process model P_{Inline} that results from the consolidation of all process models of the example choreography depicted in Fig. 1. The activities *A4*, *B2* and *C2* in the original choreography can be performed simultaneously; this is also possible in P_{Inline}. Additionally, P_{Inline} ensures that activity *D2* is always performed before *E2*. Due to the synchronous communication between process model *A* and *D* the activity *D2* was always executed before activity *E2*. As an `assign` activity can perform multiple assignments, several consecutive `assign` activities that result from the synchronous and asynchronous consolidation operations were replaced by one single `assign` activity

VI. Evaluation

In the following the impact of the performance optimization techniques is evaluated by measuring the CPU load and energy consumption generated by the SWoM when it executes the instances of deployed processes models. The CPU load is measured instead of the throughput of process instances per time unit, as we also want to determine the impact of the optimizations techniques on the energy consumption of the test server. If we measured the maximal throughput of process instances in the SWoM, we would not be able to make a statement about the energy consumption as it would be constantly high because of the high CPU load during the whole test run. However, if we are able to reduce the CPU load with the optimizations techniques, also the maximal throughput of process instances can be increased as the engine can handle a larger amount of process instance with a lower CPU load. Database I/O related performance measurements are neglected here as the operations required for message exchanges affect more the CPU than the databases. This is especially true for short running processes that are almost only performed in-memory.

A. Test Setup

The process models are deployed on a SWoM hosted on a 64 bit IBM WebSphere V 7.0 with 8 GB RAM that uses an IBM DB2 Enterprise Edition V 9.5 as database backend. The operating system used is Windows 7 64 bit Edition installed on a Intel Quad Core 3 GHz Processor with 8 GB memory Windows, WebSphere, and DB2 with Buildtime and Runtime database are installed on different disk drives.

The execution of a test run is started from a JAX-WS client application that performs Web service calls to the processes that have to be tested in order to instantiate them. The client is installed on a Windows 7 IBM Thinkpad T60p with 4 GB memory. The client machine is connected to the server hosting the SWoM via a 54 Mbit wireless network. The client application is controlled with a test drive file that defines the different test cases. For each test case the process model to be tested (called) is specified and the number of requests sent to that process model. Each request results in a new process instance that has to be executed by the SWoM. Moreover, it can be also defined how many parallel requests are simulated by the client application, i. e., how many process instances have to be executed simultaneously.

To measure the CPU load on the server, we use the Windows performance monitor that samples every second and is using a window of 480 seconds to determine the CPU load average. The energy consumption is determined with the energy measuring device Voltcraft Energy Logger 4000[4] that is attached to the power supply of the server and plugged into a socket. The device constantly records the effective power in Watt that is load by the server per minute.

B. Test Procedure

To perform a test case, the test client sends constantly 9.92 request per second to the SWoM. Each request creates an instance of process model *A* of our example choreography and thus implicitly also one instance of each of the process models *B* to *D*. Hence, a test client request results in 5 process instances. When the inline optimization is tested, only the consolidated process model P_{Inline} is instantiated, thus only one process instance is created per test client request.

One test case is executed for 40 minutes as preliminary tests have shown that the machine becomes stable after approximately 25 minutes. It can be assumed that garbage collection is stabilized and all caches are in a fairly stable

[4]http://www.conrad.com/VOLTCRAFT-ENERGY-LOGGER-4000-ENERGY.Unit.htm?websale7=conrad-int&pi=125335

Table I
PERFORMANCE AND ENERGY CONSUMPTION TEST RESULTS

Test Runs	Average CPU Load [%]	Performance Improvement [%]	Average Total Energy Consumption [Watt]	Average Engine Energy Consumption [Watt]	Energy Consumption Improvement [%]
Idle	1	-	115	-	-
Full Load	97	-	185	70	-
No optimization	64	*reference*	175	60	*reference*
Intra-Engine Transport	18	72	144	29	52
Inline Execution	13	80	138	23	61
Service Request Caching	58	9	166	51	15

state. The readings of the CPU load and the energy consumption have been selected at the end of the test. We use two readings to smooth out any possible variations. If no major differences have been found between the two readings, so the results can be assumed to be sound and solid.

The results of the test runs are shown in Table I. The different rows denote the different test runs. The first row specifies the CPU load and the energy consumption for an idle engine that does not execute any process instances. In the second row measurements are specified for a SWoM that is fully loaded (4,380 process instances per minute). The third row contains the CPU load and the energy consumption for a non-optimized SWoM that receives 9.92 requests per second from the test client and thus executes 2976 process instances per minute (9.92 request per second $\cdot$ 60 s $\cdot$ 5 instances per request). The CPU load of 64% and the energy consumption of 176 Watts serve as reference values for the performance and energy improvements resulting from the applied optimizations techniques.

Apparently, lower CPU load and energy consumption values are better as this indicates that the SWoM can handle the test client requests more efficiently due to the applied performance technique. One column specifies the total consumption of the test server while the other one uses the energy consumption measured in the idle mode as offset, i. e., it only shows the energy consumed by the engine. The column "Energy Consumption Improvement [%]" always refers to the average engine energy consumption for the non-optimized execution.

Intra-engine transport reduces the CPU load to perform the 2976 process instances per minute by 72% and the energy consumption is reduced by 52% as no SOAP messages have to be transferred between the processes instances.

The inline execution of the choreography is the most powerful optimization technique, even though the CPU load is 5% and the energy consumption is just 9% lower compared to intra-engine transport. The difference results from the reduced number of process instances created per test client request and also from the reduced number of activities executed in P_{Inline} compared to the original choreography.

The testing of the service request caching was carried out with a slightly modified version of process model *A* that invokes the synchronous Web Service always with the same request in order to guarantee cache hits. We conducted the test with a varied the number of cache expiry time-outs. A saving of 9% was observed. The energy consumption is 15% lower. With the sheer number of requests that are carried out, varying the cache expiry time has no impact on the results. If, for example, a cache expiry rate of 1 minute is chosen, then only one out of 554 (60·9.92) invocations results in a cache miss. Compared to the other optimization techniques, the improvements are significantly lower as service request caching can be only activated for the synchronous `invoke` activity *A9*. We also evaluated service request caching with another process that was used by Bianculli et al. [19] for benchmarking different workflow engines. This process contains 5 sequential synchronous `invoke` activities that call simple Web service mocks that do nothing except returning default output data. There, we could measure a throughput improvement of 105%.

VII. RELATED WORK

The number of benchmarks for workflow engines is fairly low. LabFlow-1 [20] is the first benchmark in this field and studies the impact of databases on the performance of workflow engines. However, the authors argue that this is more a database benchmark than a workflow benchmark. We are not able to make a statement about how this benchmark would perform with the SWoM as the structure of the workflows used in LabFlow-1 was not provided by authors. The benchmark by Weikum et al. [21] compares the performance of different commercial workflow engines. The benchmark measured the throughput of the engines and also studied the impact of database accesses on the engine performance. In this benchmark the maximum measured throughput was 400 process instances per hour on a SUN Sparc10 system. Bianculli et al. [19] developed a benchmark to determine how different workflow engines, such as Apache ODE and ActiveVOS, handle structured activities such as `sequence` and `flow`. We used this benchmark as second benchmark for evaluating the service request cache performance improvements.

Information about performance optimization techniques in workflow engines is also very limited. Chapter 7 in the Oracle Application Server Performance Guide [12] lists a a parameter that controls the bypassing of SOAP processing. As pointed out, it seems to apply only to the process level and not to the individual activities within the process model. Furthermore,

the actual implementation has not been disclosed nor are any performance comparison figures available. Regarding service request caching in the context of service invocation Schmidt et. al [14] propose such a cache for the enterprise service bus [15]. An appropriate cache mediation pattern for service invocation is presented in [16]. Xue [17] shows how the dynamic cache of IBM WebSphere can be exploited for service request caching in IBM Process Server.

FastSOA [22] is an architecture and software coding practice centred around XML data serialization and Web services in general. It does not address performance optimizations in BPEL engines as we do.

VIII. Conclusion and Outlook

In this paper we proposed optimization techniques for interacting business processes that are part of a choreography: Intra-engine transport, service request caching and inline execution. With *intra-engine transport*, processes that are deployed on same engine do not use the SOAP/HTTP stack to exchange messages. Instead, the communication, message serialization and deserialization overhead is avoided by passing the information directly in its engine internal format between the components of the engine via local Java calls. *Service request caching* reduces the message exchanges between interacting processes or a partner Web services in general that are called synchronously by caching the results from previous invocations. *Inline Execution* merges all interacting process models that have to be executed on the same engine into one single process model. This avoids message exchanges and the number of activities performed as the communication activities for inter process communication are not needed any more. Additionally, the number of process instances is reduced.

To compare the performance gains resulting from the three optimization techniques we implemented the techniques in a workflow engine prototype and conducted CPU load and energy consumption measurements. Apparently, a higher CPU load results in more energy consumption. In our evaluation it turned out that inline execution is the most effective optimization technique followed by the intra-engine transport that generates slightly more CPU load. Service request caching had the lowest effect as our example choreography contains just one synchronous call. For other choreographies that contain more synchronous than asynchronous calls, a significant improvement by factor two could be measured. In summary, our measurements have shown that message exchanges between interacting processes make up the largest share of the generated CPU load.

In future works, we plan to do perform measurements with long-running choreographies that run for hours or longer. There, the state of the process instances has to be persisted in the database resulting in a larger impact of the database I/O operations on the overall performance and on the energy consumption. Further, we want to measure the concrete influence of message correlation in asynchronous communication scenarios on the performance.

Acknowledgement

This work was partially funded by the BMWi projects Migrate! (01ME11055) and CloudCycle (01MD11023).

References

[1] OASIS, *Web Services Business Process Execution Language Version 2.0 – OASIS Standard*, 2007.

[2] Object Management Group (OMG), *Business Process Model and Notation (BPMN) Version 2.0*, 2011, OMG Document Number: formal/2011-01-03.

[3] F. Leymann, "BPEL vs. BPMN 2.0: Should You Care?" in *BPMN*, 2010.

[4] V. Gruhn and R. Laue, "Complexity metrics for business process models," in *BIS*, 2006, pp. 1–12.

[5] C. Peltz, "Web Services Orchestration and Choreography," *IEEE Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[6] The Web Services-Interoperability Organization, "Ws-i basic profile version 2.0," Sep 2010, http://www.ws-i.org/Profiles/BasicProfile-2.0.html.

[7] Council of the European Union, "Brussels european council, presidency conclusions," Mar 2007, 7224/1/07, Rev 1.

[8] G. Decker *et al.*, "Interacting services: From specification to execution," *DKE*, vol. 68, no. 10, pp. 946–972, Apr. 2009.

[9] F. Leymann and D. Roller, *Production workflow: concepts and techniques*. Prentice Hall PTR, 2000.

[10] D. Davis and M. P. Parashar, "Latency performance of soap implementations," in *CCGRID*, 2002.

[11] A. Ng, S. Chen, and P. Greenfield, "An Evaluation of Contemporary Commercial SOAP Implementations," in *AWSA*, 2004.

[12] *Oracle BPEL Process Manager Performance Tuning*, Oracle, 2012, http://docs.oracle.com/cd/B32110-01/core.1013/b28942/tuning-bpel.htm.

[13] R. Fielding *et al.*, *Hypertext Transfer Protocol – HTTP/1.1*, 1999, http://tools.ietf.org/html/rfc2616.

[14] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The Enterprise Service Bus: Making service-oriented architecture real," *IBM Systems Journal*, vol. 44(4), 2005.

[15] D. A. Chappel, *Enterprise Service Bus*. O'Reilly Media, 2004.

[16] F. Y. Ran *et al.*, "Message Oriented Middleware Cache Pattern – a Pattern in a SOA Environment," in *Fourth "Killer Examples" for Design Patterns and Objects First Workshop (OOPSLA 05*, 2005.

[17] Jun Xue, "Caching web services to improve the performance of business solutions in WebSphere Process Server," 2010.

[18] S. Wagner, O. Kopp, and F. Leymann, "Towards Choreography-based Process Distribution In The Cloud," in *CCIS*, 2011, Conference Paper.

[19] D. Bianculli, W. Binder, and M. L. Drago, "Automated performance assessment for service-oriented middleware: a case study on BPEL engines," in *WWW*, 2010.

[20] A. J. Bonner, A. Shrufi, and S. Rozen, "Labflow-1: A database benchmark for high-throughput workflow management," in *EDBT*, 1996.

[21] M. Gillmann, R. Mindermann, and G. Weikum, "Benchmarking and configuration of workflow management systems," in *CoopIS*, 2000, pp. 186–197.

[22] F. Cohen, *FastSOA: The Way to Use Native XML Technology to Achieve Service Oriented Architecture Governance, Scalability, and Performance*. Morgan Kaufman Publ Inc, 2007.