2013 39th Euromicro Conference Series on Software Engineering and Advanced Applications

Bringing Scientific Workflows to Amazon SWF

Matthias Janetschek, Simon Ostermann and Radu Prodan
Institute for Computer Science, University of Innsbruck, Austria
{matthias,simon,radu}@dps.uibk.ac.at

***Abstract*—In response to the ever-increasing needs of scientific applications for resources, Cloud computing emerged as an alternative on-demand and cost-effective resource provisioning approach. In this context, Cloud providers have recognised the importance of workflow applications to science and provide their own native solutions, such as the Amazon Simple Workflow Service (SWF). Nevertheless, an important downside of SWF is its incompatibility with existing workflow systems, and lack of means for reusing scientific legacy code. Similarly, existing workflow middlewares and applications require non-trivial extensions to take advantage of Cloud resources. We present in this paper a software engineering solution that allows the scientific workflow community access the Amazon Cloud through one single front-end converter, and propose a legacy wrapper service for executing legacy code using SWF. Empirical results using a real-world scientific workflow demonstrate that our automatically generated SWF application performs almost as fast as a native manually-optimised version, and outperforms other workflow middleware systems using the Amazon Cloud.**

***Keywords*—scientific workflows; cloud computing; Amazon SWF; legacy code; workflow converter**

I. Introduction

Today, scientific applications require an ever-increasing number of resources to deliver results for growing problem sizes in a reasonable amount of time. In the last 20 years, while the largest projects were able to afford expensive supercomputers, the smaller ones were forced to opt for cheaper resources such as commodity clusters or, more challenging to build, computational Grids. To program such large-scale distributed heterogeneous infrastructures, scientific workflows emerged as an attractive paradigm by allowing the programmers to focus on the composition of existing legacy code fragments to create larger and more powerful applications. Therefore numerous efforts have been spent on researching and developing integrated programming and computing environments [1] to support the workflow lifecycle and meet scientists' needs.

Nowadays, *Cloud computing* proposes an alternative such that resources are no longer owned by the application scientists, but leased from large specialised data centers on-demand and in a cost-effective fashion according to temporal needs. This separation frees research institutions from the permanent costs of over-provisioning, operation, maintenance and depreciation of resources. Nevertheless, existing workflow systems cannot senselessly take advantage of this new infrastructure without appropriate middleware support that often requires non-trivial extensions to the scheduling, enactment, resource management, and other runtime execution services. At the same time, existing Cloud providers such as Amazon recognised the importance of workflows to science and engineering and started to provide highly-tuned solutions integrated into their native platforms such as the *Amazon Simple Workflow Service (SWF)*. However, existing workflow systems [1] cannot immediately take advantage of this support because of different, incompatible languages, interfaces and communication protocols. Another downside of SWF is that it requires applications to be written in Java and to implement specific interfaces, which is problematic for scientific workflows based on the composition of legacy code fragments. Using SWF requires scientists to learn a new development and execution platform in addition to the one they regularly use.

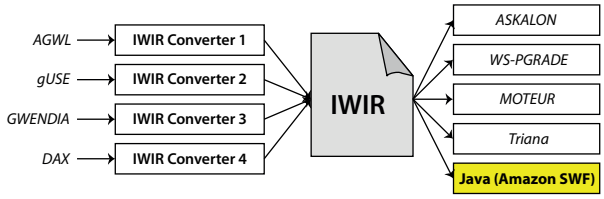


Fig. 1. SHIWA fine-grained interoperability.

To address this heterogeneity in workflow systems and underlying computing infrastructures, the SHIWA European project (http://www.shiwa-workflow.eu/) researched and developed the *Interoperable Workflow Intermediate Representation (IWIR)* [2] that enables fine-grained interoperability between workflow systems through transparent translation of workflows applications programmed in different languages. IWIR is a generic and system-neutral workflow representation able to sufficiently describe the large majority of existing workflow constructs. The common representation reduces the complexity of porting n workflow systems on m computing platforms from $O(m \cdot m)$ to $O(n+m)$. Additionally, it enables the integration of new workflow systems and new computing platforms with constant $O(1)$ complexity by implementing IWIR importers/exporters. This ensures not only interoperability across workflow systems, but also enables workflows to be executed on new external foreign (or non-native) computing infrastructures. IWIR provides additional tools and libraries to ease the development of language translators, and is currently supported by five major workflow systems: ASKALON (AGWL language) [3], Moteur (GWENDIA language) [4], WS-PGRADE (gUSE language) [5], Pegasus and [6] Triana (DAX representation) [7] (see Figure 1).

In this paper, we take advantage of IWIR and present a scalable software engineering solution that provides existing scientific workflows access to the Amazon Elastic Compute Cloud (EC2) infrastructure. By designing and implementing one single IWIR-to-SWF converter, we automatically allow all IWIR-compliant workflow systems to benefit from the SWF features and to access the EC2 infrastructure with native performance. We present a method for automatically converting a

978-0-7695-5091-6/13 $26.00 © 2013 IEEE
DOI 10.1109/SEAA.2013.13

scientific workflow specified in IWIR into Amazon SWF, and a supporting architecture for reusing and executing existing legacy code on EC2. We illustrate the integration and the advantages of our architecture with the help of a real-world scientific workflow originally programmed in the ASKALON integrated development and computing environment.

The paper is organised as follows. We discuss related work in Section II. Section III introduces the IWIR workflow model, followed by an introduction to Amazon SWF in Section IV. Section V introduces our pilot workflow application used for validation. Section VI describes the conversion process of an IWIR workflow into an Amazon SWF workflow. Section VII presents experimental results from porting our pilot application to SWF. Section VIII concludes the paper.

II. Related Work

Since the advent of Cloud computing, the scientific community showed increasing interest in bringing scientific workflows on this new infrastructure. This trend increased with the availability of commercial Clouds featuring nearly the same performance as traditional Grid parallel computers [8]. There exist two major approaches in this community effort: pure Cloud and hybrid combining Grid and Cloud infrastructure.

FutureGrid [9] provides a Cloud test-bed that allows scientists explore the features of Cloud computing and experiment without charging real costs, as commercial providers do. [10] shows a proof-of-concept astrophysics workflow called Montage using the Pegasus Grid workflow system adapted for Clouds. Similarly, [11] shows a meteorological workflow executed in combined Grid and Cloud infrastructures using the ASKALON environment. A hybrid approach for extending clusters with additional Cloud resources during peak usage for better throughput, transparent to the end-users is presented in [12]. [13] presents a similar approach using the Torque job manager. The work in [14] presents a workflow engine purposely developed for Clouds and extended Cloud federations. The Megha workflow system [15] provides a portal for submitting workflows to combined Grid and Cloud resources.

A drawback of all these efforts is that they provide custom non-interoperable solutions that isolate scientists on specific workflow system and Cloud infrastructures. In this paper, we show how the IWIR-based approach opens the Amazon EC2 infrastructure and its SWF workflow system to the scientific community through one single IWIR-to-SWF translator. The idea of a single intermediate language has been explored in other domains, for example by the UNiversal Computer Oriented Language (UNCOL) [9] proposed in 1958 by Melvin E. Conway as a solution for making compiler development economically viable. Following the UNCOL idea, the Architecture Neutral Distribution Format (ANDF) is a technology defined by the Open Software Foundation allowing common "shrink wrapped" binary programs be distributed for use on Unix systems running on different hardware platforms. Unfortunately, ANDF was never widely adopted either. IWIR is the first effort to investigate this idea on scientific workflows in distributed Grid and Cloud computing infrastructures.

III. IWIR Workflow Model

In IWIR, a *workflow application* is represented by a composite activity $A = (I, O, G)$ consisting of n input ports $I = \bigcup_{i=1}^{n} \{I_i\}$, m output ports $O = \bigcup_{i=1}^{m} \{O_i\}$, and a directed acyclic graph (DAG) $G = (A, D)$ consisting of k activities $A = \bigcup_{i=1}^{k} \{A_i\}$, interconnected through data flow dependencies: $D = \{(A_i, A_j, (O_{im}, I_{jn})) \,|\, (A_i, A_j) \in A \times A \wedge (O_{im}, I_{jn}) \in O_i \times I_j\}$, where (O_{im}, I_{jn}) represents a data transfer from the output port O_{im} of activity A_i to the input port I_{jn} of activity A_j. A data flow dependency between two activities implies a control flow precedence too. A pure control flow dependency between A_i and A_j has $D_{ij} = (A_i, A_j, \emptyset)$. We use $pred(A_i) = \{A_k \,|\, (A_k, A_i, (O_{km}, i_{in})) \in D \vee (A_k, A_i, \emptyset) \in D\}$ to denote the set of *predecessors* of activity A_i (i.e. activities to be completed before starting A_i).

There are two categories of activities in IWIR: atomic and composite. An *atomic activity*, represented by $A = (I, O, \emptyset)$, is characterised by an *activity type*, uniquely defined by a *name* and a *signature*. For example, activity names are `PrepareLM`, `LinearModel`, `PostProcessSingle` and `PostprocessFinal` for our pilot workflow introduced in Figure 2 and Section V. The signatures of `LinearModel` and `PostProcessSingle` are shown in lines 8 and 19 of Listing 1. A *composite activity*, represented by $A = (I, O, G)$, where $G \neq \emptyset$, can be of four kinds: conditional (`if`), sequential loop (`while`, `for`, `forEach`), parallel loop (`parallelFor`, `parallelForEach`), and sub-workflow (or DAG, introduced for modularity reasons).

IV. Amazon SWF Background

Amazon SWF provides a high-level method for implementing workflow applications and for coordinating their synchronous and asynchronous task executions on multiple systems, which can be cloud-based, on-premises, or both. SWF implements a work-stealing approach consisting of three parts: decider, task queues, and activity workers.

The *decider* implements the logic of the workflow. Unlike schedulers in scientific workflow systems, the decider only decides which activity to execute next based on the history of already executed tasks, and not where to execute it. However, one still has limited control by using several task queues where the decider puts the activities to be executed next.

The *task queues* hosted by Amazon are identified by their name and can be accessed via an HTTP API. There are two types of tasks. First, *decision tasks* are generated by Amazon SWF and executed by the decider every time a state change exists (e.g. start of workflow instance or activity task termination). The result of a decision task is usually a set of activity tasks that can be executed next. Second, *activity tasks* are executed by the activity workers and represent the individual pieces of work which comprise the workflow.

The *activity workers* execute the individual workflow activities. The decider and the activity workers actively listen to one or more task queues and, when a task is received, execute the corresponding code and report back the execution status to Amazon SWF. All input values of an activity are contained in the task request received from the task queue. Unlike traditional workflow systems, Amazon SWF provides no means to transfer files or prepare the execution environment. If an activity requires some input or produces some output files,

it has to transfer them by itself. For Amazon SWF, workflow activities are simply remote asynchronous procedure calls.

Developing a workflow with Amazon SWF requires the following steps: (1) develop a decider implementing the logical workflow coordination; (2) develop activity workers implementing the individual activities; (3) register the workflow at Amazon SWF; (4) start the decider and activity workers and let them listen to the SWF endpoint; (5) start the workflow.

AWS Flow Framework allows the development of Amazon SWF workflows through the AWS SDK for Java by specifying its coordination steps as a sequential Java program, where the workflow activities are represented as function calls. Functions representing activities and functions used to handle or manipulate data produced or consumed by activities need to have the `@Asynchronous` annotation (called in the following *asynchronous functions*) and their input arguments and return values need to be of type `Promise`. A `Promise` object acts as a handle to the actual data that will be available as soon as the corresponding asynchronous function has been executed. When used as input argument, a `Promise` object can also be used to represent data dependencies between several asynchronous functions. An asynchronous function, having a `Promise` object produced by another asynchronous function as input, will only be executed when the actual data referenced by the `Promise` object is available.

Amazon SWF executes a workflow application through repeated invocations of the decider program, which is executed every time a state change occurs (signaled via a *decider task*), and a history of all decider executions is recorded. To intercept all calls to asynchronous functions in the decider program, AWS uses AspectJ, a Java implementation of aspect-oriented programming [16]. An asynchronous function is instantiated only once during the entire workflow execution, and its return value is saved into the execution history. In every subsequent decider execution, the same function is not re-executed, but its result extracted from the execution history and returned as `Promise` object. An asynchronous function that has not been executed yet, is put into a queue and a `Promise` object with no actual data is returned. This data will be instantiated as soon as the corresponding asynchronous function has produced it. Before the decider finishes its execution, it examines all asynchronous functions in the queue, executes those whose dependencies are satisfied, and records their results in the execution history. The workflow execution finishes if there are no more non-executed asynchronous functions.

V. RainCloud Workflow

We introduce in this section the RainCloud workflow used in this paper for illustrating and validating our approach. Raincloud is a meteorological workflow for investigating and simulating precipitations in mountainous regions using a simple numerical linear model of orographic precipitations [17]. The workflow has been developed in the ASKALON environment by the Institute of Meteorology of the University of Innsbruck to analyse certain meteorological phenomena by extending the linear model theory. The workflow is also is used by the Tyrolean avalanche service ("Tiroler Lawinenwarndienst") for their daily issued avalanche bulletin.

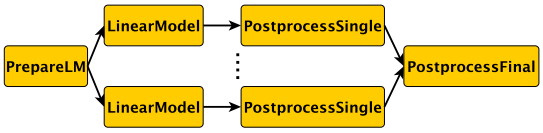


Fig. 2. Simplified view of the RainCloud workflow.

Figure 2 shows a simplified architecture of the RainCloud workflow. The first activity `PrepareLM` prepares and partitions the data for the linear model. Each partition is then processed in a parallel loop iteration by a pipeline of two activities: `LinearModel` and `PostprocessSingle`, the last one being optional. The number of parallel loop iterations can be configured by setting the appropriate input parameter. The last activity collects the output data and produces the final result. Listing 1 shows the specification of the `parallelForEach` loop in IWIR. Inside this loop, we first have the atomic activity `linearModel` (line 8), followed by an `if`-construct (line 14) containing the atomic activity `postProcessSingle` (line 19).

Listing 1. RainCloud's `parallelForEach` loop in IWIR.

```
1  ...
2  <parallelForEach name="ParallelForEach_1">
3   <inputPorts><inputPort name="isPPS" type="boolean"/>
4    <loopElements><
5      loopElement name="PLMTars" type="collection/file"/>
6    </loopElements></inputPorts>
7   <body>
8    <task name="linearModel" tasktype="linearModel">
9     <inputPorts><inputPort name="PLMTar" type="file"/>
10    </inputPorts>
11    <outputPorts><outputPort name="LMTar" type="file"/>
12    <outputPort name="outfile" type="file"/></outputPorts>
13   </task>
14   <if name="DecisionNode_1">
15    <inputPorts><inputPort name="LMTar" type="file"/>
16    <inputPort name="isPPS" type="boolean"/></inputPorts>
17    <condition>isPPS = true</condition>
18    <then>
19     <task name="postProcessSingle" tasktype="
           postProcessSingle">
20      <inputPorts><inputPort name="LMTar" type="file"/>
21      </inputPorts>
22      <outputPorts><outputPort name="PPSTar" type="file"/>
23      </outputPorts>
24     </task>
25    </then>
26    <outputPorts><outputPort name="PPSlistTars" type="file"/>
          </outputPorts>
27    <links>
28     <link from="DecisionNode_1/LMTar" to="postProcessSingle/
           LMTar"/>
29      ...
30    </links>
31   </if>
32  </body>
33  <outputPorts>
34    <outputPort name="PPSlistTars" type="collection/file"/>
35    <outputPort name="outfiles" type="collection/file"/>
36  </outputPorts>
37  <links>
38   <link from="ParallelForEach_1/PLMTars" to="linearModel/
         PLMTar"/>
39    ...
40  </links>
41 </parallelForEach>
42 ...
```

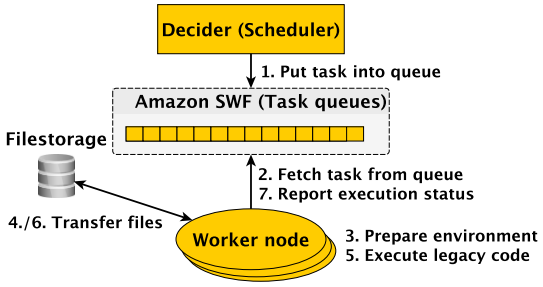


Fig. 3. Architecture of a generated Amazon SWF workflow.

VI. IWIR-to-SWF Conversion

Figure 3 shows the architecture of our solution, consisting of four parts: the decider, Amazon SWF, the legacy code execution service on each worker node, and the file storage. With Amazon SWF, the decider and workflow activities are individual Java programs, purposely designed for Amazon SWF. The goal of this paper is to present a method for translating scientific workflows from the interoperable IWIR representation to Amazon SWF with as little effort for the programmer as possible. While the abstract workflow coordination can be automatically translated into an SWF decider Java program, there is no practical way to automatically convert the legacy code implementing the concrete workflow activities into a SWF-compatible Java program. To still achieve this goal with minimal programmer involvement, we implemented an execution service that interfaces with Amazon SWF and acts as a Java wrapper for existing legacy code.

The only requirement imposed by Amazon SWF on the worker nodes is an outgoing HTTP connection to Amazon SWF. This makes Amazon SWF easy to setup with no firewall reconfiguration. Technically, direct file transfers between worker nodes are possible, but this requires a corresponding service running on the worker nodes and the firewall to be reconfigured accordingly. As we did not want to loose the advantage of an easy setup, we decided to use an intermediate file storage for the file transfers, so that there is no need for incoming connections on the worker nodes. Currently, we support only Amazon S3 as an intermediate file storage, but other file storage technologies can be easily added too.

A. Decider generation

As presented in Section III, an IWIR workflow is constructed from a top-level DAG activity which explicitly describes the data flow between its activities. Control flow constructs such as loops and conditionals are represented by composite activities. To convert an IWIR workflow into an Amazon SWF decider, we have to transform the data flow-driven IWIR DAGs and the semantics of the composite activity types into a control flow-driven Java program. Moreover, we also have to take care that the concepts of the AWS Flow Framework, namely asynchronous functions and semantics of the returned `Promise` objects, are correctly applied.

The basic principle of the conversion is that every atomic or composite activity is represented by its own *activity function* in the Java program. Algorithm 1 shows the generation process

Algorithm 1 SWF decider generation algorithm.

Input: Scientific workflow: $A = (I, O, G)$
Output: SWF decider (Java program)
1: **function** GenDecider($A = (I, O, G)$)
2: $Queue \leftarrow \emptyset$
3: GenWfStart($A, Queue$)
4: **while** Queue $\neq \emptyset$ **do**
5: $A \leftarrow$ Pop($Queue$)
6: GenActivityFunction($A, Queue$)
7: **end while**
8: **end function**
9: **function** GenWfStart($A, Queue$)
Input: $A = (I, O, G)$
10: GenWfStartProlog(I, O)
11: GenActivityFuntionCall($A, Queue$)
12: Put($Queue, A$)
13: GenWfStartEpilog(O)
14: **end function**

of the decider. The first step is the generation of a function representing the start of the workflow (line 3). The signature of this function represents the input and output ports of the workflow (line 10). In the function body, the top-level activity function is called with the appropriate input arguments (line 11). Afterwards, the results of the top-level activity function are presented to the user in an appropriate way (line 13). Every activity encountered during the conversion process with no activity function created yet is put into a queue (e.g. in line 12). After the workflow entry function has been generated, the algorithm iterates through this queue (line 4) and generates an activity function for each queue element (line 6).

B. Activity function generation

Algorithm 2 shows the generation of an activity function representing a workflow activity. The function signature of an activity function corresponds to the input and output ports, while the function body implements its semantic behaviour, including any associated DAG. For atomic activities, we only have to generate the function signature with the `Activity` annotation and an empty function body (lines 9–10). The AWS Flow Framework will then automatically generate function stubs, which allow us to communicate with the SWF task queue. For composite activities, we need to additionally generate, besides the function signature, a function body implementing the activity behaviour (lines 4-6). In the following, we describe in detail how the activity functions are generated. To facilitate understanding, we divided the code generation of the composite activity function bodies in three logical sections: (1) activity semantics, (2) DAG control flow, and (3) DAG data flow. However, these steps are not distinct, but interleaved with each other (e.g. the function call in line 5 generates not only the control flow but also the data flow).

1) Function signature: The first step in generating an activity function is the function signature. The arguments of the activity function represent the input ports and the return value the output ports of the associated workflow activity. However, this representation has some inadequateness. In a workflow representation, the input and output ports of an activity are usually identified by their names, and the number of output ports is not limited. In contrast, the arguments of a Java function are identified by their order and the return argument is restricted to one. Moreover, returning values by call by reference does not work in an SWF program because the activity functions are executed asynchronously

Algorithm 2 Activity function generation algorithm.

Input: Workflow activity: $A = (I, O, G)$
Output: Activity function (in Java)
1: **function** GENACTIVITYFUNCTION(A, *Queue*)
Input: $A = (I, O, G)$
2: **if** $G \neq \emptyset$ **then** ▷ Composite activity
3: $N \leftarrow$ GETACTIVITYNAME(A)
4: GENFUNCTIONPROLOG(N, A)
5: GENDAGCONTROLFLOW(G, *Queue*)
6: GENFUNCTIONEPILOG(A)
7: **else** ▷ Atomic activity
8: $N \leftarrow$ GETACTIVITYTYPENAME(N, I, O)
9: GENFUNCTIONSIGNATURE(N, I, O)
10: GENEMPTYFUNCTIONBODY()
11: **end if**
12: **end function**

from the rest of the program (see Section IV). In practice, the first inadequateness can be neglected when generating the decider by consistently maintaining the same parameter order. However, this may pose a problem for the legacy wrapper service, as changes in the order of the input arguments cannot be automatically distributed to this service. To address this problem, we implemented a wrapper class for the input arguments of atomic activity functions with a field for the name of the input port and a field for the actual value. The legacy wrapper service can then assign the input values to the correct input port by looking at the name field. To address the second inadequateness, we implemented another wrapper class that stores several output values into an array which is returned by the activity functions.

For example, Listing 2 shows a function signature representing the atomic activity `linearModel` of RainCloud. Because the activity has more than one output port, the corresponding function returns an object of type `PortWrapperArray` encapsulating the output values. All input values have the type `PortWrapper` because the function represents an atomic activity and, therefore, needs to interface with the legacy wrapper service. The AWS Flow framework automatically generates a stub function for interfacing with the task queues declared as `asynchronous` and returning a `Promise` object.

Listing 2. Function signature of the atomic `linearModel` activity.

```
1 @Activity(name="RainCloudActivities.linearModel")
2 public PortWrapperArray linearModel ( PortWrapper PLMTar )
```

Listing 3 contains another example of a function signature representing the composite activity `ParallelForEach_1`.

Listing 3. Function signature of the `ParallelForEach_1` activity.

```
1 @Asynchronous
2 private Promise parallelForEach_1 ( Promise<Boolean> isPPS,
      Promise<String[]> PLMTars );
```

2) Activity semantics: The next step is the generation of code that implements the semantics of the three types of composite activities: container, conditional, and loop. *Container activities* only contain other activities without additional semantics. *Conditional activities* consist of an `if-else` construct and separate activity function control flows for the two branches. The conditional expression may contain input port values that can be easily referenced by specifying the appropriate function argument. *Loop activities* are the hardest to implement because of the several IWIR loop flavours: `while`, `for`, `forEach`, `parallelFor` and `parallelForEach`. We exploited the asynchronous function invocation feature of SWF to implement parallel loops as simple sequential loops in the decider program. Because activity functions are executed asynchronously, the decider does not wait for an activity function to finish before starting the next loop iteration. To force sequential execution of activity functions inside a non-parallel loop, we have to introduce artificial dependencies between activity functions called in different iterations using `Promise`-objects.

Listing 4 shows an example of a function body representing the composite activity `ParallelForEach_1` of RainCloud. The number of loop iterations is first calculated in line 3. Lines 5–10 represent the actual `for` loop, while lines 12–13 deal with the construction of the return value.

Listing 4. `ParallelForEach_1` activity semantics.

```
1  private Promise parallelForEach_1 (...) {
2    // Get number of elements.
3    int maxIter = PLMTars.get().length;
4    // Iterate over the given array.
5    for (int i = 0; i < maxIter; i++)  {
6      // Get current element
7      Promise<String> p = Promise.asPromise(PLMTars.get()[i];
8      // Activity function control flow goes here
9      . . .
10   }
11   // Build return value.
12   Promise[] retval = new Promise[2];
13   return Promise.asPromise( retval );
14 }
```

3) DAG control flow: The workflow activities of a given DAG are sorted according to their topological order that preserves the original data flow. In the topological order, a workflow activity can only be executed after all its predecessors have been completed and produced the required input data. As a workflow may consist of several DAGs, we calculate the topological order for each DAG independently.

For example, RainCloud's `ParallelForEach_1` loop calls the activity `linearModel` whose results are fed into the activity `PostProcessSingle` depending on the value of the input parameter `isPPS`. Listing 5 shows the equivalent Java activity with calls to the contained activity functions in lines 6 and 8. The `if` statement, which determines whether `PostProcessSingle` should be executed, is represented by the `decisionNode_1` function in line 8, with the missing parameter added in the data flow step (Listing 6, line 14).

Listing 5. Control flow inside `ParallelForEach_1` activity.

```
1  private Promise parallelForEach_1 (...) {
2    int maxIter = PLMTars.get().length;
3    for (int i = 0; i < maxIter; i++) {
4      Promise<String> currEl = Promise.asPromise(PLMTars.get()
           [i];
5      // Call to atomic activity "LinearModel"
6      activityClient.linearModel(currEl);
7      // Call to composite if-activity
8      decisionNode_1(..., isPPS);
9    }
10   Promise[] retval = new Promise[2];
11   return Promise.asPromise( retval );
12 }
```

4) Data flow: The last step in generating the body of an activity function is to introduce variables that model the data flow between the enclosed activity functions. To ease the variable handling, we use the *single static assignment*

technique employed in compiler construction, which requires every variable be written once and not reused afterwards. Every value returned by an activity function is assigned to its own unique variable and passed as input to each activity function with a connected input port. The main idea is that the implementation of an activity function does not need to know how the preceding activity functions produced and stored their output values. This is also reflected in the activity function signatures (see Section VI-B1) which only consists of the input arguments from the original workflow specification. Activity functions returning more than one value return a wrapper object (see Section VI-B). The individual values contained in this wrapper object need to be extracted before they are fed to a subsequent activity function. Unfortunately, `Promise` objects can only be accessed inside asynchronous functions, otherwise an exception will be thrown. To address this problem, we implemented several asynchronous helper functions for data manipulation and conversion.

Listing 6 presents the data flow of the activity function representing the composite activity `ParallelForEach_1`. First, an array for holding the results of each loop iteration is created for each activity in lines 4 and 6. The activities' output ports are directly connected to a corresponding output port of the surrounding composite activity. Then, the return value of each activity function is stored in its own variable in lines 10 and 14. Since the activity `linearModel` returns a wrapper object, we have to convert it (line 12) before using the actual return values (lines 14 and 16). At the end of each loop iteration, the values produced in the iteration are stored into the corresponding variables (lines 16 and 18). At the end of the function body, we construct the return object and convert the variables into a more suitable form (lines 22 and 24).

Listing 6. Data flow within the `ParallelForEach_1` composite activity.

```
1  private Promise parallelForEach_1 (...) {
2    int maxIter = PLMTars.get().length;
3    // Holds output values of linearModel activity
4    Promise[] out1 = new Promise[maxIter];
5    // Holds output values of decisionNode_1 activity
6    Promise[] out2 = new Promise[maxIter];
7    for (int i = 0; i < maxIter; i++) {
8      Promise<String> p = Promise.asPromise(PLMTars.get()[i];
9      // Save linearModel return value of in lmo1
10     Promise<PortWrapperArray> lmo1 = activityClient.
           linearModel(p);
11     // Convert lmo1 in a format for further processing
12     Promise[] lmo2 = Utils.convertPWA2Pa(lmo1, 2);
13     // Input first value stored in lmo2; save return value
           into dno1
14     Promise dno1 = decisionNode_1(lmo2[0], isPPS);
15     // Store linearModel return value in a collection
16     out1[i] = lmo2[1];
17     // Store if return value in a collection
18     out2[i] = dno1;
19   }
20   Promise[] retval = new Promise[2];
21   // Convert collection to a suitable return format
22   retval[0] = Utils.convertAoP(out1);
23   // Convert collection to a suitable return format
24   retval[1] = Utils.convertAoP(out2);
25   // Return the output values
26   return Promise.asPromise(retval);
27 }
```

VII. Experiments

The goal of our experiments is to compare the performance of the RainCloud workflow in three configurations: automatically-generated SWF workflow (using the technique described in Section VI), manually-optimised SWF workflow, and original ASKALON version executed using the ASKALON middleware. To be able to interface with the EC2 infrastructure, we pragmatically extended the ASKALON middleware services such as security with Amazon credentials, information service with virtual machine image manipulation, and enactment engine with SSH-based job submission.

A. Setup

We run the experiments on 16 Amazon instances of type `m1.medium`. For the SWF workflow, we used S3 as intermediate file storage. We executed the SWF decider and the ASKALON scheduler on a dedicated host with an Intel i7-2600K quad-core processor running at 3.4 GHz and 8 GB of memory, outside of Amazon EC2. For ASKALON we used a just-in-time scheduler which maps the next ready activities on the machines delivering the earliest completion time, because it mostly resembles the SWF operation. We executed the RainCloud workflow in two scenarios: *non-congested* with 16 parallel loop iterations and two problem sizes (small and large) and *congested* with 64 parallel loop iterations and a small problem size. The small problem size corresponds to a 18×18 simulation grid and the large one to a 36×36 grid. For each scenario and workflow version, we calculated the two metrics: *average total execution* time and *cumulative execution* time of all workflow activities plus the scheduling time. To get an understanding on the amount of overhead present in a workflow execution, we further split its cumulative execution time into *processing* time (performing actual computation), *scheduling* time, *waiting* time (in an engine internal queue) due to insufficient free resources, *queuing* time due to middleware and external load latencies, and *file transfer* time.

B. Results

Figure 4 shows the total execution times for the three workflow versions with 16 parallel iterations and small and large problem sizes in the non-congested scenario. The manually-written SWF workflow is only marginally faster than the automatically generated version. We expected this result because the two versions only differ in the implementation of the decider, whose overhead is negligible compared to the total workflow execution time. Surprisingly, the ASKALON version suffers from significantly higher execution times due to the much higher overhead for transferring files between the worker nodes, as shown in Figure 5(a). We found out that this overhead is caused by the Java CoG Kit [18] employed by ASKALON as a black-box library for interfacing with Grids (through Globus plugin) and Clouds (through SSH plugin), which uses an ASKALON middleware machine outside Amazon EC2

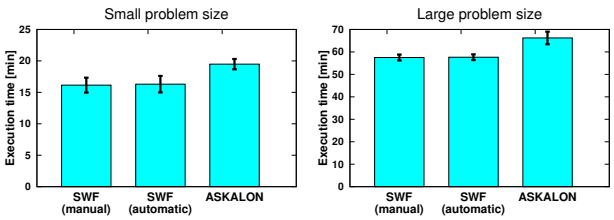
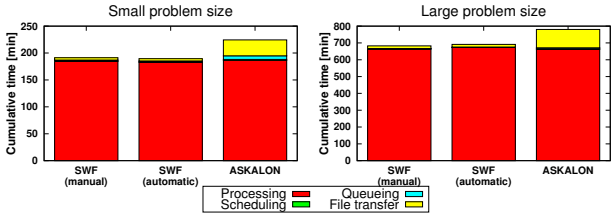
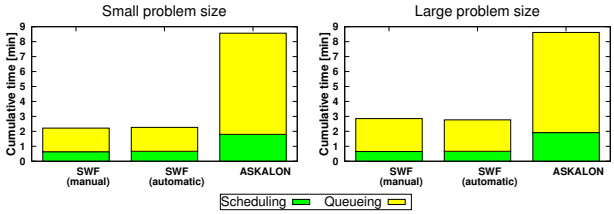


Fig. 4. RainCloud execution time in non-congested scenario.



(a) Cumulative times.



(b) Cumulative overheads except file transfer.

Fig. 5. RainCloud cumulative times in non-congested scenario.

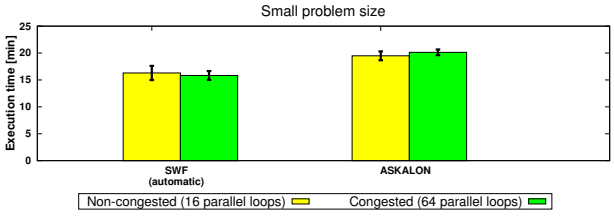


Fig. 6. RainCloud execution time with 16 and 64 parallel loops.

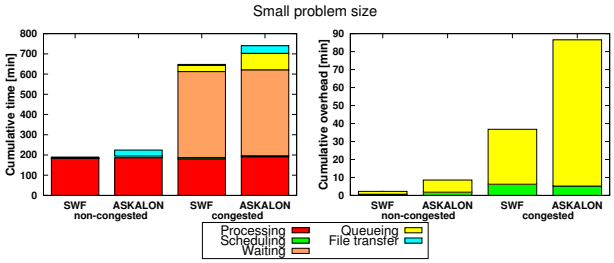


Fig. 7. Cumulative RainCloud execution times with 16 and 64 parallel loops.

as an intermediary for transferring files between two remote machines. In the following, we disregard the file transfer times to make the experiments more comparable.

The other reasons for ASKALON's performance losses are the scheduling and queuing overheads shown in Figure 5(b). The scheduling overhead in ASKALON is approximately three times higher than in SWF because it is tuned for highly heterogeneous and distributed Grid infrastructures, as opposed to Clouds that tend to be more homogeneous and located within one data centre. Because of this, the ASKALON Grid scheduler needs to interact with a resource manager for discovering the available shared resources which is not a requirement in static Clouds owned by a single organisation. Moreover, the ASKALON scheduler also needs to evaluate the external load generated by scientists sharing a specific Grid resource, not required for dedicated Cloud resources. Finally, the ASKALON scheduler is also responsible for preparing the remote execution environments (and directories) through multiple SSH connections, not required for SWF that delegates the setup of the environment to the locally running legacy wrapper service.

Also, the workflow activities wait three times longer in the queue of ASKALON compared to SWF. The average overhead per workflow activity without file transfer is approximately $4-5$ seconds for SWF and around 15 seconds for ASKALON, which is comparable for scientific workflows with long running activities. The queuing time is larger for ASKALON than for SWF because of the higher middleware stack required by ASKALON for supporting a broader range of heterogeneous infrastructures (i.e. clusters, Grids, Clouds), as opposed to SWF tuned for running in the native EC2 infrastructure only. In addition, ASKALON actively pushes workflow activities to be executed onto the worker nodes which introduces higher overhead than the pull approach used by Amazon SWF where the worker nodes actively fetch tasks from a task queue.

Figure 6 shows the total execution times in the congested scenario. The SWF version performs slightly better for 64 parallel loop iterations than for 16, however this improvement due to load imbalance on the 16 iteration parallel loop and coarse-grain activity sizes is still within the standard deviation. Using 64 iterations produces a finer-grained parallelisation and smaller activity sizes that enables a better schedule with smaller load imbalance overhead. The ASKALON version with 64 parallel iterations performs worse than with 16, but this degradation is again within the standard deviation.

Figure 7 shows the cumulative execution time for 64 parallel loop iterations. As expected, the waiting times in the internal engine queue are extremely high because there are four times as many workflow activities ready to execute than worker nodes. Again, the queuing time of ASKALON is larger than of SWF because of the higher middleware stack and the batch-mode access to resources. The average overhead per workflow activity without the file transfer and waiting overheads is of 17 seconds for SWF and 40 seconds for ASKALON, which is an increase by a factor of 3.4 for SWF and 2.7 for ASKALON compared to the previous scenario. The slight increase in execution time of the ASKALON version in the congested scenario is mainly caused by the file transfer overhead.

C. Discussion

To conclude, ASKALON has been designed to support a variety of heterogeneous and distributed computing environments, including Globus, gLite, EC2 and GroudSim [19]-based. This heterogeneity in the supported infrastructures is achieved through a modular architecture consisting of several layers and comprising complex services such as enactment engine, scheduling, and resource management. Although we paid high attention at tuning the ASKALON overheads when building the Cloud plugins, we exhibit a performance drop due to the higher middleware stack compared to SWF, tuned for working with the local, simpler, and more homogeneous EC2 infrastructure. For this reason, SWF features a much simpler architecture where the decider only decides which workflow activities can be executed next and not on which resources. The tasks of preparing the execution environment and transferring

local files from S3 need to be manually implemented by the programmer incurring a lower execution overhead, at the cost of a higher programming effort. Workflows consisting of numerous relatively short activities will mostly suffer from the larger ASKALON middleware overheads.

VIII. Conclusion

In this paper we proposed a method for automatic porting of scientific workflows to Amazon SWF, able to exploit the native performance of the EC2 infrastructure. The solution is based on the SHIWA fine-grained interoperability technology for translating workflows written across different languages through the common IWIR representation. This scalable software engineering solution enables five major workflow systems currently supporting the IWIR representation access the EC2 infrastructure through the SWF service: ASKALON, MOTEUR, Pegasis, Triana, and WS-PGRADE.

We presented in this paper the difficulties we encountered in translating an data flow-oriented ASLALON workflow into a control flow-oriented SWF decider program. The method is based on an algorithm that automatically generates the SWF decider Java program and the underlying activity functions in four phases: function signature, activity semantics, DAG control flow and data flow generation.

We presented experimental results for porting an original real-world ASKALON workflow to the EC2 infrastructure in two configurations: conversion to a Java SWF decider or execution through the ASKALON middleware connected to EC2 via an SSH plugin. The results demonstrate that the SHIWA fine-grained interoperability solution that translates an ASKALON workflow into an SWF version through the common IWIR representation is a promising alternative for porting workflows to a new infrastructure and able to exploit its native performance. Amazon SWF represents an attractive environment for running traditional workflow applications, especially those consisting of numerous relatively short activities affected by the large Grid middleware overheads. This is demonstrated by the performance of the automatically generated SWF workflow, which is similar to the manually optimised version. In contrast, porting existing Grid workflow middleware environments such as ASKALON to the Cloud, although effective, have performance drawbacks compared to the translated SWF version. The reasons of performance losses lie in the high middleware stack required for supporting a wider range of distributed and heterogeneous cluster, Grid, and Cloud computing infrastructures.

A downside of Amazon SWF is its proprietary implementation hosted by a commercial vendor who charges costs and may abandon this service anytime if it is lacking success. Another difference to clusters and Grids is the pull-based assignment of tasks to an unknown number of activity workers that requires different scheduling methods.

Acknowledgment

Austrian Science Fund project TRP 237-N23 and Standortagentur Tirol project RainCloud funded this research.

References

[1] I. J. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds., *Workflows for e-Science. Scientific Workflows for Grids*. Springer, 2007.

[2] K. Plankensteiner, J. Montagnat, and R. Prodan, "IWIR: a language enabling portability across grid workflow systems," in *Proceedings of the 6th workshop on Workflows in support of large-scale science*. ACM, 2011, pp. 97–106.

[3] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong *et al.*, "Askalon: A development and grid computing environment for scientific workflows," *Workflows for e-Science*, pp. 450–471, 2007.

[4] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and efficient workflow deployment of data-intensive applications on grids with moteur," *International Journal of High Performance Computing Applications*, vol. 22, no. 3, pp. 347–360, 2008.

[5] P. Kacsuk, "P-grade portal family for grid infrastructures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 3, pp. 235–245, 2011.

[6] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[7] I. Taylor, M. Shields, I. Wang, and O. Rana, "Triana applications within grid computing and peer to peer environments," *Journal of Grid Computing*, vol. 1, no. 2, pp. 199–217, 2003.

[8] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. Berman, and P. Maechling, "Scientific workflow applications on amazon ec2," in *E-Science Workshops, 2009 5th IEEE International Conference on*. IEEE, 2009, pp. 59–66.

[9] P. Riteau, M. Tsugawa, A. Matsunaga, J. Fortes, T. Freeman, D. LaBissoniere, K. Keahey *et al.*, "Sky computing on futuregrid and grid5000," in *5th Annual TeraGrid Conference: Poster Session*, vol. 68, 2010, p. 119.

[10] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *eScience, 2008. eScience'08. IEEE Fourth International Conference on*. IEEE, 2008, pp. 640–645.

[11] G. Morar, F. Schueller, S. Ostermann, R. Prodan, and G. Mayr, "Meteorological Simulations in the Cloud with the ASKALON Environment," in *Euro-Par 2012: Parallel Processing Workshops*. Springer, 2013, pp. 68–78.

[12] M. De Assunção, A. Di Costanzo, and R. Buyya, "Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. ACM, 2009, pp. 141–150.

[13] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 43–52.

[14] D. Franz, J. Tao, H. Marten, and A. Streit, "A workflow engine for computing clouds," in *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011, pp. 1–6.

[15] S. Pandey, D. Karunamoorthy, K. Gupta, and R. Buyya, "Megha workflow management system for application workflows," *IEEE Science & Engineering Graduate Research Expo*, Melbourne, Australia, 2009.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," *ECOOP'97Object-Oriented Programming*, pp. 220–242, 1997.

[17] I. Barstad and F. Schueller, "An Extension of Smith's Linear Theory of Orographic Precipitation: Introduction of Vertical Layers," *Journal of the Atmospheric Sciences*, vol. 68, no. 11, pp. 2695–2709, 2011.

[18] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java commodity Grid kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 89, pp. 643–662, 2001.

[19] S. Ostermann, K. Plankensteiner, and R. Prodan, "Using a new event-based simulation framework for investigating resource provisioning in Clouds," *Scientific Programming*, vol. 19, no. 2, pp. 161–178, 2011.