A Workflow Runtime Environment for Manycore Parallel Architectures

M. Janetschek[a,*], R. Prodan[a,**], S. Benedict[b]

[a] *University of Innsbruck, Austria*
[b] *HPCCLoud Research Laboratory, SXCCE, India*

Abstract

We introduce a new Manycore Workflow Runtime Environment (MWRE) to efficiently enact traditional scientific workflows on modern manycore computing architectures. MWRE is compiler-based and translates workflows specified in the XML-based Interoperable Workflow Intermediate Representation (IWIR) into an equivalent C++-based program. This program efficiently enacts the workflow as a stand-alone executable by means of a new callback mechanism that resolves dependencies, transfers data, and handles composite activities. Furthermore, a core feature of MWRE is explicit support for full-ahead scheduling and enactment. Experimental results on a number of real-world workflows demonstrate that MWRE clearly outperforms existing Java-based workflow engines designed for distributed (Grid or Cloud) computing infrastructures in terms of enactment time, is generally better than an existing script-based engine for manycore architectures (Swift), and sometimes gets even close to an artificial baseline implementation of the workflows in the standard OpenMP language for shared memory systems. Experimental results also show that full-ahead scheduling with MWRE using a state-of-the-art heuristic can improve the workflow performance up to 40%.

Keywords: scientific workflows, manycores, workflow execution plan, full-ahead scheduling

1. Introduction

Nowadays, computers exhibit an ever higher number of heterogeneous processing cores with a growing trend to combine general-purpose CPUs with specialized computing units. As a result, modern shared memory heterogeneous manycore systems have become increasingly complex to program. Traditional programming paradigms for

*Corresponding author
**Principal corresponding author
Email addresses: `matthias@dps.uibk.ac.at` (M. Janetschek), `radu@dps.uibk.ac.at` (R. Prodan), `shajulin@sxcce.edu.in` (S. Benedict)

© 2017. This manuscript version is made available under the CC-BY-NC-ND 4.0 license (see http://creativecommons.org/licenses/by-nc-nd/4.0/)
DOI: https://doi.org/10.1016/j.future.2017.02.029

shared memory computers have their roots in symmetric multi-processing and, therefore, are struggling to fully exploit the performance of today's heterogeneous systems.

Distributed computing infrastructures (DCI) such as Grids and Clouds are heterogeneous by definition for which a large number of programming methods and paradigms exist. One of the most successful paradigms for implementing high-performance computing applications on DCIs are *scientific workflows*, originally designed to easily create large and complex applications by reusing existing (often legacy and monolithic) software components and assembling them through well-defined control flow and data flow dependencies. Existing DCI workflow engines are currently mature and come with rich ecosystems that support the user in all aspects of the workflow life-cycle including creation, scheduling, execution, monitoring and steering, interfaced towards the domain scientists and ease of use rather than the computer science underneath.

Because of the similarity in terms of scale and heterogeneity, workflow systems represent today a promising alternative for development and execution of scientific applications on shared memory heterogeneous manycore architectures. Moreover, as large-scale DCI infrastructures are nowadays composed of powerful manycore parallel machines, exploiting them in an efficient fashion becomes an increasingly important requirement. However, DCI systems are inherently different from shared memory manycore systems. The parallelism in DCI applications is of coarse-grained nature because of the high overheads and latencies involved with moving the data and computation between physically distributed systems. To achieve performance in such environments, the computational tasks need to be large enough to hide these overheads and latencies. DCI workflow systems are typically designed as external services that efficiently distribute a relatively small number of computational tasks with high submission overheads and large data transfers. While such overheads are acceptable in distributed systems, tightly-coupled manycore parallel machines are much more sensitive to latencies and other sources of performance penalties. The parallelism in shared memory manycore systems is usually of fine-grained nature dealing with a high number of small tasks and insignificant overheads and latencies compared to DCI systems.

We proposed a new *Manycore Workflow Runtime Engine (MWRE)* in [10] designed to test the viability of the workflow paradigm on shared-memory manycore systems and identify its limitations and constraints. This paper extends our previous work [10] by explaining our workflow engine in more detail, presenting additional experiments showing that full-ahead scheduling is viable on manycore computers, and discussing how we can further improve performance by applying workflow transformations. More specifically, we aim to research how the workflow model, traditionally tailored for coarse-grained parallelism, can cope with fine-grained parallelism typically found in shared memory manycore applications. We purposely designed MWRE to efficiently exploit the low latency characteristics and resources of manycore parallel architectures, with special focus on efficient support for multi-objective full-ahead scheduling and enactment optimization techniques. Its distinguishing characteristic is that it *compiles* an input workflow to an imperative *stand-alone workflow program*, instead of interpreting and orchestrating the workflow specification as traditionally done by today's scientific workflow engines acting as external services. MWRE is based on a source-to-source compiler able to process abstract scientific workflows in the Interoperable *Workflow Intermediate Representation (IWIR)* [18], a common workflow specification developed in the European SHIWA project (SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs)[1] that enables translation of workflow across four major scientific workflow systems: ASKALON [5], MOTEUR [8], Triana [21] and WS-PGRADE [12]. Using an activity repository, the compiler generates a standalone C++ workflow program that independently executes on the underlying manycore infrastructure with the help of a workflow engine, linked as an external shared C++ library. Our engine uses a novel low-overhead *callback* mechanism to resolve dependencies, transfer data, and handle composite activities, rather than high-overhead reflection and type introspection as done in existing DCI engines.

MWRE workflow applications can also be used together with traditional workflow engines, e.g. as sub-workflows of a larger distributed workflow.

The paper is organised as follows. Section 2 gives a short introduction to scientific workflows and related concepts. Section 3 discusses the related work. Section 4 describes the architectural design of our new workflow engine, followed by an in-depth description on our workflows enactment technique in Section 5 and of our source-to-source compiler in Section 6. Section 7 evaluates the performance, overheads and scalability of our engine and Section 8 concludes the paper.

2. Model

The design and implementation of scientific workflows consists of two parts: an *abstract part* and a *concrete part*. We give a short overview of these two parts in the following subsections.

2.1. Workflow design: the abstract part

The abstract part (see Figure 1) of a scientific workflow specification comprises a hardware and middleware agnostic (and therefore easily portable) description of the workflow structure, the activities involved (identified by a unique name and a type), and the data and control flow dependencies between the activities. The individual activities are treated as black-boxes such that only the input and the output signatures are known.

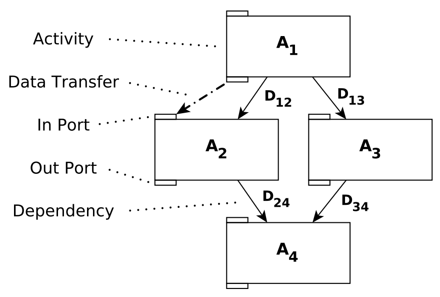


Figure 1: An abstract part of a scientific workflow.

There are usually two different types of workflow activities:

1. *Atomic activities* are basic indivisible computations units such as a legacy codes;
2. *Composite activities* combine several fine granular workflow activities, including atomic and other composite ones, to form coarse granular activities and impose a control flow on the contained inner activities.

[1]`http://www.shiwa-workflow.eu/`

Typical composite activities are sequential and parallel loops, conditional activities, and sub-workflows.

2.2. Workflow implementation: the concrete part

The concrete part of a workflow contains the hardware and middleware-dependent implementations of the atomic activities and their accompanying meta-information. This part is often highly specific to each individual workflow system and the underlying DCI. It usually contains information about the available activity implementations, locations where they are installed, how they can be executed, and any other further information indented to help the workflow engine in selecting the most appropriate activity implementation.

2.3. Workflow enactment

A workflow engine executes a workflow instance (operation usually called *workflow enactment*) by traversing the DAG representing the workflow structure, determining the state of the individual activities, transferring data from finished activities to their successors in the dependency graph, unrolling composite activities and replacing them with the resulting subgraph, and delegating the actual execution of atomic activities to the scheduling and execution subsystems. We call the resulting DAG where composite activities have been replaced with their contained subgraphs enriched with additional state information a *Workflow Execution Plan* (WEP).

We distinguish between two types of workflow enactment modes:

- In *early enactment mode* the engine re-evaluates the WEP as soon as there are activity state changes, and tries to complete it as early as possible. This mode usually comes with a much higher overhead, but results in a more complete WEP comprising more information, which allows the scheduler to better plan the workflow execution on the underlying resources;

- In *late enactment mode* (also called *lazy evaluation mode*) the engine only re-evaluates and completes the WEP when it is absolutely necessary for the workflow enactment, and only partially completes it as far as it is required for continuing the enactment.

2.4. Workflow scheduling

Scheduling describes the process of mapping atomic activities to available computing resources where they are executed. A concrete mapping of activities to computing resources is called *workflow schedule*. The scheduler usually tries to optimize the schedule by maximising or minimizing a given utility function, typically the overall execution time.

To be able to produce an optimal mapping the scheduler need to know the resource requirements of an individual activity in advance. We assume in this paper that the scheduler has accurate knowledge of the resources required by an individual activity, but in practice obtaining this knowledge is still a steep challenge.

Existing scheduling algorithms can be broadly divided into two categories [25]:

- *Just-in-time scheduling algorithms* only consider the next activities to be scheduled when deciding on a mapping and ignore the rest of the WEP. They are usually linear in complexity with the number of activities (i.e. $O(N)$) and come with a low overhead, but due to limited information the generated schedule may not be that good;
- *Full-ahead scheduling algorithms* use the entire WEP when deciding on a mapping and try to find an optimal schedule in the search space. They usually come with a much higher overhead, but consider more workflow information and therefore, produce in general better results. Generating a full-ahead schedule is an NP-hard problem [24] and therefore, most existing full-ahead scheduling algorithms are approximate heuristic algorithms [22].

3. Related Work

Most scientific workflow systems like ASKALON [5], MOTEUR [8], Pegasus [3], Kepler [1], Taverna [27], Triana [21], or WS-PGRADE [12] are targeted at DCIs such as Grids and Clouds. They feature a rather large software stack in order to communicate with lots of different middlewares and communication protocols, and are optimized for moving large amount of data between distributed computing resources.

Swift [26] is an exception by being a light parallel script-based engine not restricted to DCIs, but open for general use. It is limited to files in modelling data dependencies, and provides no ways of calculating full-ahead workflow schedules.

Swift/T [28] is a completely new implementation of Swift, which translates a workflow script written in the Swift workflow language into a native MPI program for highly scalable dataflow processing on distributed systems. Like Swift, it does not provide support for full-ahead scheduling.

Pegasus MPI cluster [19] uses MPI for communication and job submission to execute large, fine-grained workflows on distributed petascale systems. It has been developed to execute traditional scientific workflows on HPC systems with exotic, highly optimized networks that do not support a TCP/IP stack.

Dispel4py [7] is a Python library for describing abstract stream-based workflows for distributed data-intensive applications. It supports different backends like MPI, Apache STORM and Python multiprocessing, with no support full-ahead scheduling.

Nextflow[2] is a Java-based DSL similar to the Swift workflow language which uses the UNIX pipeline concept as programming model for parallel and scalable programs. It also does not support full-ahead scheduling.

Bobolang [6] is a specialized DSL for data-parallel algorithms. It uses Bobox [2] as backend, which is a parallel processing framework intended for data-intensive computations based on a non-linear pipeline.

The most prevalent parallel programming API standard on shared memory systems is OpenMP[3]. Other native parallel programming APIs like MPI[4] or Charm++ [13] tar-

[2] `http://nextflow.io/`
[3] `http://openmp.org/`
[4] `http://www.mpi-forum.org/`

get distributed memory systems, but have the common drawback of not modelling data dependencies and not providing means of generating a complete full-ahead schedule.

StarSS [17] is a dependency-driven task execution API for shared memory systems, which also supports distributed memory systems through a hybrid MPI/StarSS approach. It provides directives that annotate C/Fortran source codes to define tasks and dependencies between them. StarSS does not allow reusing legacy workflows and does not provide any infrastructure for full-ahead scheduling.

JOpera [15] is a management tool for business workflows that creates Java byte code from the workflow specification through an event driven state machine instead of directly interpreting the specification. JOpera comes with an integrated workflow design and execution environment as an Eclipse plugin hiding the compilation process from the user. JOpera is tailored to business workflows and does not support advanced full-ahead scheduling.

In summary, Swift, Dispel4py and Bobolang/Bobox are the best-suited alternatives for implementing scientific workflows on manycore systems aside our proposed system. However, they do not support full-ahead scheduling, or are specialized in a very specific problem class. Furthermore, workflows implemented using Swift, Dispel4py or Bobolang do not natively run on the intended target platform. To the best of our knowledge, there is no related work that implements a native specialised compiled-based scientific workflow engine which is specifically focused on multi-objective full-ahead scheduling on shared-memory manycore parallel architectures.

4. Manycore Workflow Runtime Engine (MWRE)

Based on the requirements outlined in the introduction, we designed a *Manycore Workflow Runtime Engine (MWRE)* tailored to heterogeneous shared-memory manycore systems. Our main design principle was to provide a set of feature similar to the ones found in the current DCI workflow engines, while addressing the special characteristics and requirements of manycore systems. In this section we discuss the basic architecture of our workflow system and give short descriptions of the components.

We use the Persistence of Vision Raytracer (POV-Ray) workflow as a running example to illustrate the workflow representation, translation and enactment. POV-Ray [16] is a free tool for creating three-dimensional graphics, which is known to be a time-consuming process used not only by hobbyists and artists, but also in biochemistry research, medicine, architecture, and mathematical visualization. We modelled a POV-Ray rendering scenario as a workflow depicted in Figure 2, where the description of a movie can be

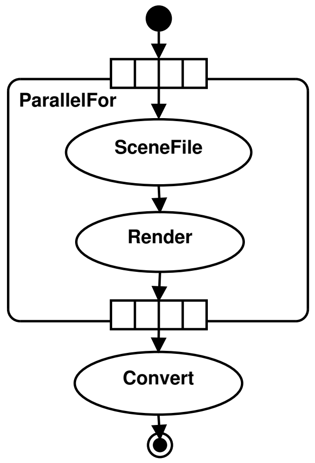


Figure 2: The POV-Ray workflow.

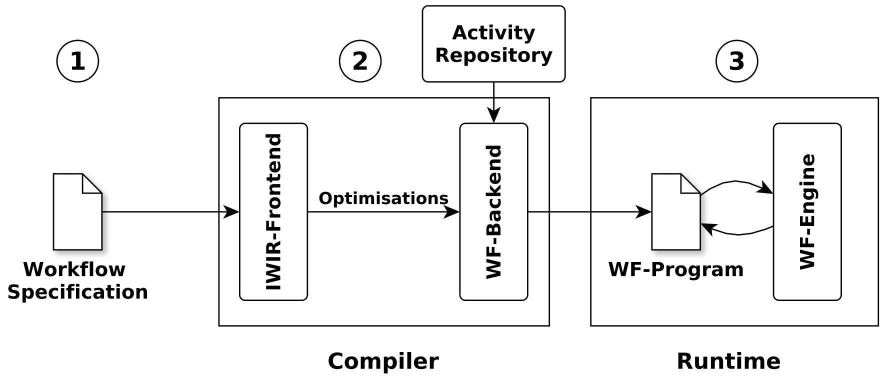


Figure 3: MWRE architecture.

separated in several scenes, each scene being composed of several frames that can be rendered (e.g. in `.png` format) in a `parallel for` loop. The `Scene` activity before the `Render` returns the scene file for a given scene name. Finally, all frames are merged into a `.mpg` movie using a `Convert` activity (e.g. running a png2yuv followed by an `ffmpeg` conversion).

Figure 3 presents the overall architecture of the MWRE consisting of three main parts: ① the workflow specification, ② a source-to-source compiler, and ③ the workflow runtime environment. The main difference between the MWRE's architecture and the traditional DCI workflow engines is the source-to-source compiler that generates a native C++ program from the workflow specification. Furthermore, MWRE does the mapping of abstract activity types to concrete activity implementations at compile-time, instead of runtime as done by most DCI engines.

4.1. Workflow specification

The input to MWRE is a workflow specification (①) encoded in the Interoperable Workflow Intermediate Representation (IWIR) [18]. IWIR is an intermediate workflow specification language designed in the European SHIWA project[5] to enable interoperability of different workflow systems. As most workflow environments require workflows to be written in their respective workflow language to be able to execute them, exchanging workflows usually means that the workflow specification needs to be translated from one workflow language to another. IWIR eases this translation process by providing a common intermediate representation and thus reducing the number of required translators from $2 \cdot n^2$ to $2 \cdot n$. IWIR is currently supported by four workflow systems: ASKALON [5], MOTEUR [8], Triana [21] and WS-PGRADE [12].

Using IWIR to specify MWRE workflows has several advantages allowing reusing existing tool chains with no need to implement existing functionality. First, because it

[5]`http://www.shiwa-workflow.eu/`

is designed for interoperability, it captures all concepts and constructs found in most workflow languages, and can therefore be seen as a superset of the major workflow languages. Second, it is well-defined and easy to parse. Third, it allows automatic translation and reuse of existing workflows for experiments across different systems without further modifications. Fourth, it allows integration of new languages and new platforms with O(1) complexity through IWIR front-end or back-end support. And fifth, the implementation of a MWRE workflows is not tied to a specific development environment, but domain scientists can use their favourite workflow development tools from any workflow environment that supports IWIR translation.

Listing 1 shows the IWIR specification of our POV-Ray workflow. The `toplevel` composite activity represents the start of the workflow and is defined in lines 2 – 66. The input ports of the `toplevel` activity (one file and two integers) are defined in lines 3 – 7 and also represent the workflow input ports. A real-world POV-Ray workflow has more input parameters, but we omitted them for brevity reasons. The output ports of the `toplevel` activity (one file) defined in lines 55 – 57 again represent the workflow output ports too. Lines 9 – 45 represent the `parallel for` `ParallelFor` loop activity containing the atomic activities `SceneFile` (lines 16 – 23) and `Render` (lines 24 – 33). The loop activity is then followed by the atomic activity `Convert`, represented in lines 46 – 53.

4.2. Source-to-source compiler

The IWIR workflow specification is then given to the source-to-source compiler (②), which translates it into a C++ *workflow program* using the API provided by MWRE. We believe that compiling the workflow application into a native executable program is the way to achieve the "best" performance on shared-memory manycore systems while minimizing enactment latencies and other middle-ware overheads. The compiler is also responsible for mapping the abstract workflow activity types to concrete activity implementations. Performing this task at compile-time, instead of runtime as done by most DCI workflow systems, saves additional overhead and eases configuration of the workflow program. For this purpose, the compiler has access to a repository that contains activity implementations in form of pre-compiled libraries, binaries, GPU kernels (i.e. `CUDA`, `OpenCL`), source-code snippets, and shell-scripts. The compiler also allows a 1-to-*n* mapping of activity types to implementations. Metadata stored for each mapping allows the scheduler to efficiently select the most suitable implementation at runtime. Using a source-to-source compiler also allows applying transformations to the workflow structure to optimize the performance of the generated workflow program, similar to the compiler optimisations for shared memory systems. We describe the source-to-source compiler in Section 6 in detail.

4.3. Runtime environment

The runtime environment of MWRE (③) invokes and executes the workflow program created by the source-to-source compiler. It is implemented as a C++ library considering several performance concerns, such as overhead, memory footprint, and resource utilization. We describe MWRE workflow engine in Section 5 in detail.

Listing 1 IWIR specification of the POV-Ray workflow.

```
<IWIR version="1.1" wfname="NewWorkflow" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="toplevel">
    <inputPorts>
      <inputPort name="scene" type="string"/>
      <inputPort name="totalFrames" type="integer"/>
      <inputPort name="framesPerActivity" type="integer"/>
    </inputPorts>
    <body>
      <parallelFor name="ParallelFor">
        <inputPorts>
          <inputPort name="scene" type="string"/>
          <inputPort name="numFrames" type="integer"/>
          <loopCounter name="frameCounter" from="1" to="" step=""/>
        </inputPorts>
        <body>
          <task name="SceneFile" tasktype="SceneFile">
            <inputPorts>
              <inputPort name="scene" type="string"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="povFile" type="file"/>
            </outputPorts>
          </task>
          <task name="Render" tasktype="Render">
            <inputPorts>
              <inputPort name="povFile" type="file"/>
              <inputPort name="startFrame" type="integer"/>
              <inputPort name="numFrames" type="integer"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="frames" type="collection/file"/>
            </outputPorts>
          </task>
        </body>
        <outputPorts>
          <outputPort name="frames" type="collection/collection/file"/>
        </outputPorts>
        <links>
          <link from="ParallelFor/scene" to="SceneFile/scene"/>
          <link from="SceneFile/povFile" to="Render/povFile"/>
          <link from="ParallelFor/numFrames" to="Render/numFrames"/>
          <link from="ParallelFor/frameCounter" to="Render/startFrame"/>
          <link from="Render/frames" to="ParallelFor/frames"/>
        </links>
      </parallelFor>
      <task name="Convert" tasktype="Convert">
        <inputPorts>
          <inputPort name="frames" type="collection/file"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="outFile" type="file"/>
        </outputPorts>
      </task>
    </body>
    <outputPorts>
      <outputPort name="finalMovie" type="file"/>
    </outputPorts>
    <links>
      <link from="toplevel/totalFrames" to="ParallelFor/frameCounter/to"/>
      <link from="toplevel/framesPerActivity" to="ParallelFor/frameCounter/step"/>
      <link from="toplevel/framesPerActivity" to="ParallelFor/numFrames"/>
      <link from="toplevel/scene" to="ParallelFor/scene"/>
      <link from="ParallelFor/frames" to="Convert/frames"/>
      <link from="Convert/outFile" to="toplevel/finalMovie"/>
    </links>
  </blockScope>
</IWIR>
```

5. MWRE Workflow Enactment

This section describes the enactment model of our workflow engine. In order to be able to meet the requirements of shared-memory manycore systems, we designed it significantly different from traditional scientific workflow engines for DCIs by paying special attention to the efficient use of resources, low overheads, and arbitrarily complex data types for input and output activity ports.

5.1. Workflow enactment model

Workflow engines for DCIs usually rely on reflection and type introspection to generate the complete workflow structure and retrieve detailed activity information. Based on this information, the engine executes the workflow and handles the dependency resolution and the data transfers. One problem with this approach is its high overhead and increased resource use. Furthermore, to reduce the complexity and simplify the engine implementation, only a restricted set of data types for ports is usually supported. While this is sufficient for DCIs where complex data structures are usually transmitted via files, it is more efficient on shared memory systems to directly transfer them via shared memory requiring support for arbitrarily complex data types. To circumvent these problems, we use a novel design which, instead of relying on reflection and type introspection, employs a callback mechanism to resolve dependencies, transfer data, and handle composite activities.

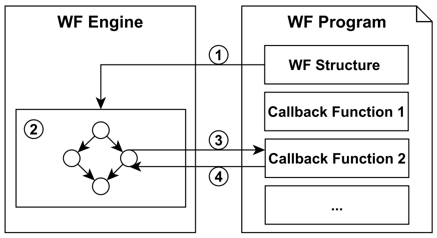


Figure 4: MWRE workflow execution model.

Figure 4 shows the execution model of our engine. First (①), the engine reads the workflow structure from the workflow program. Listing 2 shows the structure of the C++ workflow program generated from the POV-Ray workflow representation. Lines 1 – 4 list the available activity implementations of the `SceneFile` atomic activity, lines 5 – 7 the implementations of the `Render` atomic activity, and lines 8 – 11 the implementations of the `Convert` atomic activity. Afterwards, lines 12 – 19 encode a table containing all workflow activities. Each line in this table represents a single activity and contains the activity name, the parent activity which represents the composite activity the activity is contained within, the successors and predecessors of the activity representing the data- and control-flow dependencies, the callbacks associated with the activity, and other activity dependent information. Finally, line 21 defines the workflow itself and defines the initial start activity.

The engine maintains bookkeeping record information for the workflow and each activity instance. A record entry includes, among other things, the current state of the activity instance and references to the input and output data buffers. By amending the engine with an API that allows accessing these records, third-party applications can easily monitor the workflow progress. Furthermore, workflow steering can be easily implemented by allowing manipulation of the input and output data structures by third-party applications. However, due to its architecture, the engine cannot provide

Listing 2 POV-Ray workflow program snippet.

```
ActivityImplemenation SceneFileImpls[] = {
   /* {type, function-pointer, meta-data} */
   PThread, &renderImplFunc, {{“SSE”, “SSE2”}}
};

ActivityImplemenation RenderImpls[] = {
   PThread, &renderImplFunc, {{“SSE”, “SSE3”}, {“key2”, “value2”}}
};

ActivityImplemenation ConvertImpls[] = {
   PThread, &convertImplFunc1, {{“SSE”, “SSE”}, {“key3”, “value3”}}
   OpenCl, &convertImplFunc2, {{“OpenCL”, “2.1”}}
};

ActivityTemplate Activities[] = {
   /* (name, ID, parentID, branchID, succs, preds, callbacks, activity-specific) */
   AT::Container(“toplevel”, 1, 0, 0, {}, {}, callbacks, ...),
   AT::Atomic(“SceneFile”, 2, 1, 0, {3}, {}, callbacks, SceneFileImpls, 1, ...),
   AT::PForLoop(“PFor”, 3, 1, 0, {5}, {2}, callbacks, ...),
   AT::Atomic(“Render”, 4, 2, 0, {}, {}, callbacks, RenderImpls, 1, ...),
   AT::Atomic(“Convert”, 5, 1, 0, {}, {3}, callbacks, ConvertImpls, 2, ...),
};

/* (workflow name, activities, id of start activity) */
Workflow workflow(“povray”, Activities, 1);

int main() {
   wf_input wf_int = ...;
   wf_output wf_out = WorkflowEngine.startWorkflow(workflow, wf_in);
}
```

data type information inside input/output data buffers and, therefore, third-party applications used for workflow steering need to retrieve this information from other sources.

From the workflow structure, the engine constructs the WEP (②) and repeatedly traverses it during the enactment. When traversing the WEP, the engine executes so-called *visitor functions* for each node. In traditional engines, the visitor functions are directly responsible for the dependency resolution and data transfers. In contrast, the visitor functions in our engine are only responsible for orchestrating the execution of the associated pre-compiled callback functions (③) that directly modify the state of the associated workflow activities (④). The callback functions are part of the workflow specification, and it is the responsibility of the source-to-source compiler to generate them. Each callback function implements a specific functionality, such as transferring the input data, collecting the output data from children for composite activities, or resolving the condition of a `while` loop. This approach has the advantage of keeping the workflow engine lightweight and efficient. Instead of knowing all the details about the workflow, the engine only knows what is necessary to traverse the workflow structure and to keep track of the execution status, with no need to implement generic functionality for dependency resolution, data transfer and other tasks that introduce performance overheads. Another advantage is that it allows arbitrary data type support and facilitates extensibility. Every functionality that needs information about data types is encapsulated in a purposely-tailored callback function used in a particular workflow. New functionality can be easily implemented in callback functions too without needing to modify the engine.

Listing 3 Visitor function of a `parallel for` loop.

```
1:  function visitPFor(activity instance $AI_i$)
2:      if $\neg AI_i$.allInputsAvailable then
3:          $AI_i$.inputCallback
4:      end if
5:      if $AI_i$.state < ReadyForExecution then
6:          $AI_i$.forCounterCallback
7:      end if
8:      if $AI_i$.state = ReadyForExecution then
9:          if $\neg AI_i$.isUnrolled then
10:             UnrollLoop($AI_i$)
11:         end if
12:         visitChildren($AI_i$)
13:     end if
14:     if $AI_i$.state $\geq$ ReadyForExecution $\wedge \neg AI_i$.allOutputsAvailable then
15:         $AI_i$.outputCallback
16:     end if
17:     if $AI_i$.state < Finished $\wedge$ all children are finished then
18:         $AI_i$.setState(Finished)
19:     end if
20: end function
```

Listing 3 shows the generic visitor function used to visit a `parallel for` loop (Figure 5 shows a simplified state diagram of the `parallel for` loop). At first, it calls the input callback function in line 3 to fetch the input data from its predecessors, if they are not already available. If the `for` loop is not ready for execution (meaning that the for loop counter has not been yet evaluated), the evaluating `for` counter callback gets executed in line 6. When the for loop is ready for execution (meaning that the iteration space of the loop counter is known), the loop gets unrolled and an activity instance is created for each child and each iteration in line 10. Afterwards, the visitor functions for the children activities are called in line 12. In our case, the atomic activity `render` is the only child. The visitor function of an atomic activity executes the input callback function and, when all inputs are satisfied, the activity gets executed and the output data is available in the output buffer of the atomic activity. The visitor function of the `for` loop activity executes the output callback function in line 15, which fetches data from the output buffers of the child activities and stores them into the output buffer of the loop activity. After all child activities have finished their execution, the activity state is set to `Finished` in line 18.

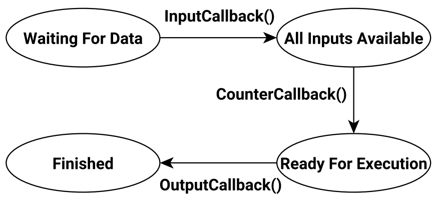


Figure 5: Simplified state diagram of a `parallel for` activity.

As an example, Listing 4 shows the implementations of the callback functions of the `parallel for` loop (`ParallelFor`) of the POV-Ray workflow. The first callback (line 1) is the *input callback* responsible for initializing the activity's input data buffer. For this, it accesses the input data buffer of the parent activity, checks whether it contains valid data, saves it into the input buffer of the `for` activity, and sets the `allInputsAvailable` flag to `true` if all input data fields contain valid data to avoid unnecessary function invocations. The second callback (line 12) is the *output callback*

Listing 4 Callback functions of the POV-Ray `ParallelFor` loop activity.

```
 1: function forLoopInputCallback(ForLoopInstance $for$)
 2:     $for\_in \leftarrow$ retrieveInStruct($for$)                    ▷ Initialisation of $for\_in$ input port
 3:     $parent\_in \leftarrow$ retrieveInStruct(getParentInstance($for$))
 4:     if isValid($parent\_in.povFile$) then
 5:         $for\_in.povFile \leftarrow parent\_in.povFile$
 6:     end if

 7:     [. . . ]          ▷ Similar code for initializing $totalFrames$, $framesPerActivity$ and $scene$ ports

 8:     if isValid($for\_in.povFile$, $for\_in.totalFrames$, $for\_in.framesPerActivity$, $for\_in.scene$) then
 9:         $for$.setAllInputsAvailable(true)
10:     end if
11: end function

12: function forLoopOutputCallback(ForLoopInstance $for$)
13:     $for\_out \leftarrow$ retrieveOutStruct($for$)
14:     for $i \leftarrow 0$ to $for$.getIterationCount() do
15:         $child\_out \leftarrow$ retrieveOutStruct(getChildInstance($for, i, 0$))
16:         if isValid($child\_out.frames$) then
17:             $for\_out.frames$.set(i, $child\_out.frames$)
18:         end if
19:     end for
20:     if containsNValidEntries($for$.getIterationCount, $for\_out.frames$) then
21:         $for$.setAllOutputsAvailable(true)
22:     end if
23: end function

24: function forLoopCounterCallback(ForLoopInstance $for$)
25:     $for\_in \leftarrow$ retrieveInStruct($for$)
26:     if isValid($for\_in.totalFrames, for\_in.framesPerActivity$) then
27:         $for$.setCounter(0, $for\_in.totalFrames/for\_in.framesPerActivity$, 1)
28:         $for$.setState(ReadyForExecution)
29:     end if
30: end function
```

responsible for filling in the output data buffers by accessing the output data buffers of its children, checking the validity of the data, storing the data into the output buffer of the `for` loop, and setting the `allOutputsAvailable` flag to `true` if all output data fields contain valid data. The third callback (line 24) is the *for counter callback* responsible for initializing the loop counter of the `for` loop activity `PForLoop` by checking if all required data is available, setting the counter, and declaring the activity as ready-for-execution by modifying its state.

5.2. Scheduling and activity execution

After the WEP has been evaluated for the first time and after every change, the workflow engine sends the new WEP to the scheduler. The scheduler then maps atomic activities to computing resources. Traditional workflow engines for DCIs often only lazily evaluate the WEP as far as it is required for workflow enactment to enact the ready to execute activities. However, full-ahead schedulers require a WEP that is as complete as possible to achieve a better schedule and, therefore, need to compute the WEP on their own. While this was not a major issue in traditional engines, as they are more tolerant to high overheads and usually run on own external dedicated systems, the performance overhead are of a greater concern for workflows on shared memory manycore systems. Because of this, we designed the WEP maintained by the workflow engine as complete as possible and such that all the relevant information is shared with the scheduler to avoid unnecessary recalculations.

Each activity implementation comes with meta-data in form of key-value pairs that can be used by the scheduler to select an appropriate implementation for execution. There is a set of well-defined key-value pairs such as the key `OpenCL` used to specify the OpenCL version of a specific activity implementation, or `SSE` to specify the required version of the Streaming SIMD Extension. Moreover, users can also define their own arbitrary key-value pairs. For example, line 6 in Listing 2 specifies that the activity implementation requires *SSE3* and there is also a user-defined key-value pair `key2=value2`.

After the scheduler has selected appropriate activity implementations and generated a schedule, the individual scheduling decisions are sent to the appropriate execution subsystems (e.g. `pthread` subsystem for `x86` code, `CUDA`/`OpenCL` subsystem for GPU kernels). For this purpose, MWRE provides a common extensible interface for the execution subsystems designed to support a wide variety of computing resources and to allow the scheduler interact with a specific execution subsystem without knowing the details of the underlying implementation. The execution results are then send back to the scheduler, which then handles them accordingly and notifies the workflow engine of any activity state changes.

Whenever an activity execution fails, the scheduler decides how the fault is resolved. We offer an API for implementing third-party schedulers and using this API the scheduler can resubmit the activity as it is, select a different activity implementation, map the activity onto a different resource, or simply fail the entire workflow. For example, our scheduler implementations try to resubmit a failed activity three times and after that the entire workflow is put into failed state, but other implementations may come up with more sophisticated fault handling methods.

6. Source-to-Source Compiler

In our MWRE workflow environment, we employ the Insieme compiler[6] to convert a workflow specification written in IWIR into a C++ workflow program using our engine. Insieme is a research source-to-source compiler targeted at automatically optimizing parallel programs for homogeneous and heterogeneous multi-core architectures, which currently supports C, Cilk, OpenMP and OpenCL languages. Insieme uses a single high-level parallel intermediate representation (IR) called INSPIRE [11] in all stages of the compilation process. INSPIRE is the best-suited compiler-based IR because it allows us to concentrate on a single language (unlike e.g. `gcc`, which uses different IR languages in different compiler stages), it includes constructs for high-level constructs like `for` loops (unlike e.g. LLVM whose IR language only consists of simple assembler-like instructions), and it additionally includes constructs directly representing parallel concepts (like e.g. `parallel for` loops). High-level constructs ease program analysis, while dedicated parallel constructs allow to precisely represent

[6]http://www.insieme-compiler.org

an IWIR workflow. Furthermore, INSPIRE has been designed as an extensible IR allowing to easily add new data types or program constructs.

The conversion process from an IWIR workflow specification into an equivalent MWRE workflow program consists of three phases: (1) translation of the IWIR specification into INSPIRE, (2) compiler transformations on the INSPIRE IR, and (3) conversion of the INSPIRE IR into a MWRE workflow program.

6.1. Phase 1: IWIR translation into INSPIRE

IWIR is a block-structured, data flow-oriented workflow language with control flow constructs very similar to modern imperative programming languages like C. Therefore, IWIR workflows can also be represented by IR languages originally designed for imperative languages, for example by replacing all IWIR constructs with their corresponding INSPIRE high-level counterparts.

Listing 5 shows the INSPIRE program representing our POV-Ray example workflow. To ease the program analysis, each activity is represented by a function that encapsulates the code implementing the activity and whose signature represents the activity input and output ports. In our example, function `fun003` (lines 22 – 29) represents the start of the workflow and defines the workflow input (lines 23 – 25) and output ports (line 28). The `_iwir_wf_input_` and `_iwir_wf_output_` functions are helper functions whose only purpose is to help identify the workflow input and output ports. The `fun002` function (lines 15 – 20) represents the `toplevel` activity corresponding to the IWIR snippet between lines 2 – 66 in Listing 1. The `fun001` (lines 11 – 13) and `fun000` (lines 1 – 9) functions represent the `parallel for` `ParallelFor` activity corresponding to the IWIR snippet between lines 9 – 45 in Listing 1. The `fun001` function defines the parallel loop itself and the activity ports, while the `fun000` function represents the loop body. Finally, line 31 represents the root of the workflow program. The `_iwir_input_wrapper_` and `_iwir_output_wrapper_` functions in each function call are helper functions used to determine the function arguments representing the input and the output ports. Atomic activities are represented by a call to the `_iwir_atomic_` function, the first parameter being the name of the atomic activity type, the second parameter being the atomic activity type, and the rest of the parameters representing the input and output ports and their names.

Imperative languages, including INSPIRE, use variables and operations upon variables to describe the data flow. Data flow-driven programs like IWIR can easily be converted into an imperative program by representing every data dependency as a variable using a *single static assignment*, meaning that every variable is written only once, which ensures that we can easily analyse the data flow to recreate the data dependencies. The only exceptions are loop ports that can be written more than once, however, they can only be written once per loop iteration, which makes them still easy to analyse. IWIR also does not allow any data manipulations outside of atomic activities, therefore, an INSPIRE program representing an IWIR workflow is only allowed to transport data from one activity to another, but not to manipulate this data.

6.2. Phase 2: INSPIRE compiler transformations

Compiler transformations are usually applied to a program in order to optimize one or more program characteristics, such as execution time or program size. They

Listing 5 INSPIRE representation of the POV-Ray workflow.

```
let fun000 = fun(ref<ref<array<char,1>>> scene, ref<int<4>> numFrames,
      ref<collection<collection<file<ref<array<char,1>>>>>> frames, int<4> counterStart,
      int<4> counterStop, int<4> counterStep) -> unit {
  for(decl int<4> forCounter = counterStart .. counterStop : counterStep) {
    decl ref<file<ref<array<char,1>>>> povFile =
          (var(undefined(type<file<ref<array<char,1>>>>)));
    _iwir_atomic_("SceneFile", "SceneFile", _iwir_input_wrapper_(scene), "scene",
          _iwir_output_wrapper_(povFile), "povFile");
    decl ref<collection<file<ref<array<char,1>>>>> renderFrames =
          (var(undefined(type<collection<file<ref<array<char,1>>>>>)));
    _iwir_atomic_("Render", "Render", _iwir_input_wrapper_(povFile), "povFile",
          _iwir_input_wrapper_(forCounter), "forCounter", _iwir_input_wrapper_(numFrames),
          "numFrames", _iwir_output_wrapper_(renderFrames), "renderFrames");
    (ref_collection_at(frames, forCounter) := ( *renderFrames));
  };
};

let fun001 = fun(ref<ref<array<char,1>>> scene, ref<int<4>> totalFrames, ref<int<4>>
      framesPerActivity, ref<collection<collection<file<ref<array<char,1>>>>>> frames) ->
      unit {
  pfor(getThreadGroup(0), 1, ( *totalFrames), ( *framesPerActivity), bind(counterStart,
        counterStop, counterStep){fun000(_iwir_input_wrapper_(scene),
        _iwir_input_wrapper_(framesPerActivity), _iwir_output_wrapper_(frames),
        counterStart, counterStop, counterStep)});
};

let fun002 = fun(ref<ref<array<char,1>>> scene, ref<int<4>> totalFrames, ref<int<4>>
      framesPerActivity, decl ref<file<ref<array<char,1>>>> outFile) -> unit {
  decl ref<collection<collection<file<ref<array<char,1>>>>>> frames =
        (var(undefined(type<collection<collection<file<ref<array<char,1>>>>>>)));
  fun001(_iwir_input_wrapper_(scene), _iwir_input_wrapper_(totalFrames),
        _iwir_input_wrapper_(framesPerActivity), _iwir_output_wrapper_(frames))
  _iwir_atomic_("Convert", "Convert", _iwir_input_wrapper_(frames), "frames",
        _iwir_output_wrapper_(outFile), "outfile");
  _iwir_wf_output_(type<file<ref<array<char,1>>>>, outFile)
};

let fun003 = fun() -> unit {
  decl ref<ref<array<char,1>>> scene = _iwir_wf_input_(type<ref<array<char,1>>>);
  decl ref<int<4>> totalFrames = _iwir_wf_input_(type<int<4>>);
  decl ref<int<4>> framesPerActivity = _iwir_wf_input_(type<int<4>>);
  decl ref<file<ref<array<char,1>>>> outFile =
        (var(undefined(type<file<ref<array<char,1>>>>)));
  fun002(_iwir_input_wrapper_(scene), _iwir_input_wrapper_(totalFrames),
        _iwir_input_wrapper_(framesPerActivity), _iwir_output_wrapper_(outFile))
  _iwir_wf_output_(type<file<ref<array<char,1>>>>, outFile)
};

fun003()
```

can also be applied to an INSPIRE representation of a workflow application. However, compiler transformations for MWRE workflows have different optimizations goals and constraints. Traditional compiler transformations are mostly targeted towards exploiting hardware characteristics and loop optimisation, their only constraint being program correctness. In contrast, MWRE workflows only contain an abstract specification of the workflow structure translated into a WEP by the workflow engine that executes the workflow by traversing it. Furthermore, IWIR and consequently MWRE do not allow any data manipulation outside of atomic activities that are treated as blackboxes and may have more than one implementation. This excludes most traditional compiler transformations which require data manipulations or operate on low-level program constructs or on individual instructions. However, compiler transformations having other optimization goals can be applied to workflows, as described in the following:

- *Reduce the number of executed atomic activities*, where the most effort is spent when executing a workflow. Therefore, this transformation category is the most likely to achieve good results. When applying traditional compiler transformations, atomic activities should be treated as single instructions.
- *Reduce the engine overhead* by reducing the number of composite activities and/or transforming the workflow structure. Since the time spent in the workflow engine is usually a fraction of the total workflow execution there is only limited benefits from such transformations with respect to the execution time. However, optimizations falling into this category may help with the next optimization goal.
- *Improve and/or facilitate scheduling* by modifying the workflow structure so that the scheduler can find better mappings leading to faster execution plans.

A compiler transformation is legal when the resulting program is semantically equivalent to the original program, which means that the transformed program produces the same results as the original program. When applying compiler transformation to a workflow program, we may change the order of atomic activities, or eliminate an individual atomic activity. This poses a problem because, according to the workflow paradigm, atomic activities are regarded as black-boxes without any knowledge of its internals. There could be a hidden state governing the execution of atomic activities, and by applying compiler transformations we could accidentally alter it in a way that causes the transformed program to produce false results.

To solve this problem, we need to relax the black-box principle and provide (limited) information on what happens inside an activity. To achieve this, we use a tag systems which allows attaching tags to activities. For example, we use a *side-effect freeness* tag (the term comes from functional programming indicating functions that always produce the same output for the same input) to indicate that the tagged activities do not have a global state influencing their execution of an atomic activity. The tags are stored inside the activity repository as part of the activity description. All tags added to an activity also need to be valid for all implementations of that activity. Before a workflow transformation is applied, the source-to-source compiler checks the tags of the affected activities to determine whether the transformation can be safely applied.

There are traditional compiler transformations that fall into one of the categories mentioned above, and can be theoretically applied to workflow programs. But there are practical implications that hinder the application of traditional transformations without tailoring them specifially to workflow programs first. To demonstrate the problem let us try to apply an existing compiler transformation to our POV-Ray workflow example. The first step is to analyse the workflow program to identify optimization possibilities. In our example a data-flow analysis would reveal that the input value of the `SceneFile` atomic activity are loop invariant, and therefore we can move this activity out of the loop and place it right before it using a traditional compiler transformation called *loop-invariant code hoisting*, and thus improve performance by having the affected activities executed only once instead of every loop iteration.

Listing 6 Data structures representing the ports of a POV-Ray workflow (we omitted the boolean ``‘‘valid data’’`` flag for each structure component and the structures for the `Convert` activity for brevity reasons).

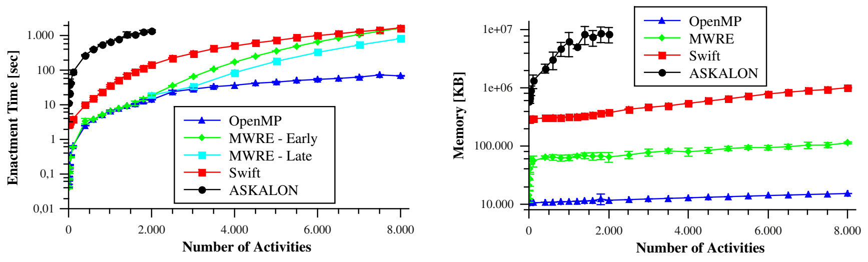
```
struct toplevel_input {
    string string;
    int totalFrames;
    int framesPerActivity;
}
struct toplevel_output {
    string finalMovie;
}
struct SceneFile_input {
    string scene;
}
struct SceneFile_output {
    string povFile;
}
struct PForLoop_input {
    string povFile;
    int totalFrames;
    int numFrames;
    string scene;
}
struct PForLoop_output {
    collection frames;
}
struct RenderTask_input {
    string povFile;
    int startFrame;
    int numFrames;
}
struct RenderTask_output {
    collection frames;
}
```

However, when applying the loop-invariant code hoisting transformation of a traditional compiler, the `SceneFile` activity would be placed before the `parallel for` loop (between lines 11 and 12 in Listing 5). This would break the rule that every composite activity needs to be encapsulated in its own function, which violates the INSPIRE workflow structure outlined in Section 6.1 that can no longer be translated into a MWRE workflow program in the next step. Instead, a workflow-aware version of the loop-invariant code hoisting transformation needs to place the `SceneFile` activity inside the function representing the `toplevel` activity (line 15 in Listing 5).

6.3. Phase 3: MWRE workflow generation

The last phase deals with the conversion the (optimized) INSPIRE program into a MWRE workflow program. It consists of three sub-phases: In the first sub-phase, we generate C data structures (i.e. `structs`) representing the input and output ports of each activity. In the second sub-phase, we replace the functions representing the individual activities and the contained INSPIRE constructs with equivalent MWRE constructs and construct tables representing the complete workflow structure. Finally in the third sub-phase, we analyse the variables and their data-flow to reconstruct the original data-flow and generate the callback functions implementing said data-flow.

We again use our example POV-Ray workflow to describe the translation process. First, we traverse the entire INSPIRE workflow program and look at the signatures of



(a) Enactment time.

(b) Memory utilisation.

Figure 6: Experimental results for the POV-Ray workflow.

the functions and function-calls representing the workflow activities to learn about their input- and output-ports. Using that information we generate the structures (`structs`) representing the ports of each activity (see Listing 6). Afterwards, we again traverse the INSPIRE workflow program, only this time we create the tables representing the workflow structure (see Listing 2). The INSPIRE program structure outlined in Section 6.1 allows us to identify the individual workflow activities in the INSPIRE workflow program, and for each identified activity we create a row in our activity table (Lines 12 – 19 in Listing 2). For each encountered atomic activity we consult the activity implementation repository to retrieve all available implementations and construct a table representing these implementations (e.g. Lines 1 – 4 in Listing 2 lists the available implementations for the `SceneFile` activity). Also we generate the callback function implementing the semantics of composite activities like e.g. the callback function that evaluates the loop counter of our `parallel for` loop. Finally, we analyse the variables of the INSPIRE program and their data-flow to reconstruct the original data-flows and generate the callback functions implementing the found data-flow (see Listing 4).

7. Experiments

The purpose of our experiments is twofold. We first compare in Section 7.1 the perfomance and overhead of our workflow engine with two related state-of-the-art workflow systems. Second, we evaluate in Section 7.2 the overhead of workflow execution with full-ahead scheduling to find out how well full-ahead scheduling performs on shared-memory manycore computers which exhibit completely different characteristics than DCIs. We conducted all the experiments on a multi-core system with four Intel Xeon E7-4870 10-core CPUs at 2.40 GHz with a total of 128 GB of RAM. We used the GNU GCC C/C++ compiler version 4.9.1 to compile the generated workflow programs with the `-O3` optimisation flag.

7.1. Workflow enactment performance and overhead

The goal of these experiments was to verify the callback-driven approach of designing the engine and to give a general estimate of the overhead and scalability associated with maintaining a WEP on shared memory many-core systems. Furthermore we also compared the performance and overhead of *early evaluation* mode (labelled `MWRE-Early`) and *late evaluation* mode (labelled `MWRE-Late`) to get an estimate on the costs of maintaining an as complete as possible WEP.

We conducted the experiments by comparing our engine with two related ones: a fully-fledged workflow environment for DCIs (ASKALON [5]) and a general purpose lightweight workflow scripting language (Swift [26]). To allow a low baseline comparison, we also implemented synthetic OpenMP versions for each workflow application, as OpenMP is the current standard for programming parallel applications on shared memory architectures. We automatically generated the MWRE workflow programs from the IWIR specifications (exported from ASKALON) and the concrete parts from the ASKALON's resource management deployment files using a Java source-to-source compiler that translates IWIR to C++. For each experiment, we recorded the enactment time of the workflow programs in both default and lazy WEP evaluation modes, and their memory utilisation. The enactment time refers to the time spend by the engine while evaluating the WEP and resolving dependencies. Regarding memory, we used the resident size reported by the operating system for MWRE and OpenMP, while for ASKALON and Swift we used the memory statistics delivered by the Java runtime API. Since there is no difference in memory consumption between `MWRE-Early` and `MWRE-Late`, we report a single joint result for both. To obtain a more accurate measure of the workflow enactment overhead and its memory consumption, we replaced the actual atomic activities with dummy implementations that only create empty files to satisfy data dependencies, and recorded the makespan of the workflow executions. We used the minimum completion time (MCT) [14] scheduling algorithm in all experiments as it has low overhead, a low linear complexity, and is resilient to prediction inaccuracies. We use a logarithmic y-axis in all the figures for a better visualisation.

7.1.1. POV-Ray

We used again the POV-Ray workflow with the simple structure presented in detail in Section 4 but without the `SceneFile` activity (see Figure 7).

The results in Figure 6 show that until about 2000 – 3000 activities the performance of MWRE is similar to OpenMP. Above 2000 activities the performance of `MWRE-Early` is significantly worse than OpenMP, and the performance difference increases the higher the activity count. The performance of `MWRE-Late` is slightly better, being similar to OpenMP till about 3000, and after it gets worse than OpenMP but it stays better than `MWRE-Early` showing twice the performance at about 8000 activities. Swift is 50 times worse than MWRE for a low number of activities, but this difference decreases till it has the same performance as `MWRE-Early` and is only twice as worse than `MWRE-Late` for 8000 activities. ASKALON's performance is 70 –120 times worse than MWRE and it can only handle about 2000 activities till it exceeds the memory available to Java.

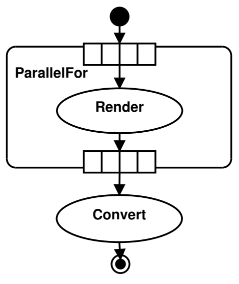
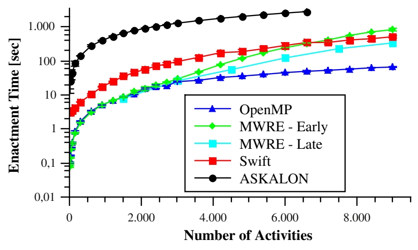
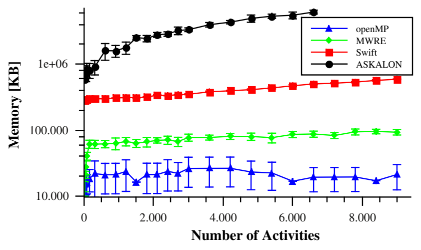


Figure 7: Experimental POV-Ray workflow.

After a rapid increase in the beginning, MWRE's memory utilization remains relatively constant between 90 – 110 MB and slowly increases with the number of activ-



(a) Enactment time.



(b) Memory utilisation.

Figure 9: Experimental results for the RainCloud workflow.

ities. We consider this a low consumption given that real-world activity implementations typically consume gigabytes of memory. In comparison, OpenMP uses between 10 – 15 MB, Swift between 280 – 1000 MB, and ASKALON between 590 – 8500 MB.

7.1.2. RainCloud

RainCloud is a meteorological workflow for weather simulation in mountainous regions using a simple numerical linear model of orographic precipitations [20]. The workflow is currently used by the Tyrolean avalanche service for their daily avalanche bulletins. Its structure, displayed in Figure8, is very similar to POV-Ray, but contains a few additional conditional activities. In our experiments, the conditional activities always evaluate to true so that the PPS and PPF activities are executed.

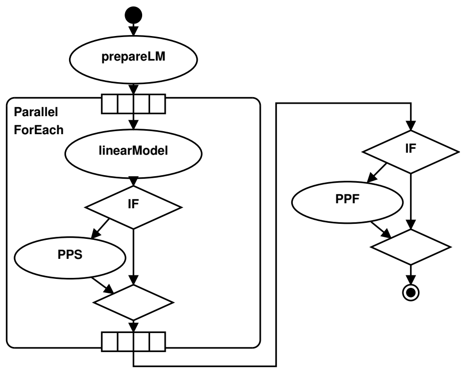
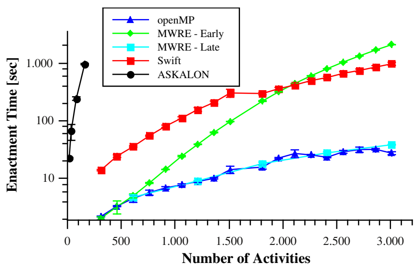


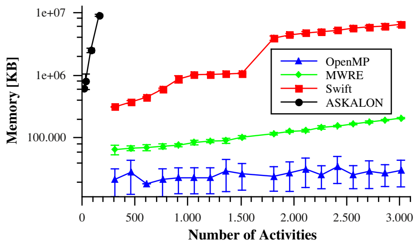
Figure 8: RainCloud workflow.

The results in Figure 9 show that until about 3000 activities the performance of MWRE is similar to OpenMP. Above 3000 activities the performance of `MWRE-Early` and `MWRE-Late` is significantly worse than OpenMP, and the performance difference increases the higher the activity count. The performance of `MWRE-Late` is again significantly better than `MWRE-Early` after 3000 activities. Swift is 40 times worse than MWRE for a low number of activities, but this difference decreases till it shows even better performance as `MWRE-Early` and is only slightly worse than `MWRE-Late` at around 9000 activities. ASKALON's performance starts out 350 times worse than MWRE but at around 6000 activities, where it runs out of memory again, it is only 10 times worse than `MWRE-Early`.

After a rapid increase, the memory utilization of MWRE is relatively constant between 60 – 95 MB and slowly increases with the number of activities. In comparison, OpenMP uses between 16 – 21 MB, Swift between 283 – 590 MB, and ASKALON between 590 – 6200 MB.



(a) Enactment time.



(b) Memory utilisation.

Figure 11: Experimental results for the Montage workflow.

7.1.3. Montage

Montage [9] is a well-known workflow in the scientific computing community created by NASA/IPAC, which stitches together multiple images to create mosaics of the sky. Montage has a rather complex structure briefly sketched in Figure 10, making its enactment the most computationally expensive from all our workflows.

The results in Figure 11 show that until about 600 activities the performance of MWRE is similar to OpenMP. Above 600 activities the performance of `MWRE-Early` is rapidly degrading, even getting much worse than Swift at around 2000 activities. In contrast, `MWRE-Late` always performs similar to OpenMP. Swift starts 7 times worse than MWRE for a low number of activities, but this difference decreases till it is even twice as fast as `MWRE-Early` at around 3000 activities. In this experiment we cannot directly compare ASKALON to the rest because ASKALON begins to exceed available memory at around 160 activities due to the internal implementation of the collection data type, extensively used for activity ports in Montage.

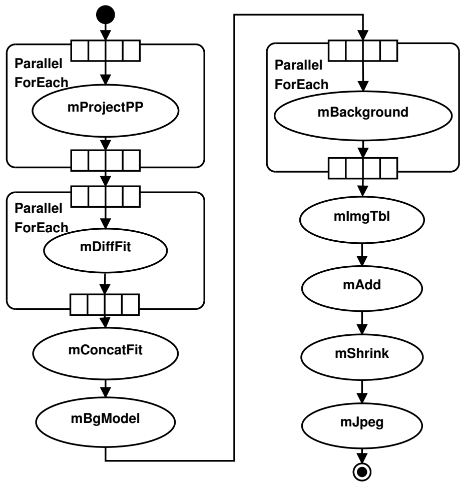
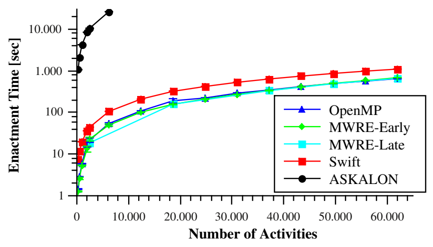


Figure 10: Montage workflow.

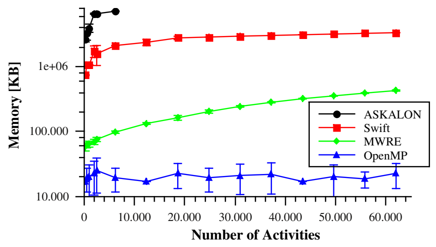
The memory utilisation of MWRE is relatively constant between 65 – 207 MB and slowly increases with the number of activities. In comparison, OpenMP uses between 21 – 30 MB, Swift between 312 – 6500 MB, and ASKALON between 614 – 9000 MB.

7.1.4. Sparselu

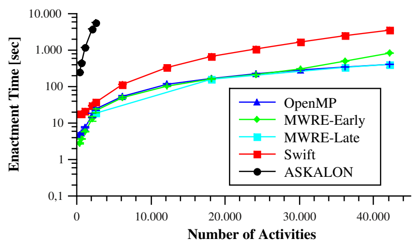
The Sparselu workflow (see Figure 12) adapted from the `sparselu` program from the BOTS benchmark suite [4] does LU factorisation of sparse matrices. The workflow comes in two flavours: with a `for` and with a `while` sequential outermost loop. In



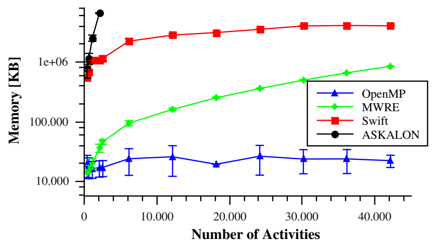
(a) Enactment time (`For-Length`).



(b) Memory utilisation (`For-Length`).



(c) Enactment time (`For-Width`).

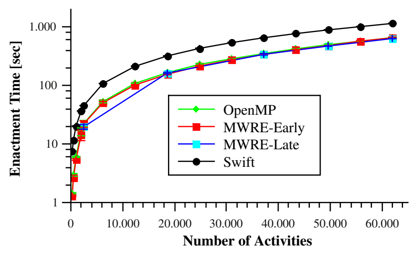


(d) Memory utilisation (`For-Width`).

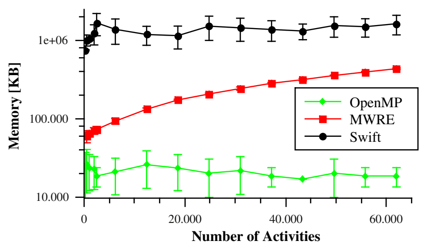
Figure 13: Experimental results for the `For` variant of the Sparselu workflow.

terms of WEP generation in MWRE, there is a big difference between a `while` and a `for` loop. While we can unroll a `for` loop once we know the loop iteration counter and generate the complete WEP, this is not possible for a `while` loop that requires re-evaluation of the loop condition after each iteration with no indication on the total number of iterations. The workflow contains three consecutive `parallel for` loops with the same number of iterations within the sequential outermost loop. The experiments labelled `*-Length` in Figures 14 and 13 only modify the iteration number of the sequential outermost loop (i.e. the length of the workflow), while the experiments labelled `*-Width` only modify the iteration number of the inner `parallel for` loops (i.e. the width of the workflow). In the first case, we increase of total number of activities while keeping the number of activities scheduled at the same time constant, while in the second case we also modify the number of activities simultaneously scheduled.

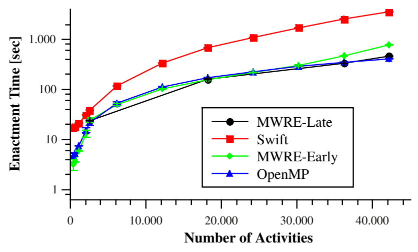
The results in Figures 14 and 13 show that the performance of MWRE is similar to OpenMP most of the time. Also there is no significant difference in performance between the `For` and the `While` variants. However, there is a significant performance difference between the `*-Length` and the `*-Width` variants. The performance of the `*-Length` variants always stays similar to OpenMP, whereas the performance of the `*-Width` variants gets worse than OpenMP at about 30000 activities. The performance of Swift is about 4 – 7 times worse than MWRE in most cases. However, in the `*-Width` variants it is only two times worse than `MWRE-Early` at around 60000 activities. ASKALON on the other hand, is about 500 – 800 times worse than MWRE, and at around 6000 activities it again runs out of memory.



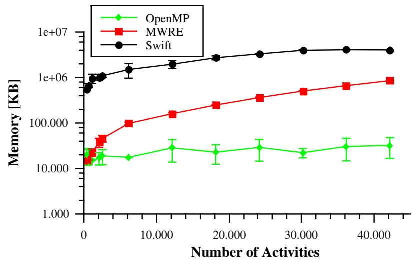
(a) Enactment time (`While-Length`).



(b) Memory utilisation (`While-Length`).



(c) Enactment time (`While-Width`).



(d) Memory utilisation (`While-Width`).

Figure 14: Experimental results for the `While` variant of the Sparselu workflow.

The memory utilisation of MWRE is significantly higher than in the previous experiments (between 15 – 820 MB), but comparable if we take into account the higher number of activities involved. In comparison, OpenMP uses between 15 – 32 MBs, Swift between 550 – 4100 MB, and ASKALON between 770 – 7200 MB.

7.1.5. Discussion

The experimental results show that MWRE has a better performance and lower memory usage than other traditional workflow systems on shared-memory manycore systems most of the time. It even shows similar performance than OpenMP in most cases. However, its scalability is limited and nearly all experiments show that at a certain workflow size the performance of MWRE begins to degrade. This is because MWRE maintains a complete WEP of the workflow in order to facilitate full-ahead scheduling. Maintaining a WEP is a complex stateful operation and at a certain size the costs of maintaining a WEP becomes too high causing poor

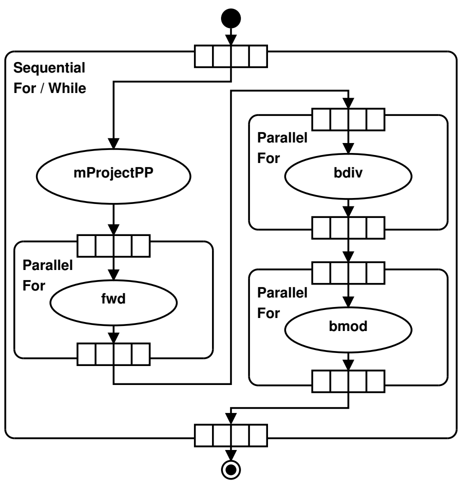


Figure 12: Sparselu workflow.

performance. At which workflow size the costs of maintaining a WEP seems to be individual for each workflow. In our experiments a simple workflow like POV-Ray only shows similar performance to OpenMP up to 2000 – 3000 activities, whereas a more complex workflow like Sparselu can scale up to more than 60000 activities.

One of the factors that influences the scalability is the number of simultaneously scheduled activities. POV-Ray is basically a single parallel loops, which means that nearly all activities are scheduled at the same time. In contrast, Sparselu contains several parallel loops within a sequential loop which means that only a fraction of the total workflow activities are simultaneously scheduled. This is further confirmed by comparing the `*-Length` and `*-Width` variants of the Sparselu workflow. The `*-Length` variants do not change the number of simultaneously scheduled activities, and they show performance similar to OpenMP in all cases. In comparison, the `*-Width` variants do increase the number of simultaneously scheduled activities, and at around 30000 activities their performance begins to worsen.

Another factor limiting scalability seems to be the number of re-evaluations of the WEP caused by our *early evaluation* policy. The most prominent example is the huge gap in the performance of `MWRE-Early` and `MWRE-Late` in the Montage workflow. Montage excessively uses `parallel ForEach` loops together with `Union`-Ports, which means that every finished loop iteration causes a notification of the successor activities that new data is available when `MWRE-Early` is used. In contrast, `MWRE-Late` only informs the successors when the complete loop has finished. Also in most other workflows `MWRE-Late` scales better than `MWRE-Early`, although the performance difference is not that dramatic. This also indicates that there is some optimization potential by finding some middle ground between `MWRE-Early` and `MWRE-Late` and thus reducing the number of re-evaluations.

The results also indicate that there is no significant performance difference between generating the complete WEP from the beginning (i.e. `For` variants of Sparselu) and successively completing the WEP as the workflow execution progresses (i.e. `While` variants of Sparselu).

7.2. Workflow execution using full-ahead scheduling

We believe that full-ahead scheduling is one of the keys to success for workflows on shared-memory manycore computers. We know from DCIs that full-ahead scheduling algorithm produce a better schedule than just-in-time scheduling algorithms in most cases. But workflows on DCIs exhibit different characteristics than parallel programs on shared-memory manycore computers. While Workflows on DCIs usually consist of relatively few, long running atomic activities, parallel programs on manycores consist of a rather large number of short running tasks. Also, on DCIs the workflow engine and consequently the scheduler are implemented as extern services having their own dedicated computing resources, while on manycores the workflow engine and the scheduler have to compete with the atomic activity execution sub-system for computing resources. Because of this the scheduler has to re-evaluate the schedule more often on manycore computers leading to more overhead, while at the same time workflow execution on manycores is much more susceptible to scheduling overhead because of shared computing resources and much shorter execution times of atomic activities.

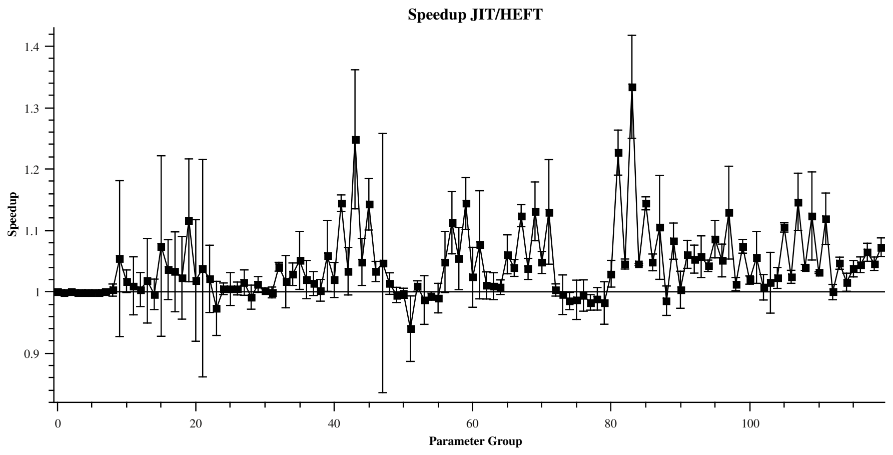


Figure 15: Speedup of HEFT versus MCT.

To find out whether full-ahead scheduling can still deliver its promises despite the above mentioned problems, we executed a large number of workflows exhibiting characteristics of parallel programs for manycores using a well-known full-ahead scheduling algorithm. In order to have a baseline we can compare the results to, we also executed the same workflows using a just-in-time scheduling algorithm. As full-ahead scheduling algorithm we use HEFT [14] and as just-in-time algorithm we use Minimum Completion Time (MCT). HEFT is one of the most widely known full-ahead scheduling algorithms consisting of two phases. In the first phase, it recursively assigns a rank to each activity according to its highest distance to the last activity of the workflow (i.e. B-rank):

$$rank\left(a_i\right) = \overline{w_i} + \max_{n_j \in succ\left(a_i\right)} \left(\overline{c_{i,j}} + rank\left(a_j\right)\right),$$

where $a_i \in A$ refers to the i-th activity, $\overline{w_i}$ is the average execution time of activity a_i across all available resources, $succ\left(a_i\right)$ is the set of activities that immediately depend on a_i, and $\overline{c_{i,j}}$ is the average communication costs of the data transferred between activities a_i and $a_j \in succ\left(a_i\right)$ across all pairs of available resources. In the second phase, it traverses the ranked list of activities and schedules each of them using the MCT algorithm.

To evaluate the workflow execution performance for a large number and variety of workflows, we used an algorithm for automatically generating random workflows [23] by varying the following parameters as presented in Table 1:

- *Average number of activities v* in the workflow;
- *Workflow shape* α by randomly generating the workflow height from a uniform distribution with a mean value equal to $\frac{\sqrt{v}}{\alpha}$, and the width of each level randomly selected from a uniform distribution with mean value $\sqrt{v} \cdot \alpha$;

- *Output degree of an activity* o_i representing the maximum number of successors a workflow activity is allowed to have;
- *Computational heterogeneity* β by randomly generating the execution time for each activity a_i on each resource from the interval $\left(\overline{w_i} \cdot \left(1 - \frac{\beta}{2}\right), \overline{w_i} \cdot \left(1 + \frac{\beta}{2}\right)\right)$, where $\overline{w_i}$ is the average execution time of a_i.

We set the computation to communication ratio parameter to zero, since on shared memory systems the communication is usually much faster than the computation. We set the average activity execution time $\overline{w}$ to five seconds to get a more realistic result for shared memory manycore systems that allow a much finer-grained parallelism compared to DCIs that feature average execution times of several minutes or hours.

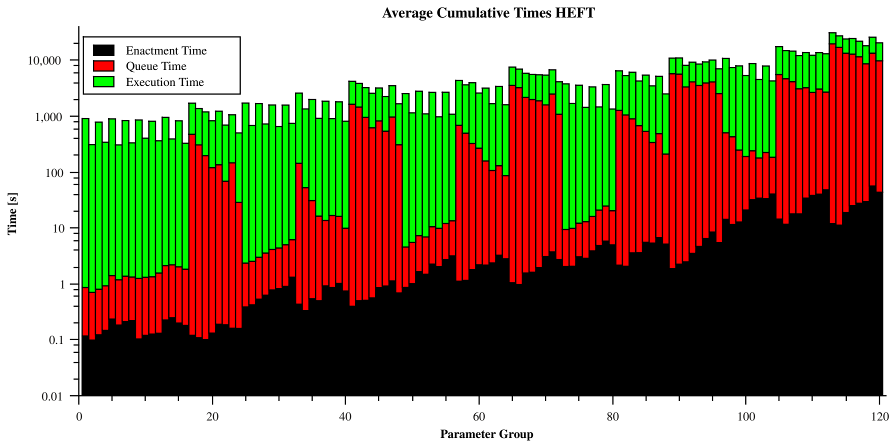
Parameter	*Value set*
v	{ 300, 600, 900, 1200, 2400 }
α	{ 1, 2, 4 }
o_i	{ 2, 4, 8, 10 }
β	{ 1, 1.9 }

Table 1: Random workflow generation parameters.

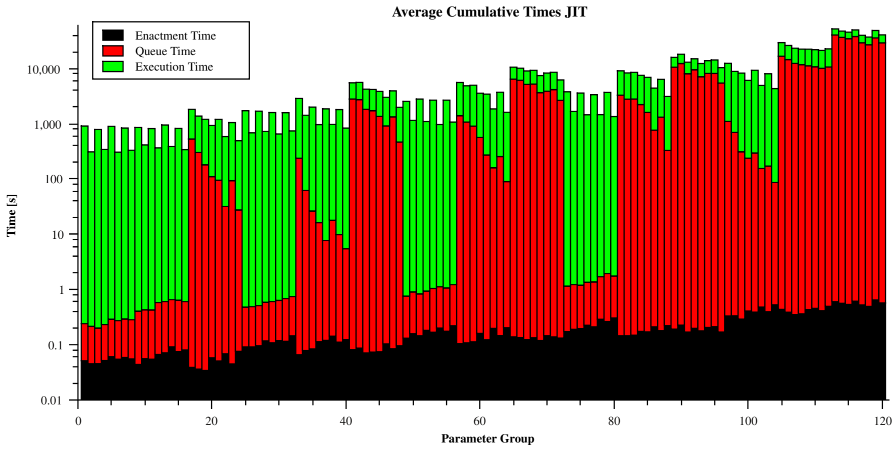
We generated three different workflows for each parameter combination, recorded the average workflow execution time for each parameter combination, and computed the speedup of HEFT compared to MCT. Furthermore, we recorded the workflow enactment time (spent in the workflow engine and the scheduler) together with the queuing and actual execution time for each activity.

Figure 15 shows the speedup results of our experiments. The x-axis represents the parameter groups starting with 1, defined as the cross-product of the parameter values from Table 1. For example, the parameter group 20 represents the parameter values $v = 300$, $\alpha = 4$, $o_i = 4$ and $\beta = 1.9$. It can be seen that HEFT generally exhibits a better performance than MCT for a large number of workflows, with performance gains of up to 40%. The number of activities v does not have a significant impact on performance in our experiment. In contrast, the shape parameter α significantly influence performance. For $\alpha = 1$ (parameter groups 1 – 8, 25 – 32, 49 – 56, 73 – 80 and 97 – 104), we hardly obtain any speedup and get even slowdowns. Only for $\alpha > 1$ we obtain significant speedups. The parameter α controls the height and width of a workflow, and the higher its value the more parallelism a workflow contains. This means that a workflow needs a minimal amount of parallelism for full-ahead scheduling to produce good results. The influence of the parameters o_i and β on the speedup cannot be clearly determined. There are some indications that higher values of o_i and β tend to produce a better speedups, but the differences are smaller than the error distribution. Therefore, they are not significant enough for a conclusion.

Figure 16 with a logarithmic y-axis shows the cumulative sum of enactment times, activity queuing times and activity execution times. The x-axis again represents the parameter groups. In this figure, the number of nodes v significantly impacts the enactment time. The number of nodes grows with the growing parameter group number, as also the enactment time for both HEFT and MCT. The activity queuing time is influenced by both number of nodes v and workflow shape α. The higher the number of nodes v the higher the average queue time. But there are also sudden jumps in the queue time which correspond with the workflow shape α being set to 2 (the smaller jumps



(a) HEFT.



(b) MCT.

Figure 16: Cumulative time analysis of HEFT and MCT.

like e.g. at parameter group 57 – 64) and being set to 4 (the higher jumps like e.g. at parameter group 65 – 72). When we compare the results of HEFT with the results of MCT then we see that HEFT has higher enactment time, which is proportional to the number of nodes *v*, than MCT. Nevertheless, the enactment time is in all instances still much smaller than the sum of queue times and execution times. But HEFT has a lower activity queue time than MCT. When we look at the total times, then the lower queue times more than compensate for the higher enactment times so that in the end HEFT has a better performance than MCT in most cases.

The results clearly show that workflow execution on shared-memory manycores using a full-ahead scheduling algorithm can still gain significant performance compared to just-in-time scheduling, despite suffering from a higher overhead compared to workflow execution on DCIs and manycore computers being more susceptible to overhead. Even with a low average activity execution time of only five seconds, the overhead caused by the WEP generation and full-ahead scheduling algorithm stays in an acceptable range.

8. Conclusion

We described in this paper a new lightweight Manycore Workflow Runtime Environment (MWRE), designed to efficiently enact scientific workflows on modern shared memory manycore computing architectures with a special emphasis on full-ahead scheduling. MWRE is compiler-based and translates workflows represented in the XML-based IWIR specification into an equivalent C++-based workflow program. The workflow program efficiently enacts the workflow as a stand-alone executable by means of a new callback mechanism to resolve dependencies, transfer data, and handle composite activities, rather than high-overhead reflection and type introspection used in existing DCI engines.

We compared the performance and overhead of MWRE's engine with two representative traditional engines: a Java-based one designed for DCIs (ASKALON) and a script-based one designed for manycore architectures (Swift). Our results demonstrate that employing a compiled workflow program tailored to the needs of manycore platforms exhibits a lower enactment time and less memory consumption. In particular, MWRE efficiently handles complex workflows with a high number of activities, and even achieves a similar enactment time to synthetic OpenMP versions for certain types of workflow applications. MWRE performs much better than ASKALON in all situations, and it can also perform better than Swift depending on the workflow. It can even come close to the performance of OpenMP. However, MWRE suffers from limited scalability due to the overhead involved with maintaining a WEP. The scalability limits are highly dependent of the complexity of the WEP and the number of simultaneously scheduled activities. In our experiments, workflows with a small number of simultaneously scheduled activities and a fairly complex WEP scale well up to more than 60000 activities, while other workflows with a very high number of simultaneously scheduled activities only scale up to 2000 – 3000 activities. Furthermore, we experimented with two different WEP evaluation strategies, namely early evaluation and lazy evaluation. The results show that for highly complex WEPs there is a large potential for optimization, but simpler WEPs only slightly benefit from the early evaluation. The memory consumption of MWRE is significantly higher than OpenMP, but within low acceptable limits, and much better in other workflow engines designed for DCIs. We also evaluated the performance and overhead of MWRE using a full-ahead scheduler (HEFT). The results show that full-ahead scheduling approaches, despite suffering from higher overheads than on DCIs and manycore computers generally being more susceptible to overheads, can also gain performance in shared-memory manycore environments up to 40%. The results show that the higher overheads are compensated by a significantly lower queue wait times for workflow activities.

We also evaluated the application of compiler transformations to workflow programs. While they are theoretically possible and also have valid use-cases, we could not find any traditional compiler transformation that can be actually applied to workflow programs as it is due to the specific requirements of their internal representation in the compiler. Any compiler transformation that is intended to be applied to workflows needs to be specifically tailored to these requirements first.

In future research, we plan to further investigate the WEP evaluation modes with the goal to combine the benefits of early evaluation with the performance of late evaluation. Also, we plan to expand our work on compiler transformations for workflow programs by modifying existing compiler transformations to make them compatible and finding new compiler transformations specifically tailored to the needs of workflow programs.

Acknowledgement

This work is supported by the Austrian Science Fund (FWF) project TRP 237-N23: "Workflows on manycore processors" and by the joint FWF – Indian Department of Science and Technology (DST) project No. INT/AUA/FWF/P-02/2013.

[1] D. Barseghian, I. Altintas, M. B. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E. T. Borer, E. W. Seabloom, et al. Workflows and extensions to the kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics*, 5(1):42–50, 2010.

[2] D. Bednárek, J. Dokulil, J. Yaghob, and F. Zavoral. Bobox: parallelization framework for data processing. *Advances in Information Technology and Applied Computing*, pages 189–194, 2012.

[3] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[4] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *International Conference on Parallel Processing*, pages 124–131. IEEE Computer Society, 2009.

[5] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, J. Q. Stefan Podlipnig, M. Siddiqui, H.-L. Truong, A. Villazón, and M. Wieczorek. ASKALON: A development and grid computing environment for scientific workflows. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 450–471. Springer, 2007.

[6] Z. Falt, D. Bednárek, M. Kruliš, J. Yaghob, and F. Zavoral. Bobolang: A language for parallel streaming applications. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 311–314. ACM, 2014.

[7] R. Filgueira, A. Krause, M. Atkinson, I. Klampanos, A. Spinuso, and S. Sanchez-Exposito. dispel4py: An user-friendly framework for describing escience applications. In *Proceedings of 11th IEEE eScience 2015, Munich, Germany, September 1-4*, pages 454–464. IEEE, 2015.

[8] 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*, 22(3):347–360, 2008.

[9] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, 2009.

[10] M. Janetschek, R. Prodan, and S. Benedict. A workflow runtime environment for manycore parallel architectures. In *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science*. ACM, 2015.

[11] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 7–18. IEEE Press, 2013.

[12] P. Kacsuk. P-GRADE portal family for grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23(3):235–245, 2011.

[13] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[14] M. Maheswarana, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, November 1999.

[15] C. Pautasso. Compiling business process models into executable code. *Handbook of Research in Business Process Management*, pages 318–337.

[16] T. Plachetka. POVRAY – Persistence of Vision Parallel Raytracer. In *Proceedings of Computer Graphics International '98*, pages 123–129, 1998.

[17] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[18] K. Plankensteiner, R. Prodan, M. Janetschek, J. Montagnat, D. Rogers, I. Harvey, I. Taylor, Á. Balaskó, and P. Kacsuk. Fine-grain interoperability of scientific workflows in distributed computing infrastructures. *Journal of Grid Computing*, 11(3):429–455, 2013.

[19] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, and P. J. Maechling. Enabling large-scale scientific workflows on petascale resources using mpi master/worker. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, page 49. ACM, 2012.

[20] F. Schüller, S. Ostermann, M. Janetschek, R. Prodan, and G. Mayr. The raincloud project: Harnessing cloud computing for a meteorological application at the tyrolean avalanche service. In *Geophysical Research Abstracts*, volume 15, page 9710, 2013.

[21] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana applications within grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[22] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260 –274, mar 2002.

[23] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[24] J. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[25] M. Wieczorek, A. Hoheisel, and R. Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Generations Computer Systems*, 25(3):237–256, 2009.

[26] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[27] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.

[28] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 95–102. IEEE, 2013.

Matthias Janetschek received his master degree in Computer Science from the University of Innsbruck, Austria, in 2011. He is currently a Ph.D. student at the Institute of Computer Science, University of Innsbruck, Austria. His research in the area of parallel and distributed systems comprises scientific workflows and bringing them to shared-memory manycore computers.

Radu Prodan received the Ph.D. degree from Vienna University of Technology, Vienna, Austria, in 2004. He is currently an Assistant Professor at the Institute of Computer Science, University of Innsbruck, Austria. His research in the area of parallel and distributed systems comprises programming methods, compiler technology, performance analysis, and scheduling. He participated as main investigator in numerous national and European projects. He is currently the scientific coordinator in the Horizon 2020 project ENTICE. He is the author of over 70 journal and conference publications and one book. He was the recipient of an IEEE Best Paper Award.

Shajulin Benedict received his Ph.D. degree in Grid scheduling from Anna University, Chennai. After his Ph.D., he worked as a postdoctoral researcher in the Technical University of Munich under the guidance of Professor Michael Gerndt. Currently, he leads the HPCCLoud Research Laboratory at the St. Xavier's Catholic College of Engineering in India. His research interests include energy-aware scheduling, performance analysis of HPC Cloud applications, HPC application developments, and so forth. He is a University Rank Holder in reward for his academic excellence. He has received two research grants from Germany and two projects from the Department of Science and Technology, India (including the Indo-Austrian project EASE).