Integrated Modeling of Distributed Applications and Infrastructure

Christian Straube, Dieter Kranzlmüller
Munich Network Management (MNM) Team,
Ludwig-Maximilians-Universität München, Germany,
Leibniz Supercomputing Center (LRZ), Munich, Germany
straube@nm.ifi.lmu.de

***Abstract*—The challenge of extreme-scale science requires distributed computing capabilities that enable individual high-end machine to work as a collective whole rather than isolated silos and disconnected islands. Amongst other requirements this requires new modes and models of federation of infrastructures, co-design of distributed applications and infrastructure, as well as better prediction of application/workload execution profiles, namely the time to completeness and resources utilized; these in turn require better estimates of collective resource performance. Better prediction of execution profiles and collective resource performance, as well as the co-design and co-modeling of distributed applications and infrastructure requires well-defined models of both application and infrastructure, as well as their integration. This paper describes a stack of integrated models that to provide the ability to estimate the execution profile of real workloads, and discusses the semantics of these models and how they are integrated. We validate our capabilities by comparing predictions with the actual execution of a bag-of-task-type application.**

***Keywords*-tbd**

I. Introduction

This requires (i) New models for federation of resources, (ii) co-design of applications, (iii) better understanding and prediction of how a range of new and emerging applications will utilize this infrastructure.

It is established that at extreme scales there is a need for co-design of applications and infrastructure. No surprises, that this is true of distributed applications and infrastructure as well. However, the first generation of distributed cyberinfrastructure (DCI) is essentially a consequence of gluing layers together, characterized by unclear interfaces, semantic overlap between layers and functional redundancy and incompleteness...

There are two tracks to resolving the above situation: the first is the design of a Next-Generation Middleware (NGMW) which provides the common layer which removes the impedance mismatch between applications and infrastructure, i.e., makes the co-design possible. The second track involves the co-modeling of applications and infrastructure so as to collectively reason across levels. This paper will focus on the second track, viz., co-modeling of distributed applications and infrastructure.

Currently distributed computing suffers from the ability to predict the execution profile of an arbitrary workload. For example, given a general purpose workload it is difficult to estimate how long it will take to execute, what resources might be available, what subset of those resources might be actually used.

Scope of Paper

Challenge (i) models of different granularity, (ii) semantic incompleteness yet overlap! (iii)...

We need (i) the ability to reason at different levels , (ii) as well as integrated across different levels.

Contribution – Overview, what the paper contributes.

(i) Models of A*, W* and I*: ... (ii) Integration of these models: (iii) Validation with real experiments: Notwithstanding the fact that the semantic scope of the models is restricted, we believe this work provides a conceptual advance in that this is to the best of our knowledge, the first demonstrated ability to reason about how real workloads/applications will perform on real infrastructure.

In order to federate resources that deliver collective well-defined capabilities, the ability to quantitatively estimate and predict functional capabilities of the individual components is critical. Our paper also provides the context in which a model of federation (of capabilities) will be provided, and thus setting the stage for addressing higher-level considerations such as how a given workload should be distributed for a given infrastructure, as well as how resources should be federated to provide optimal execution for a given workload.

Structure –

The paper is structured as follows: the next section outlines models for the application (A*), workload (W*) and Infrastructure (I*) in detail. It then shows how these self-contained, semantically complete models can be integrated to model and predict execution properties of a distributed applications on a HPDC infrastructure. This integration provides the initial ability to estimate (restricted measures) of performance of a given application on a given infrastructure.

Our model is validated by comparing the predicted performance (as measured by the time-to-completion) estimates with the actual time taken on an infrastructure which is modeled. The models are thus not validated individually, but rather the integrated capability to execute a specific application on a given infrastructure; the tools/implementations and the design of experiments to perform the validation are presented in §3, results and analysis are presented in §4.

II. Our Approach

The aim of our work is to realistically model the execution of distributed applications, without any of the complexity of managing/deploying/executing them. As depicted in Figure 1, our approach consists of semantically well-defined layers: an application modeling layer (A*), in which there are multiple models to capture the full breadth and variation of distributed applications; the different models in this layer, typically express the units of decomposition, their interaction and other core application characteristics; a layer that describes two different types of workloads, and their transformation. W* model captures the description of the two workloads, as well as the mapping between them; the first workload type — application defined workload, is mapped onto a set of specific resource consumptions**FIXME: language**; and an infrastructure modeling layer I* which captures the performance determinants of an HPC infrastructure and its provided capabilities.

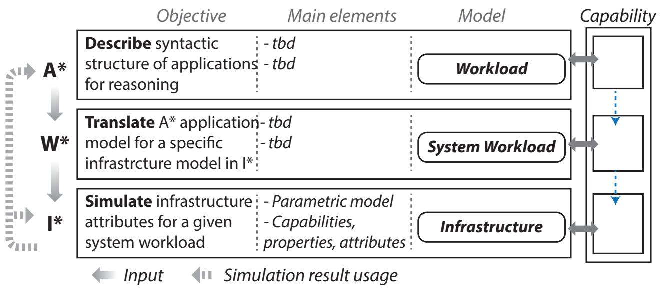
Objective
Main elements
Model
Capability
A*
W*
I*
Describe syntactic structure of applications for reasoning
- tbd
- tbd
Workload
Translate A* application model for a specific infrastrcture model in I*
- tbd
- tbd
System Workload
Simulate infrastructure attributes for a given system workload
- Parametric model
- Capabilities, properties, attributes
Infrastructure
Input
Simulation result usage

Fig. 1. Our approach consists of three semantically well-defined layers, splitting the integrated modeling into three interacting domains

A. The three layers of our approach

A* Application Models – As the modeling stack's uppermost layer it provides the means to represent and to reason about actual applications. A* represents a *set* of application models, because we believe that a single model for all applications and application types is not capable of covering the big variety of contemporary applications, e.g., control-flow and data-flow oriented applications [1].

W* Workload Models – Independent of the applied specific model, all A* models represent an application instance as an *Application Workload (AW)*, which describes the application components and their relationships. The description is at a relatively high level and hence, not necessarily bound to a specific implementation. The formulation of an application workload is thus considered the least common denominator for the diverse A* application models.

The exact semantic and syntactic structure of the AWs is defined by W*, which further also captures different transformations such a workload can undergo while maintaining semantic correctness. Specifically, the transformations covered by W* include the syntactic restructuring of the workload (i.e. its components and relations); the mapping of the AW to a specific implementation; and the derivation of information on the execution properties of the workload. Those transformations are informed by specific implementation and execution choices, and also guided by properties of the target resources, i.e. by explicit infrastructure models. The result of the W* transformations is a *System Workload* (SW), which is much more constrained than an AW, and geared toward a specific execution environment and target infrastructure. Unlike AW, *System Workload (SW)* represents an explicit execution plan for an application instance, including specifics about component distribution and resource consumption estimations.

I* Infrastructure Models – The modeling stack's bottom layer represents the *High Performance Distributed Computing* (HPDC) infrastructure resources that are consumed by an application instance. SWs are mapped to infrastructure components to derive predictions on the applications' execution properties, in particular on its performance. Just like the A* layer, the I* layer represents a *set* of infrastructure models, because of the same reasons. In the scope of this paper, a parametric model that supports what-if analysis [2] is used.

B. Experiments, Results, and Contribution

To re-emphasize: the purpose of the paper, and thus also of the experiments described in this section, is to verify the viability of the approach of end-to-end modeling, via a model stack which spans application level models, over workload transformation models, to models of distributed computing infrastructures. Along those lines, the experiments serve 4 distinct purposes:

Employing multiple different models and their interactions requires a particularly thorough analysis of the general feasibility, accuracy, and productivity of our approach. *Feasibility analyses* addresses the application of a real world scenario, consisting of a scientific application and an HPDC infrastructure – it places real world concepts unambiguously in one of the outlined three layers (cf. Section II-A), and describes them within that layer's model (cf. Figure 1). It stresses made assumptions and assesses their correctness. Finally, it covers the run time behavior of simulations and predictions, to ensure that the models' size or bulkiness do not outweigh their benefits [3], [4]. *Accuracy analysis's* shows that applying extensive abstraction and omitting technical details do not negatively effect the results, but simplify the result interpretation [5]. *Productivity analysis* aims at illustrating that the presented approach achieves the above outlined objectives. The paper will support these analysis types with experiments.

Even if experiments covered only one machine, our approach can be applied for distributed infrastructures because of its very generic definition. For reasoning about infrastructure attributes, like performance/TTC in our experiments, knowing resource utilization per time step is extremely helpful for reasoning. Additionally, we saw that extremely fine-grained models are not too helpful, because it is a very challenging task to gain all information to fill each field. For instance, not having root privileges on one of the experiments machine it was nearly impossible to get all data. And our models are already quite abstract.

A lot of work. Big part is about empirical data gain etc. Rest is about prediction and modeling, but each is focusing on a very specific topic, e.g., or Dramsim2. We provide a) models that abstract sufficiently to a suitable level, to cover a wide range, not focusing on a specific topic. Additionally, we integrate them. Compared to other approaches, our approach is able to describe and predict "extreme scale science" applications and infrastructures.

III. Technical Details

The previous section provided an overview of the objectives, the essential components, and the salient features of our approach and methodology. This section describes the specifics of the models and how they integrate to provide an *end-to-end* approach to understanding how applications can utilize distributed infrastructure.

A. A* Modeling Approach

It is important to distinguish the A* approach from specific application models that are captured by/representative of the A* approach. There are multiple ways to solve a specific problem; the specific solution is often a consequence of many factors, including resources available and ease of implementing changes. It is a non-trivial task to capture all the solutions to a given problem using a single model, just as it is difficult, to capture all possible solutions via a single model of all distributed applications. As a consequence, we eschew a single monolithic all-inclusive model for all distributed applications, in favour of many multiple models which expose the "degree of freedom" of interest and relevance for the problem under investigation. This is a defining feature of the A* approach.

On that background, A* is considered a *set* of models – where each model of A* covers a certain class of application. More formally, the A* approach assumes that (a) a large fraction of distributed applications can be subdivided into reasonable non-overlapping application classes based upon their structural composition; (b) that the execution properties of applications within a given class are comparable; (c) that those execution properties are agnostic to implementation details; (d) and that the application semantics is invariant to the distribution of that application's components. In other words, the assumption is that it is possible to describe applications in a way devoid of both semantic specificity and implementation specificity, while still remaining flexible enough to be able to reason about distribution and execution properties.

Characterizing A Models:* As the uppermost layer of the modeling stack, an A* model provides means to represent and to reason about actual applications. It focuses on the overall syntactic structure of the modeled applications, but *not* on the applications' semantics. The scope of this work is to map a specifically structured application to an infrastructure, but not to address the actual semantic decomposition of the application algorithmics.

To elucidate, consider the following example: an image bokeh and blur emulator operates on a very large image (2D pixel field). The decomposability of the problem into separate, distributable components depends significantly on the blurring algorithm employed, as well as its parameters. The simple approximating approach of using a Gaussian Blur with large radius for example, is equivalent to a series of Gaussian Blurs with smaller radii – which offers reasonable decomposability as the algorithm has well defined locality and thus computational load is uniform over all pixels. In contrast, more 'correct' bokeh algorithms simulate the optical effects of a physical lens and are much harder to decompose, as computational load is less uniform (it may depends on the pixels' lightness, for example), and less local (it may depend on the pixels' depths information that needs to be reconstructed in the first place – which in itself is a non-local algorithm). Our current approach to models on A* level would be to accept whatever decomposablity parameters an application algorithm suggests, and to model the relation of those parameters towards execution behavior of the distributed application.

Individual A* models cover different application classes – but *all* models in A*, model those applications as sets of interconnected components.

The granularity of the modeled application components are given by potential "natural" distribution options – in other words, the expected component boundaries signify potential distribution points. Interconnects between these components thus cover both information and state exchanges between components, and have causal, temporal and spatial relationships. A* models do not describe *how* those interconnects and relations are technically realized (implemented), but they *do* aim to support the reasoning about the applications' execution properties for different properties of those interconnects' implementations. A* models thus apply to classes of applications which have comparable execution properties, even if the actual semantic workload and implementation details differ significantly.

The specific A* model used in the scope of the work presented in this paper is very simplistic, as the paper focuses more on the model *integration* than on the viability of the individual models – this paper assumes a trivially and statically decomposed bag of uniform tasks (similar to the Gaussian Blur variant from the example above).

Limitations: The semantic decomposition of applications is an important determinant for the structure and specifics of application components – but semantic decomposition is explicitly not addressed at this point. Further, only a limited set of relationship types are considered (which for example do not cover transactional relationships). The granularity of the modeled components is limited, and the component properties are not very general, but rather a selection with focus on the A* modeling purpose.

B. Workload Model (W*)

The Workload Modeling layer W* is the mediator between application level (A*) and Infrastructure level (I*). In this sub-section, we outline the main characteristics of the W* approach to workload modeling, discuss the main advantage of this two-state approach and present some challenges/limitations of our approach.

Characterizing W Models:* The W* model is comprised of a semantically well defined application workload (AW), a well-defined system workload (SW) and a set of transformation relationships between these workloads. The AWs description are generated based upon input provided by the application layer; the SW incorporate specific resource/platform information.

W* constrains the application models to output a workload of consistent semantics, i.e., the different applications models can be mapped into the same description of an AW. Thus while A* and an A* model capture the variability in the expressed application properties (such as the chosen implementation, or the parameterized decomposition), that variability is removed by W*. As such, W* describes the transformations of the applications' requirements, toward a complete and explicit description of the expected infrastructure capability consumptions. Furthermore, the application's distribution structure becomes fixed and frozen in the transition to SW, and thus becomes suitable to map to explicit infrastructure resources.

The range of workload transformations which can be covered by W* is limited by the type of components and relations described by the W* modeling language, and by the type of resource requirement properties it supports. Component types currently cover simple executables, but are extensible. Relation types are causal, temporal and/or spatial. In particular for spatial relations, W* supports the description of network channel properties, such as expected latency and bandwidth, or size, number and sequence of exchanged data packets.

Modeling Approach: The transformations from the initial state (AW) to the final state (SW) prescribed by W*, require transformations of several attributes, of which we list three common ones: (i) transformations of the application component structure of the AW to the execution structure of the SW, (ii) transformations of the resource requirements of the AW to expected capability and resource consumptions of the SW, and (iii) mapping to a specific application implementation.

Transformations on the execution structure (i) need to maintain semantic integrity of the application – but can otherwise re-arrange (collate, split, re-order) the components of an AW. More specifically, the transformation can use the parameters for the application decomposition, as defined by the A* model layer. It will generally be useful and in fact necessary to take the properties of the targeted infrastructure into account while making those transformation choices.

The transformations of resource requirements (ii) toward a fully constrained set of expected resource consumptions is a second order transformations, in the sense that it will follow from the result of the structural transformations (i). While that transformation is semantically simple, it is technically difficult to implement (as discussed in *Limitations* below).

The mapping to specific application component implementations (iii) is not very relevant at this point in time, as only few applications can switch between implementation choices. A notable exception are several workflow systems, where the workflow definition indeed focuses on the semantic application tasks, and a mapping to the actual implementation components is left to the system. The work presented in this paper is not addressing this transformation at this point.

Model Assumptions: Capability requirements include compute (number of FLOPs[1]), memory (number of bytes allocated), and Disk I/O (number of bytes read/written). It is assumed that (a) AW components and relations can be mapped to very specific qualitative and quantitative capability requirements; (b) the distribution structure remains static over the runtime of the workload, and (c) that the execution of the workload components is agnostic to the explicit infrastructure resources they are executed on.

Limitations: It is generally difficult to *predict* exact application workload consumptions [**?**]. There are three possible approaches to obtain reasonable prediction: (a) application analysis, (b) application measurements, and (c) human input. Analysing application algorithms and their dependency on input data and execution environment (a) is highly non-trivial, in particular for distributed applications – and is considered to be out-of-scope for our work (see *Halting Problem*). Measurements of application executions (b) can result in truthful data on application workload consumptions, but it is generally costly to perform those measurements in a way that they allow useful predictions for a variety of input data and execution environments. The use of user-supplied data (c) is generally cheap, but inexact – to some extent is a special case of (b) where the user performs the measurements on a subsample of application runs.

The work presented here, which focuses on motivation and validation of the general modeling approach, relies on a simple variant of (b): the performance of the application is measured over a range of input parameters, on one resource, and those values are used during the W* transformation, under the assumption that the profiling results hold true for other resources. That transformation thus also binds the workload to a specific implementation and execution layout (i.e. the one used for the profiling measurements).

C. *HPDC Infrastructure Model (I*)*

In the applied I* model [6], (HPDC) infrastructure components represent hardware entities and are described by three terms: capability, property, and attribute.

An *infrastructure capability* is a qualitatively well-defined low level functionality like data transfer, data storage, or computation, which is employed to select suitable infrastructure components for executing system workload provided by W*. Infrastructure capabilities split the infrastructure component set into *worker* and *communication* infrastructure components, like a core or Intel's QuickPath Interconnect, respectively. Further technical details, like a disk's block size, a core's frequency, or a network interface's Ipv6 address are com-

[1]The term *FLOPs* is used to denote the plural of FLOP, i.e. the plural of *FLoating Point Operation*. This should not be confused with *FLOPs per second*, which are denoted as *FLOPs/second* in this paper.

pletely omitted, since all information required for simulation are provided by one of the three previously discussed terms.

An *infrastructure attribute* is a quantitative metric describing an infrastructure component as black box during system workload execution at a particular time step, e.g., its power consumption or reliability during (system) workload execution. For each infrastructure attribute, there is a generic concept, comprising an identifier, a description, and a dial, e.g., "power consumption in Watt per time step". Looking at programming, the concept could be seen as a "method signature". "method bodies" are implemented by common mathematical equations, which are using infrastructure component properties and characteristics, like the current utilization factor or an infrastructure component's age, as parameters. Listing 1 exemplary implements the previously given attribute concept [7].

Dividing an infrastructure attribute into a generic concept and an implementation enables the definition of different equations for (potentially) each infrastructure component but the same infrastructure attribute. This is especially required by the diversity of existing approaches and their differing benefits dependent on the infrastructure component type, the context, and the objective [8]. Aggregation of infrastructure attributes comprises an attribute concept, an infrastructure component set, and an aggregation rule that is applied on all infrastructure attribute values of the contained infrastructure components.

```
float calculatePowerConsumption() {
   return (currentUtilizationFactor < 50) ? 5 : 16;
}
```

Listing 1. Exemplary implementation of the infrastructure attribute concept power "power consumption"

An *infrastructure property* is a quantitative and system workload execution agnostic metric considering infrastructure components as white boxes at any time step, e.g., describing the theoretically maximum reading and writing throughput of a storage disk. Just as infrastructure attributes are infrastructure properties divided into a generic concept and a realization. In contrast to infrastructure attributes, the realization is achieved by stating a fixed value instead of an equation. These values can be gathered from vendor specifications, benchmarking, trace analysis or any other source for (empirical) data. To each infrastructure component, an arbitrary set of infrastructure property concepts and accordant values can be assigned. Furthermore, it is possible to prepare different infrastructure property values for the same infrastructure component to ease *what-if* analysis.

Infrastructure topology is described as graph, whose nodes are infrastructure components and whose non-directed edges are logical communication dependencies. This means that *every* hardware element, particularly a bus or a switch, is or can be modeled as infrastructure component and hence, as graph node. Modeling communication as component and not as dependency enables the assignment of infrastructure property vectors and infrastructure attribute calculation rules to communication infrastructure components [2], which is required because of two reasons: (i) nearly all infrastructure attributes are strongly influenced by communication aspects, e.g., system performance is built upon not only computing cores and I/O performance, but also communication interconnect [9]; (ii) infrastructures render functionality by a complex qualitative and quantitative component interplay, which hardens identifying the specific contribution of a single component to infrastructure's functionality [10], and hence, infrastructure attribute simulation is required to consider *all* infrastructure components in order to avoid a delusive interpretation. Implicitly defined by the graph, infrastructure components are considered as being *atomic*, like a CPU core. *Composite* infrastructure components, like a compute node or even an entire data center, are described as sub graph.

We assume a "perfect" information situation in terms of (i) infrastructure component dependencies, (ii) infrastructure property values, and (iii) infrastructure attribute calculation rules. In particular, this means that infrastructure component dependencies are known, and that concrete property values and infrastructure attribute calculation rules are available, e.g., a Southbridge's throughput (*infrastructure property*) or a calculation rule to get a SATA controller's energy consumption (*infrastructure attribute*). Gaining the assumed data, e.g., by employing instrumentation and measurement, is a challenging task, especially for integrated components like the above mentioned Southbridge. Nevertheless, there are many existing approaches to address this challenge, e.g., cycle-accurate energy consumption measurements [11], but since we neither provide an exhaustive list of approaches nor present a particular one, we can only *assume* information availability and retrieval feasibility.

Furthermore, we assume that every infrastructure component provides at least one capability or in other words, that to each infrastructure component a specific capability can be assigned. HPDC infrastructure complexity and heterogeneity render a guaranteed assignment statement impossible, but expendability of an infrastructure component without at least one exposed capability allows assuming it.

Finally, we assume that mapping of system workload elements on infrastructure components using the exposed capabilities is possible. In a production real operating system, mapping faces manifold challenges, pursues different goals [12], [13], and is implemented in several ways, like operations research, constraint programming [14], [15], resource allocation constraints [16], or Markov Decision Processes [17]. This complexity and variety cannot be covered by only one modeling approach. Hence, we assume that an abstracting, simplified mapping algorithm is capable of realizing the most important commonalities.

The entire realm of caching is not addressed by the approach's version presented in this paper, but it will be incorporated in Future Work to respect the great impact of caching on several infrastructure attributes, like performance. To ease reasoning, infrastructure attribute values are stored per time step, i.e. a physical time measure that can be defined in a suitable size, e.g., in seconds, hours, or milliseconds.

```
"app_description"   : {
  "type"            : "application_instance",
  "name"            : "mandelbrot",
  "components"      : [
    { "id"          : "master",
      "type"        : ["task", "concurrent"],
      "module"      : "mb_master",
      "parameters"  : "app_id:my_mb_123",
      "properties"  : "mem:8192"
    },
    { "id"          : "workers",
      "type"        : ["task", "concurrent"],
      "cardinality" : "1..10",
      "module"      : "mb_worker",
      "parameters"  : "app_id:my_mb_123",
      "properties"  : "mem:1024"
    }
  ],
  "relations"       : [
    { "id"          : "master-worker",
      "type"        : ["link", "bidirectional"],
      "head"        : "master",
      "tail"        : "worker",
      "properties"  : ["chunk:1024", "nchunks:128"]
    }
  ]
}
```

Fig. 2. **Application Workload Description:** the overall structure of the application is reflected, as is the parameterization of the application decomposition – the `cardinality` parameter of the worker component is such a parameter. The AW does not reference any specific infrastructure resources, nor does it reference a specific implementation of the application's semantics.

```
"app_description"   : {
  "type"            : "workload_instance",
  "name"            : "mandelbrot #001",
  "components"      : [
    { "id"          : "master",
      "type"        : ["task", "concurrent"],
      "exe"         : "my_mb_master",
      "args"        : "--app_id=my_mb_123",
      "properties"  : ["mem:8192", "flops:2.5G"]
    },
    { "id"          : "workers.1",
      "type"        : ["task"],
      "exe"         : "my_mb_worker",
      "args"        : "--app_id=my_mb_123",
      "properties"  : ["mem:1024", "flops=15G"]
    },
    ...
    { "id"          : "workers.8", ... }
  ],
  "relations"       : [
    { "id"          : "master-worker",
      "type"        : ["tcp"],
      "head"        : "master",
      "tail"        : "worker.1",
      "properties"  : ["latency:<30ms", "bw:>10mb/s"]
    },
    { ...
      "head"        : "master",
      "tail"        : "worker.2",
      ...
    }
  ]
}
```

Fig. 3. **System Workload Description:** The SW is now mapped to an explicit implementation (executables are specified), and it features explicit resource consumption information (number of FLOPs etc). The application decomposition parameters are fixed as well (to 8 workers in this example).

D. Layer Integration

The previous sections described the three distinct models, or rather three model sets, which apply to separate layers: applications, workloads and infrastructures. While the individual models within these respective sets can vary significantly in their approach and scope, the sets are nevertheless bound tightly enough to allow for vertical integration. More specifically, both the set of A* models and the set of I* models is defined to use the same concept of a workload as defined by W* – any A* model will *produce* application workloads (AWs) which are usable as input for W* models, and any I* model will *consume* system workloads (SWs) as *produced* by W*.**FIXME: make sure the above is indeed discussed in the different model sections**.

Figures 2 and 3 show exemplary renderings of an AW and a SW, for a simple bag-of-task application, as exchanged through our modeling stack. Both are expressed as JSON documents, which capture the application structure in a flat hierarchy of component descriptions and relationship descriptions.

1) Transformation Rules: W* models allow to reason about different types of operations which transform an application workload (AW) into a system workload (SW). Specifically, these transformations may map the application components to an explicit execution layout, for example by mapping to different implementations of the application, or by choosing specific decomposition constraints toward expected infrastructure properties. Further, the W* transformations may supply additional annotations to the workloads, thus supporting later decisions toward mapping of components to resources, and execution of the components.

In our modeling stack, we focus on the supplemental informations: the application workloads are enriched by adding very specific resource consumption estimates, which are derived from a previous analysis of various application instances' execution profiles. It is those specific information which allow us to later simulate the system workload (SW_{sim}) execution on an I* infrastructure model.

But complementary, we also use other transformation to derive an alternative system workload (SW_{emu}) which maps the application workload to an emulator implementation, Synapses, which can be used to run the workload in controlled fashion on read infrastructures, to compare the model predictions to experiments.

We will shortly discuss Synapse later on in subsection **??**, and will there also support our claim that the applied transformations are truthfully preserving the original application's execution properties i.e. are semantic consistent. 'Semantic consistent' does *not* refer to application semantics (which is in fact lost when transforming the AW towards Synapse based SW_{emu}), but rather to the preservation of the application's execution properties, which represent the *model semantics* we are interested in.

IV. Experiments

We conducted several experiments in order to validate our approach and to gain insights into the considered applications, infrastructures, and modeling challenges. In particular, we were aiming at assessing the approach's accuracy in representing an application instance. This section explains experimental design and methodology, and it describes experiment implementation to foster reproducibility.

A. Methodology and Design

To re-emphasize: the purpose of the paper, and thus also of the experiments described in this section, is to verify the viability of the approach of end-to-end modeling, via a model stack which spans application level models, over workload transformation models, to models of distributed computing infrastructures. Along those lines, the experiments are designed to serve 4 distinct purposes:

1) **(i)** support the assumption that the system workload representation of our application is a valid and precise representation of the application;
2) **(ii)** prove that Synapse (see section III-D1) is a suitable and precise tool to mimic and emulate application resource consumptions;
3) **(iii)** prove that the used model of the infrastructure yields application performance predictions which are compatible with observed application performance; and
4) **(iv)** demonstrate that the prediction of application performance can be used to reason about workload configuration and distribution.

All experiments use the same metric, *time to completion* (TTC), as central property to characterize the applications execution, because (a) it is easy to measure and to compare, (b) it is relevant to application execution in general, (c) it is directly correlated to infrastructure properties in I* (cf. Section III-C), i.e. *Floating Point Operations per Second* (FLOPs/second) and *throughput*, which makes it easier to evaluate and discuss the results in a meaningful way.

Used Application Workloads and Infrastructures: By varying both the workload parameters (amount of computation, memory and disk I/O), and of the infrastructure attributes (resource performance and configuration), the experiments are ultimately designed to support the claim that our modeling stack is able to predict application time to completion in sufficient detail, thus exposing the application's execution properties.

As described in Section III-D, a particular system workload is defined by a set of application components with additional infrastructure attributes describing the expected resource utilization. For our experiments, the application components are assumed to consume three types of resources: compute (a certain number of FLOPs), memory (allocating and writing a certain amount of memory – in MBytes) and storage (writing a certain amount of data to disk – in MBytes). These three types represent very fundamental application activities. We assume no finer grained structure of the application activities, and assume that all three types can run concurrently, thus competing for resources. We further assume that multiple application instances can also run concurrently, again competing for resources. We make no finer grained assumptions on, for example, operating system I/O and thread scheduling, which modern operating systems implement in order to improve overall system performance. The ranges of parameters used for the application workload are listed in table I.

experiment id	**# applications**	**compute** *MFLOPs*	**memory** *MByte*	**storage** *MByte*
WS.C	[1-64]	6000	0	0
WS.M	[1-64]	0	256	0
WS.S	[1-64]	0	0	1000
C[1-16]	[1-16]	[1000-8000]	1000	1000
M[1-16]	[1-16]	4000	[0-1000]	1000
S[1-16]	[1-16]	4000	1000	[0-8000]

TABLE I
Overview of system workload parameters for different experiments. Parameters in square brackets are varied over the specified range. Weak Scaling experiments (*WS.**) operate on a fixed workload size, and measure scaling over a variable number of concurrent application instances – *C*, *M* and *S* stand for compute, memory and storage loads, respectively. The remaining experiments all use a mixed workload, and vary both the number of applications, and one of the workload parameters, thus performing a coarse parameter sweep over the range of parameters suitable to our application.

The different infrastructure configurations used for both modeling and execution are listed in table II, and visualized in figure 4 – they represent a desktop computer system (*boskop*), and two FutureGrid compute clusters (*india* and *sierra*).

B. Implementation

This section describes the implementation of the previously outlined experiment design. Reflecting the different elements of our approach (cf. Section II), the implementation is divided in system workload emulation, infrastructure modeling, and mapping system workload elements onto infrastructure components.

1) System Workload Emulation: As outlined above, the experiments compare the performance of an application executed on an infrastructure with the prediction of a model of the same application on a model of the same infrastructure. The application used for our experiments is a bag of independent tasks, where each task computes a subset of the mandelbrot set, and stores the resulting image to local disk (on `/tmp/`). In order to have tighter control over the actual resource consumption of the executed application, we use a synthetic workload, as implemented by *Synapse*. Describing Synapse itself is not a topic for this paper – but we will quickly motivate the choice and present some supportive evidence that Synapse can truthfully reproduce the execution behavior of our chosen application.

Synapse: Synapse stands for *SYNthetic Applications Emulator* – it is a light-weight, python-based utility which can emulate relatively arbitrary application workloads, by consuming well specified amounts of compute, memory, storage and network resources, in well specified sequences and patterns. The actual workload is rendered in small code snippets in ANSI-C, which are highly portable, and tuned to reproduce the exact same behavior over different resources, operating systems and compilers.

Figure **??** exemplarily shows the runtime behavior for our mandelbrot bag-of-task application (for a single task), and the same information for a Synapse instance which is configured to emulate the exact same workload.

FIXME: AM: add figure

The emulation does though introduce some additional uncertainty – emulation is by definition not perfect. We chose to use an emulated workload for two reasons: (a) to have tighter control over the variation of resource consumption parameters, and (b) for improved reproducibility of the experiments on different infrastructures.

Synapse: The **syn**thetic **ap**plication**s** **e**mulator also interprets SWs, and emulates the application instance thus described. Synapse in the wider sense *is* an application, but one which is tunable to mimic (emulate) any other applications' resource consumption. We use Synapse to compare the model predictions to actual execution of the SWs.

2) Infrastructures: Aiming at a broad and diverse data foundation for analysis and at coverage of different infrastructures and their challenges, we executed the emulated system workload (cf. above Synapse) on three infrastructures, i.e. *India*, *Sierra*, and *Boskop*. Furthermore, we tried to isolate and analyze influencing factors, e.g., resource sharing of FutureGrid systems hardens a comprehensive control of a particular node for experiments, which is not a problem for a common desktop system. Table II provides key specifications of the employed infrastructure and Figure 4 depicts their component graphs[2].

India and *Sierra* are cluster systems in the FutureGrid[3]. India's CPU cores have an integrated memory controller and are directly connected to memory via Intel's *QuickPath Interconnect* (QPI), a high-speed, packetized, point-to-point interconnect [18]. Its disks are connected via SATA 3.0GB/s. Since Sierra is slightly older, it still uses a front-side bus (FSB), which was replaced by QPI since 2008 in Xeon, Itanium, and certain desktop platforms. Compared to the India system, Sierra has a Northbridge and a Southbridge, i.e. the *Memory Controller Hub* (MCH) and the *I/O Hub* (IOH), respectively. *Boskop* is a customary desktop system without any specific HPC or parallel computation configuration.

Infrastructure property values, which are depicted in Figure 4, as well, were gathered from (manufacturer) specifications (S), by measurement (M), or by assessment (A), if no of the other two ways was possible. Measurement was done by

[2]Intel, Xeon, QPI, AMD, and Phenom are registered trademarks

[3]manual.futuregrid.org/hardware.html

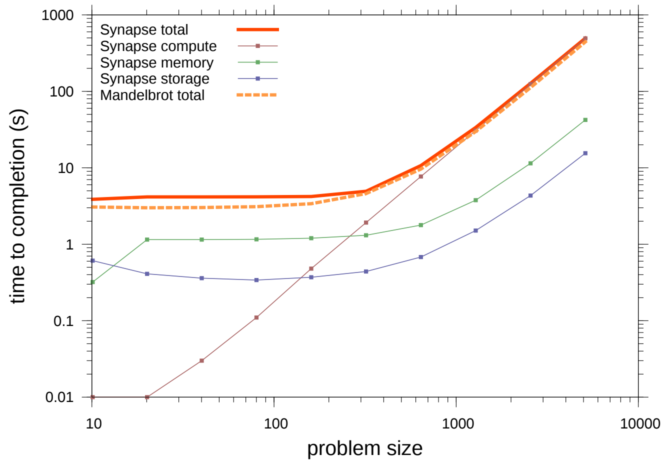

Fig. 5. Verifying Synapse on Boskop: resource consumption metrics from a profiled application are executed as a SW by Synapse – resulting in the shown TTC relations: Synapse is able to truthfully reproduce the application's execution behavior over a wide range of parameters.

using a benchmark or by executing analysis tools like `lspci`. Guided by the chosen measurement of time to completion, we gathered throughput and FLOPs, provided in "amount of data in Gigabyte per second" (GB/s) and "floating point operations per second", respectively. Since throughput is often provided in *Gigatransfer/second* (GT/s), we applied the following formula to get GB/s: Channel width (bits/transfer) transfers/second = bits transferred/second. FLOPs/second were calculated by using different values from GeekBench[4].

3) Scheduling: Executing system workload on an HPDC infrastructure requires scheduling, i.e. mapping tasks on services or (distributed) resources that can execute them. As explained in Section **??**, this is a very complex and versatile task. Especially close to hardware, where experiments are conducted (cf. infrastructure figures), scheduling is well known as a very complex area consisting of several specialized branches, e.g., sharing the memory interface among multiple cores competing for memory bandwidth [19]. We justified the assumption that a simple scheduling algorithm can cover the most important aspects. Consequently, we used a very simple scheduling algorithm: two or more competing tasks get exactly the same amount of infrastructure property, independent of priority, task size, or any other common scheduling parameter.

C. Results and Analysis

We present a number of figures which compare time to completion (TTC) for well defined system workloads as predicted by the modeling stack, versus measured TTC as measured for the Synapse emulation, on a variety of systems (see table **??**).

All graphs present the same type of information: they plot TTC over the number of concurrent applications. Red bold lines denote the experimentally observed TTC, orange bold dashed lines represent the modeled TTC. Additionally, red, green and blue squares represent the TTC for a single

[4]urlhttp://browser.primatelabs.com/geekbench3

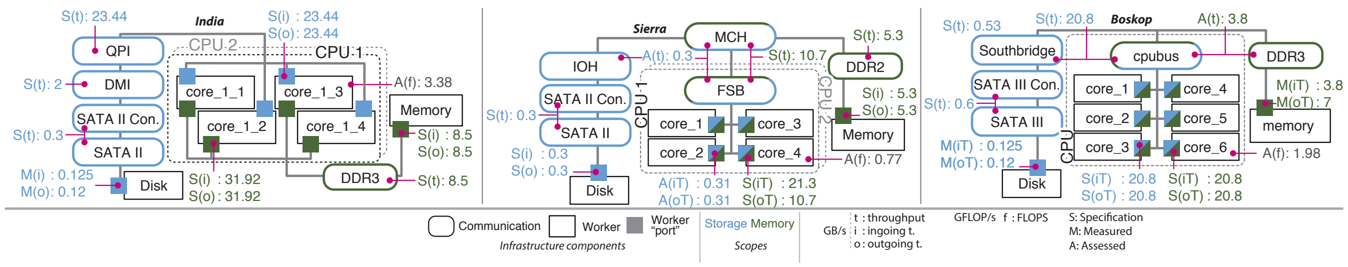

Fig. 4. Topology and property values of the infrastructures employed for experiments

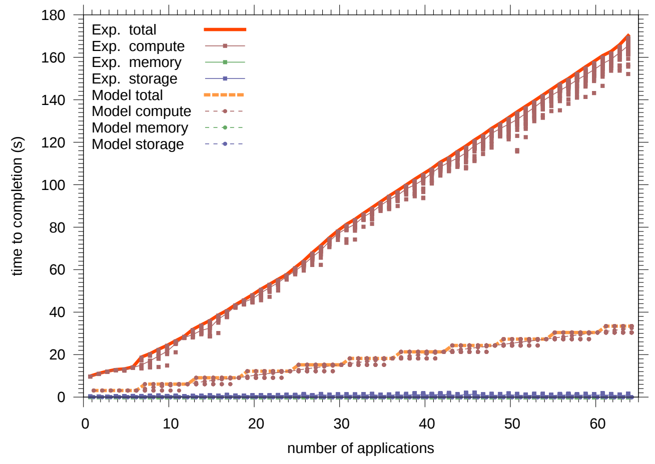

Fig. 6. Weak Scaling of compute load on Boskop: while both model and experiment agree on the overall trends in good detail, we observe an offset by a constant factor – for discussion, see text.

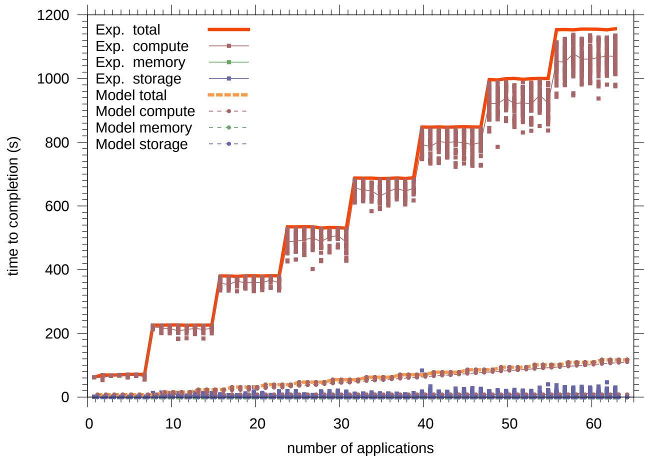

Fig. 7. Weak Scaling of compute load on Sierra: very similar to the Boskop results, we observe the same overall trends, and again the constant offset factor. Note the excellent reproduction of stepping caused by the 8-core CPU configuration.

application's compute, memory and storage subcomponent, respectively, with thin lines of the same colors representing the respective means. Similarly, colored circles and thin dashed lines represent the same information for the modeled application instances.

Weak Scaling Experiments: Figures 6 and 7 show the TTC for 1 to 64 application instances which all consume the exact same amount of resources (6 GFLOPs compute, zero memory I/O and zero disk I/O), for Boskop and Sierra, respectively. Both graphs convey the expected results: constant performance as long as application instances can be scheduled on a empty core, then a linear increase of TTC once all cores are used, etc (Boskop has 6 cores, Sierra 8). The individual experiments (i.e. each data point) is running on a random node on Sierra, as assigned by PBS – but as all nodes feature identical hardware configurations, the data remain predictable and smooth, with very little inter-node noise. This is observed for both compute and memory loads (no memory graph shown) – for discussion on disk I/O, see further below.

The graphs show some amount of noise for the Synapse applications – this is mostly attributed to noise introduced by the kernel scheduler (also ignored in the model – we observe frequent and continuous relocation of the application threads across the different cores), and to the background OS load (which is ignore by the modeled data) – we observe a larger background load on Boskop, up to 15%, which runs a desktop

System	Type	CPU	Chipset	Memory	Disk
India	IBM iDataPlex dx 360 M2	Intel Xeon X5570@2.66GHz	Intel X85	DDR3-1066	HGST Ultrastar A7K1000
Sierra	IBM iDataPlex dx 340	Intel Xeon L5420@2.5GHz	Intel 5400	DDR2-667	WDC WD1600YS-23S
Boskop	Desktop system	AMD Phenom II 1055T@2.8GHz	AMD 890GX, AMD SB850	DDR3-1333	HGST Deskstar 7K1000.C

TABLE II
OVERVIEW OF PARAMETERS USED TO CONDUCT DIFFERENT EXPERIMENT RUNS OF SYSTEM WORKLOAD EMULATION

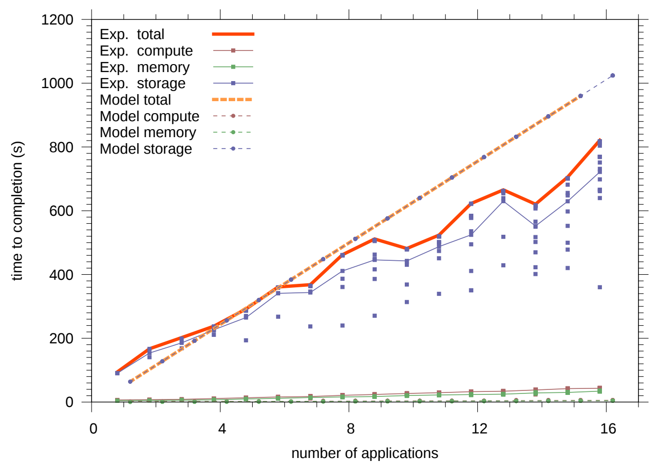

Fig. 8. Storage-heavy applications Boskop

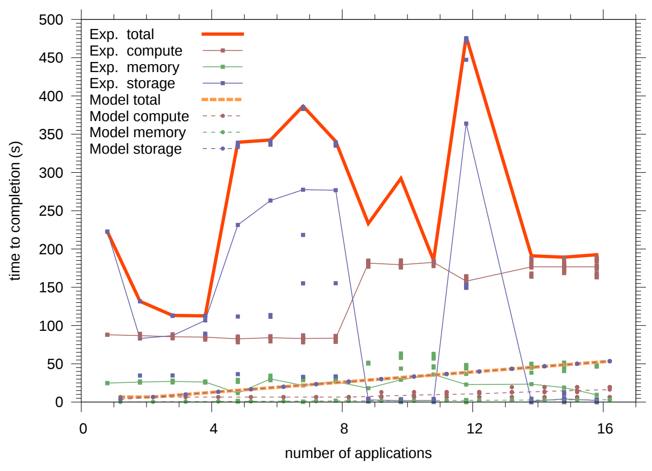

Fig. 9. Compute-heavy applications on Sierra

environment, which seems to lead to higher noise in the data points overall.

While the memory and storage sub-timers are exactly zero in the modeled data points, as expected, we see a small contribution in the experimental data points – this to some extent represents an systematic experimental error, caused by simply creating and destroying the idle workload threads. That contribution increases as the overall system load increases, i.e. for many applications, or on resource starvation (CPU, memory), but is always small compared to the overall TTC.

Figure 8 exemplarily shows the result of a weak scaling experiment with a storage dominated workload – each application instance consumes 5 GFLOPs, allocates and writes 1 GByte memory, and writes 1 GByte to the local disk. The overall TTC is dominated by disk I/O, and reproduces nicely the data predicted by the model. The model and experiment values are in better agreement than for the compute dominated loads, because (a) the assumed peak performance of the disk is very closer to the achievable performance, and the synthetic workload can utilize that performance very well. We cannot fully attribute the super-optimal performance to caching effects, as that effect *increases* with larger experiments, where data should not easily fit into any caches anymore. Instead we assume that the peak disk performance is given for a single writing process, and that concurrent processes are able to write more efficiently, in total. We were not able to find any specific information on how disks are *expected* to behave on large concurrent writes, so the model assumes a trivial constant disk performance.

Noise is increasing *significantly* for concurrent disk access – possibly owed to very quick cache invalidation, and due to the kernel I/O scheduler operation – neither are covered by our model.

The memory and compute timings increase linearly, as expected, and are, as discussed above, off the modeled data by a constant factor.

An exemplary graph for a compute-heavy experiment is shown in 9, where an increasing number of concurrent application instances (1 to 16) consume 5 GFLOPs, allocate and write 1 GByte of memory I/O, and write 1 GByte to a local disk. We see the expected stepping after 8 application instances, due to the inset of core satiation. Most importantly, we see that the overall TTC is dominated by the disk I/O load – apart from some data points where disk I/O actually failed do to full disks on the respective node.

Importantly, the compute timings are again very independent on the actual node running the experiment (a random node is used for each data point) – but the disk performance varies very widely over different nodes. We found it in all experiments which involve even smaller amounts of disk I/O, that the variation in disk performance over the cluster made the overall TTC prediction very difficult, due to very large noise and frequent failures. That was observed on Sierra, and even more so on India. Where the disk I/O load was very low, the results were much closer to what was expected from the bare compute and memory observations, and was also in sync with our results on Boskop.

We are not fully sure what causes the large noise in disk I/O over different cluster nodes – all nodes are using the same disk configuration, and the same overall hardware configuration. Our guess is that this can be attributed to background maintenance tasks which perform frequent but small disk writes, such as logging, or caching for the shared cluster file systems (which were not used by our experiments). We intent to investigate this effect further, but given the observed amount of noise it seems unlikely that better predictions of disk I/O performance are possible on these clusters.

V. Related and Relevant Work

At a first glance, there are several approaches and tools addressing the paper's topics and some of them are even time-tested and mature. Unfortunately, these approaches neither integrate nor interface, two vital requirements to achieve our objectives (cf. Seciton II). Additionally, they use a different methodology (Section V-A) or apply a different focus (Section

V-B). Nevertheless, there is a plethora of work that supports our approach, e.g., by providing exhaustive data.

A. Methodology

In general, there are three prediction methodologies, i.e. (i) application code analysis in terms of code inspection or algorithmic complexity, (ii) extrapolation of and statistical calculations on empirical data, and (iii) simulation.

Code analysis is known as being a difficult and cumbersome task, requiring deep insights about the considered application. Even if this methodology is very helpful to improve a particular application, e.g., by removing runtime flaws, it is rather unsuitable for us, since we are aiming at integrating modeling of the entire stack (cf. Section II).

Retrieving *empirical data* can be done by trace analysis or benchmark execution. Even if there are low-intrusive approaches for *trace analysis*, like *Darshan*, which aims at capturing an accurate picture of application I/O behavior [20], there is unavoidable overhead and instrumentation effort, and data is only valid for the considered application. Both drawbacks render this approach unsuitable for our objectives. There are several tools for *benchmark execution*, distinguished by their extend and focus. In every way, they use proxies representing a group of applications, which requires a mapping and interpretation of results for a particular application. Even if this challenge is supported by classifications, like the one provided by H. Shan et al. [21] who did a scientific workload survey from the National Energy Research Supercomputing Center (NERSC), this mapping causes unavoidable bias. Parametric benchmarks cover a broad range of application behaviors [21] without requiring the mapping, but using a big set of parameters, like IOR [22], making their usage very complex.

Both ways to retrieve empirical data in common is the imperative to physically execute software at least once. This requires either a (costly) physical copy of the considered HPDC infrastructure completely or partly under laboratory conditions, or the execution on a productive system, which is rarely possible because of HPC infrastructures' steady usage [23], [24]. Furthermore, requiring physically executions hardens "what-if" analysis, because each parameter change, e.g., application component distribution or infrastructure component properties, requires a new execution. This is not true for our approach (cf. Table I). Summarizing, despite the extreme high value of using empirical data to inspect a certain application, simulation is the weapon of choice to achieve our objectives.

B. (Predictive) Simulation

In the realm of the previously selected (predictive) simulation the situation is rather the same, which means there are several mature approaches, but none of them is applicable for our objectives: the approaches majority is narrowed to a partition of our approach, e.g., *Dramsim2* [25] is a time-tested tool for describing cycle accurate DDR2/3 memory systems, but it does not cover other infrastructure components. The *Structural Simulation Toolkit* (SST) [24] covers a bigger set of infrastructure components, but it does not provide an interface for (system) workload definition and consideration. Obviously, there are lots of other unnamed tools, but to the best of our knowledge, they are very mature in accurately copying physical real world, but not suitable out-of-the box for our objectives. Unfortunately, they do not provide interfaces for integration either, so adaption would be very costly and time-consuming.

C. Modeling

Supporting simulation, we consider modeling approaches for the three layers (cf. Figure 1).

1) Workload Description – A:*

2) System Workload Description – W:* Reasons, why other approaches are not suitable.

(i) *Focusing on domain-semantics*, which are neglected as explained in Section **??**. An example for this class is *Triana* [26], providing more than 400 tools for data analysis and manipulation [27], e.g., audio analysis [28] and image processing. Another example is *Kepler* [29], providing high flexibility in semantics representation, e.g., each table column can be exposed as port [27]. Even if (nearly) all of these approaches and projects are mature, they are completely or partly lacking functionality for system workload calculation for PMEA.

(ii) *Focusing on one aspect of system workload calculation*, like workflow definition or scheduling. Kepler, for instance, provides graph-based modeling functionality to create abstract scientific workflow models [30], and *Pegasus* (*Planning for Execution in Grids*) [31] maps these models onto distributed resources. Even if the (hardware-independent) definitions of Kepler, Wings [32], or other specialized projects, like the *Java Commodity Kit* (CoG) Karajan [33], or the functionality of Pegasus could contribute to an abstract scientific application model (cf. Section **??**) or to a scheduling implementation in the Utilization Mapper, respectively, heavy transformation and adaption would be required or important characteristics, like the granularity level, wouldn't be sufficient.

(iii) *Not fulfilling requirements*, e.g., applying a control-flow like Pegasus, Swift [34], or Triana [35].

(iv) *Focusing on HPDC infrastructure subsets*, like HPC infrastructures. There is a very small set of approaches targeting infrastructure attributes, like performance [36], [37]. But none of these approaches considers system workload generically to use it in PMEA.

3) Infrastructure Description – I:* Searching for a model to be used completely or partly as extension base, we analyzed several of the manifold (meta) models and languages to describe a network, computer system or a distributed system, e.g., the SNMP, the *Grid Laboratory Uniform Environment* (GLUE) [38], or the *Common Information Model* (CIM) [39], just to mention some of the most prominent ones. Analysis result is that none of the models cover infrastructure property and especially infrastructure attribute description sufficiently. Extending one of these models in a way that the result would cover all elements required by our approach would cause an immense overhead. For instance, CIM contains lots of hardware related details, like `MediaAccessDevice.MaxBlockSize`. Additionally, most of the classes mix up hardware details and infrastructure properties, e.g., the CIM class `NetworkAdapter` describes `PermanentAddress` (hardware detail) and `MaxSpeed` (infrastructure property). In summary, the benefits of extending an existing model would be outweighed by the huge overhead.

VI. Discussion

Future Directions

The next-generation of distributed applications must be supported by middleware that supports well-defined functional *capabilities* without regards to the specific underlying technology employed to provision the capabilities and how they are delivered. How infrastructure can be federated to provide invariant capabilities remains an important question.

In the future, we will extend our models to support the ability to predict which resources to utilize (and why), as well as enable reasoning on both the workload and application levels to discern optimal configurations.

Acknowledgment

We would like to acknowledge Mark Santcroos and Matteo Turilli for helpful discussions and initial scoping. Santcroos contributed to discussions on application modeling, and Turilli towards infrastructure modeling; both contributed towards better understanding of workloads (W*), which is still work in progress.

References

[1] D. Woollard et al., "Scientific Software as Workflows: From Discovery to Distribution," *IEEE Softw.*, vol. 25, no. 4, pp. 37–43, 2008.

[2] C. Straube and D. Kranzlmüller, "A Meta Model for the Analysis of Modification Effects onto HPDC Infrastructures," in *Submitted to 27th Intl. Conf. on Architecture of Computing Systems (ARCS'14)*, 2014.

[3] S. Robinson, *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, Ltd., 2004.

[4] S. Pakin, "The Design and Implementation of a Domain-Specific Language for Network Performance Testing," *IEEE Trans. Parallel Distrib. Syst*, vol. 18, no. 10, pp. 1436–1449, 2007.

[5] L. Chwif et al., "On Simulation Model Complexity," in *Proc. of the IEEE Winter Simulation Conf.*, 2000.

[6] C. Straube and D. Kranzlmüller, "An Approach for System Workload Calculation," in *Proc. of the 12th Intl. IASTED Conf. on Parallel and Distributed Computing and Networks (PDCN'14)*, 2014.

[7] T. Talbot and H. Davis, "Verizon NEBS TM Compliance: Energy Efficiency Requirements for Telecommunications Equipment," Verizon, Tech. Rep. VZ.TPR.9205, 2011.

[8] C. Straube and D. Kranzlmüller, "Model-Driven Resilience Assessment of Modifications to HPC Infrastructures," in *Proc. of the 6th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids in Conjunction with Euro-Par 2013*, 2013.

[9] D. Chen et al., "Looking Under the Hood of the IBM Blue Gene/Q Network," in *Proc. of the Intl. ACM/IEEE Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12)*, 2012.

[10] M. Al-Mashari and M. Zairi, "Creating a Fit Between BPR and IT Infrastructure: A Proposed Framework for Effective Implementation," *Journal of Flexible Manufacturing Systems*, vol. 12, no. 4, pp. 253–274, 2000.

[11] N. Chang et al., "Cycle-Accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI," in *Proc. of the Intl. Symposium on Low Power Electronics and Design (ISLPED'00)*, 2000.

[12] J. Yu et al., "Workflow Scheduling Algorithms for Grid Computing," in *Proc. of the Metaheuristics for Scheduling in Distributed Computing Environments*. Springer, 2008, vol. 146, pp. 173–214.

[13] M. Wieczorek et al., "Comparison of Workflow Scheduling Strategies on the Grid," in *Proc. of the Parallel Processing and Applied Mathematics*. Springer, 2006, vol. 3911, pp. 792–800.

[14] J. Jaffar and M. Maher, "Constraint Logic Programming: a Survey," *The Journal of Logic Programming*, vol. 19/20, pp. 503–581, 1994.

[15] J. Blazewicz et al., *Scheduling Computer and Manufacturing Processes*. Springer, 2001.

[16] P. Senkul et al., "A logical framework for scheduling workflows under resource allocation constraints," in *Proc. of the 28th Intl. Conf. on Very Large Data Bases (VLDB'02)*, 2002.

[17] J. Yu et al., "Cost-based scheduling of scientific workflow applications on utility grids," in *Proc. of the 1st Intl. Conf. on e-Science and Grid Computing*, vol. 3648. Springer, 2005.

[18] "An Introduction to the Intel®QuickPath Interconnect," Intel Corporation, Tech. Rep. 320412-001US, 2009.

[19] D. Kaseridis et al., "Minimalist Open-Page: a DRAM Page-Mode Scheduling Policy for the Many-Core Era," in *Proc. of the 44th Intl. ACM/IEEE Symposium on Microarchitecture*, 2011.

[20] P. Carns et al., "24/7 Characterization of Petascale I/O Workloads," in *Proc. of the IEEE Intl. Conf. on Cluster Computing and Wkshp.s (CLUSTER'09)*, 2009.

[21] H. Shan et al., "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *Proc. of the ACM/IEEE Conf. on Supercomputing (SC'08)*, 2008.

[22] R. Hedges et al., "Parallel File System Testing for the Lunatic Fringe: the Care and Feeding of Restless I/O Power Users," in *Proc. of the 22th IEEE Conf. on Mass Storage Systems and Technologies*, 2005.

[23] S. Laan, *IT Infrastructure Architecture – Infrastructure Building Blocks and Concepts*. Lulu Press, Inc., 2011.

[24] A. Rodrigues et al., "The Structural Simulation Toolkit," *ACM SIGMETRICS Performance Evaluation Review - Special Issue on the 1st Intl. Wkshp. on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS'10)*, vol. 38, no. 4, pp. 37–42, 2011.

[25] P. Rosenfeld et al., "DRAMSim2: A Cycle Accurate Memory System Simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[26] D. Churches et al., "Programming Scientific and Distributed Workflows with Triana Services," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1021–1037, 2006.

[27] E. Deelman et al., "Workflows and e-Science: An Overview of Workflow System Features and Capabilities," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528–540, 2009.

[28] I. Taylor et al., "Distributed Audio Retrieval using Triana (DART)," in *Proc. of the Intl. Computer Music Conf. (ICMC)*, 2006.

[29] I. Altintas et al., "Kepler: an Extensible System for Design and Execution of Scientific Workflows," in *Proc. of the 16th Intl. Conf. on Scientific and Statistical Database Management*, 2004.

[30] J. Yu and R. Buyya, "A Taxonomy of Scientific Workflow Management Systems for Grid Computing," *Journal of Grid Computing*, vol. 3, no. 3, pp. 171–200, 2005.

[31] E. Deelman et al., "Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[32] Y. Gil et al., "Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows," in *Proc. of the 19th Annual Conf. on Innovative Applications of Artificial Intelligence (IAAI)*, 2007.

[33] G. Laszewski et al., "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8, pp. 645–662, 2001.

[34] Y. Zhao et al., "A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data," *SIGMOD Records*, vol. 34, no. 3, pp. 37–43, 2005.

[35] X. Fei and S. Lu, "A Dataflow-Based Scientific Workflow Composition Framework," *IEEE Trans. Serv. Comp.*, vol. 5, no. 1, pp. 45–58, 2012.

[36] L. Carrington et al., "A Performance Prediction Framework for Scientific Applications," in *Proc. of the Computational Science (ICCS'03)*. Springer, 2003, vol. 2659, pp. 926–935.

[37] ——, “A Framework for Application Performance Prediction to Enable Scalability Understanding,” in *Proc. of the Scaling to New Heights Wkshp.*, 2002.

[38] S. Andreozzi et al., “GLUE Specification v. 2.0,” Open Grid Forum, Tech. Rep. GFD-R-P.147, 2009.

[39] DMTF, “Common Information Model (CIM) Metamodel (Specification),” Distributed Management Task Force (DMTF), Tech. Rep. DSP0004, 2012.