Towards Integrated Modeling of Distributed Applications and Infrastructures

Christian Straube*[0], Andre Merzky†[0], Shantenu Jha†[1], Dieter Kranzlmüller*
* Ludwig-Maximilians-Universität München, Germany, Leibniz Supercomputing Center (LRZ), Munich, Germany
† RADICAL, Rutgers University, Piscataway, NJ 08854, USA

***Abstract*—Currently it is not possible to reliably predict the execution profile of a general-purpose application on distributed infrastructure; exact numbers will vary, but estimates of time-to-completion could be off by an order of magnitude, or more. The reasons for this unsatisfactory state-of-affairs is complex and defy simple reductions. Contributing factors include the absence of general-purpose conceptual and quantitative models of different layers of the stack, and their integrative utilization. Consequently this results in limited ability to reason about exeuction profiles – including estimates of performance, suitabilty of resources etc. This mirrors the fact that existing distributed computing is characterized by gluing together different components. We describe a hierarchy of models – application, workloads and infrastructure, that are integrated to provide the ability to estimate the execution profile of real workloads. We discuss the semantics of these models, as well as their integration. We validate both the individual models as well as their integrative end-to-end capabilities by comparing our predictions with the actual execution of a bag-of-task-type application. Furthemore, this paper outlines an emerging research agenda, and initial important progress towards this objective, that is, to the best of our knowledge we believe we have provided a first end-to-end estimate of a real workload on actual high-performance distributed infrastructure.**

I. Introduction

One of the several challenge of extreme-scale science, is the requirment of distributed computing capabilities that enable individual high-end machine to work as a collective whole rather than isolated silos and disconnected islands. For example, science problems that require the distributed and in-situ analysis of data, ultimately require that the computing resources be logically aggregated. Amongst other requirements this requires better prediction of application workload execution profiles, namely the time to completeness and optimal resources to be utilized, which in turn require integrated modeling of distributed applications and infrastructure, as well as better estimates of collective resource performance.

However, the first generation of distributed cyberinfrastructure is essentially a consequence of gluing layers together, characterized by unclear interfaces, semantic overlap between layers and functional redundancy and incompleteness [**?**]. Partly as a consequnce, but also due to the lack of adequate conceptual models, distributed computing currently suffers from the inability 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.

We address this gap head-on, viz., we aim to provide the first approach that attempts to integrate models of distributed applications with models of high-performance and distributed infrastructure, with the explicit intent of being able to estimate relevant performance metrics (such as time-to-completenss) of applications that are representative of the type that use production distributed cyberinfrastructure. Needless to say, it is critical to establish the scope, granularity and some basic assumptions that underscore and permeate this paper.

Consistent with Ref [**?**], we define distributed applications are those that (i) need multiple resources or (ii) would benefit from the use of multiple resources; examples of how they could benefit include increased peak performance, throughput, and reduced time-to-solution or reliability. Thus the models of the applications we consider, as well as the specific applications that we use as the basis of our experiments and to validate our models — a Mandelbrot bag-of-tasks, is by this definition a classic distributed application. Our approach (and constituent models) are 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.

Before discussing the scope of the paper, some explaination about the title of the paper is in order. We unambigiously establish the integration of models of applications and ifrastructure. In this paper we do not actually perform experiments that utilize multiple resources concurrently, however, our experiments and models can be extended to incorporate multiple concurrent resources; we do however, capture performance over distinct heterogeneous resources, a necessary (but not sufficient) condition for distributed infrastructure. More important than the specific experiments, is the consideration that we believe we have set up a framework that has all the logical components to model and simulate distributed applications execution in a distributed environment via the decomposition of the workload onto different resources. This will form the basis for future work, and hence our work is entitled, "Towards Integrated Modeling of Distributed Application and Infrastructure"

Scope: In this paper, we present semantically complete and independent models of workload (W*) and infrastructure (I*).

[0]These authors contributed equally to the work.

[1]Contact Author: *shantenu.jha@rutgers.edu*

We also introduce A* – a suite of models, as an approach to capture the broad range of distributed applications; it is worth noting we do not discuss specific application models but focus on the sailient features of our approach. A major contribution is the integration of the models and their design so as to support end-to-end integration. Furthmore, we introduce a synthetic application emulator (synapse), which allows the translation of an application instance into a workload described by W* and emulate its resource consumption/utilization. We believe synapse will be a useful tool for the community interested in understanding resource consumption of applications. 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 demonstrate performance estimates as well as provide reason about how real workloads/applications will perform on real infrastructure. Last, but not least, our work sets the stage for investigating how, given a workload and a set of resources, the workload should be distributed to provide optimal execution for a given workload.

There is a classic trade-off between accuracy and prediction on the one-hand, versus general-purpose on the other hand. Different levels of our modeling stack have been designed with different design points in consideration. The application modeling approach (A*) is definitely geared towards generality; in fact it a suite of models precisely to accomodate the broad range of distributed applications possible. In contrast W* constrains the broad set of applications that can be modeled by fairly restricted semantics; similarly I* is geared towards sufficiently specificity so as to provide meaningful performance numbers.

Structure: The paper is structured as follows: the next section outlines our approach and introduces the A* approach to modeling applications, as well as explicitly discussing an A* model; we also briefly discusss models for workload (W*) and Infrastructure (I*) in detail. Details of the models, with examples and illustrative (JSON) code snippets are provided in Section 3, which also addresses how the self-contained, semantically complete models can be integrated to model and predict execution properties of a distributed applications on a HPDC infrastructure. The end-to-end integration is critical for it provides the initial ability to estimate (restricted measures) of performance of a given application on a given infrastructure. In Section 4 we describe the objective, design and implementation of our experiments, as well presenting the results and analysis. Section 5 discusses related and relevant work. We conclude with a discussion of the main contribution of our work and some future directions.

II. General Approach

The aim of our work is to model the execution of distributed applications, to reason about their utilization of distributed infrastructures. 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 breadth and variation of distributed applications; a workload layer (W*) that describes the transformation of application workloads; and an infrastructure modeling layer (I*) which captures the performance determinants of an HPC infrastructure and its provided capabilities.

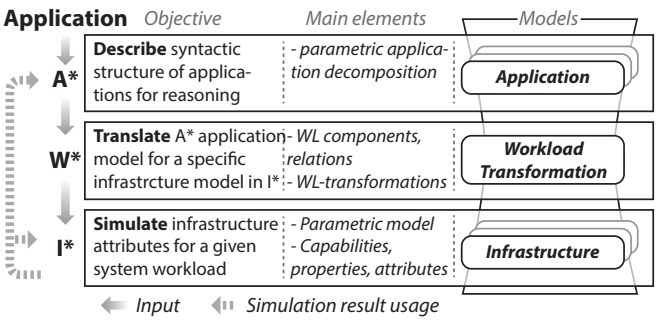

Fig. 1. The integrated modeling spans three semantically well-defined modeling domains.

A* Application Models – 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 [**?**]. The different models in this layer typically express the units of decomposition, their interaction and other core application characteristics. Independent of the specific model, all A* models represent an application instance as an *Application Workload (AW)*, which describes the application components and their relations.

W* Workload Models – The exact semantic and syntactic structure of those 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 syntactic restructuring of the workload and the derivation of 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 – it represents an explicit execution plan.

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. The I* layer again 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 [**?**] is used.

III. Technical Details

A. Modeling Approach in the A* Layer

It is important to distinguish the A* approach from specific application models that are 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. The A* approach thus 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.

As the uppermost model of the stack, an A* model provides means to represent and to reason about actual applications – most importantly about its decomposition. It focuses *not* on the modeled application's semantics, but rather on its syntactic structure – while the semantic decomposition of applications is an important determinant for the structure and specifics of application components, it is explicitly not addressed, yet. Individual models in A* cover different application classes – but *all* models in A* represent those applications as sets of interconnected components – an application workload, see W* layer description below.

The granularity of the modeled application components are given by potential distribution options. Relations between components cover information and state exchanges, and are classified as causal, temporal or spatial. A* models do not describe *how* those relations are implemented, but they aim to support the reasoning about the applications' execution properties for different properties of those implementations. A* models thus apply to classes of applications which have comparable syntactic structure, even if the actual semantic workload and implementation details can differ significantly.

B. Workload Modeling and Transformation – the W* Layer

The Workload Modeling layer W* again represents a *set* of workload models which mediate between application (A*) and Infrastructure level (I*). Any model in W* is comprised of a well-defined application workload (AW), a well-defined system workload (SW), and a set of transformation relationships between these workloads. While A* models capture the variability in the application structure, that variability is quenched by W* – W* models describe the transformation of application requirements, towards explicit infrastructure capability consumptions. The application's distribution structure is fixed and frozen in that transition to SW, and thus becomes suitable to map to explicit infrastructure resources.

The transformations from the initial state (AW) to the final state (SW) prescribed by W* can operate on any workload attributes – we focus on two transformantions: (i) from application component structure (AW) to execution structure (SW); and (ii) from application resource requirements (AW) to expected capability and resource consumptions (SW).

Transformations on the execution structure (i) need to maintain semantic integrity of the application – but can otherwise arbitrarily re-arrange the components of an AW. The transformation can use the parameters for the application decomposition, as defined by the A* model. It will generally be useful and necessary to interpret properties of the target resources 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). Ultimatele, applications consume capabilities offered by resources. The expected type and amount of capabilities needed to execute an application component are its *capability requirements* – such as compute (number of FLOPs[1]), memory (number of bytes allocated), and Disk I/O (number of bytes read/written).

It is generally difficult to predict exact resource consumptions [**?**]. There are three possible approaches to reasonable predictions: (a) application analysis, (b) application profiling, and (c) human input. Analysing application algorithms and their dependencies on input data and execution environment (a) is non-trivial, in particular for distributed applications – and is considered out-of-scope for our work (see *Halting Problem*). Application profiling (b) can result in truthful data on resource consumptions, but it is generally costly to do for a many input data and execution environments. The use of user-supplied data (c) is often cheap, but inexact – it is basically a case of (b) where the user estimates values based on previous experience.

The work presented in this paper focuses on motivation and validation of the general modeling approach – for that purpose we employ 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 parametric I* model [**?**], (HPDC) infrastructure components represent hardware entities and are described by three terms: capability, property, and attribute.

[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.

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 completely 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 [**?**].

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 [**?**]. 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 [**?**], 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 [**?**]; (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 [**?**], 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.

Simulating the execution of SW on an HPDC infrastructure requires scheduling, i.e. mapping tasks on services or (distributed) resources that can execute them. Task mapping faces manifold challenges, pursues different goals [**?**], [**?**], and is implemented in several ways, like operations research, constraint programming [**?**], or resource allocation constraints [**?**]. This complexity and variety cannot be covered by only one modeling approach. Hence, we abstract mapping algorithm commonalities and provide an interface that can be implemented by arbitrary existing (specialized) approaches.

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 [**?**], 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 ele-

ments on infrastructure components using the exposed capabilities is possible and that an abstracting, simplified mapping algorithm is capable of realizing the most important mapping algorithm 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.

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*.

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.

a) 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.

```
"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.

We will shortly discuss Synapse in subsection IV-A2, and will also support the claim that the applied transformations are truthfully preserving the original application's execution properties.

IV. Experiments

This section presents a number of experiments designed to validate our approach, and, to a lesser extent, assess its predictive accuracy – to reiterate: this paper does not claim that the presented work is fully applicable to a large range of applications, workload transformations and infrastructures – but it rather intents to validate the general approach, by demonstrating its applicability for one simple (albeit reasonably realistic) application use case. 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 that application;
2) **(ii)** prove that Synapse (see section III-D0a) is a suitable and precise tool to mimic and emulate application resource consumptions;
3) **(iii)** prove that the used infrastructure model 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.

```
"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).

This section will describe the application, tools and infrastructures used in the experiments, and then continues to discuss the individual items listed above.

A. Application, Tools and Infrastructure

1) Exemplar Application Workload: The used exemplar application is the computation of the mandelbrot set (MB) via a uniform bag of tasks, which each task computing a certain subset of the MB – the tasks are for the experiments tuned to each consume the exact same amount of resources. The application can be configured via some parameters (boundaries of complex plane, iteration depth etc), which consequently alter the quantity of resources consumed by each task. Communication and recombination of the partial results is ignored.

The application consumes three types of resources: compute (a certain number of FLOPs), memory (allocating and writing a certain amount of memory – in Bytes) and storage (writing a certain amount of data to disk – in Bytes). These three types represent in general very fundamental application activities – with network interactions the most obviously missing one. We assume no finer grained structure of the application activities, and assume that all three types can occur concurrently, thus competing for resources. We further assume that multiple application instances can 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.

All experiments focus on the *time to completion* (TTC) as central metric to characterize the application's execution, because (a) it is easy to measure and to compare, (b) it is relevant to application execution in general, and (c) it is directly correlated to modeled infrastructure properties, which makes it easier to evaluate and discuss the results in a meaningful way.

2) Synapse: We implemented Synapse, which 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 implemented 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.

Synapse also interprets SWs as described in section III-D, and emulates the application instance thus described. Synapse in the wider sense *is* an application implementation, but one which is tunable to mimic (emulate) any other applications' resource consumption while being semantically useless. Synapse is used in our experiments to compare the model predictions to actual execution of the SWs, as the Synapse instrumentation and its controlled execution allows us to gain more insight into the details of the SW execution – for example, Synapse enables us to to cleanly separate the TTC contributions from compute, storage and memory interactions.

The Synapse workload 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.

3) 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 makes it difficult to achieve full control over a particular compute node, unlike on common desktop system. Table I provides key specifications of the used infrastructure and Figure 4 depicts their component graphs.

India and *Sierra* are cluster systems in the FutureGrid[2]. 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 [**?**]. India's disks are connected via SATA 2.0. As Sierra is slightly older, it still uses a front-side bus (FSB). Compared to the India system, Sierra has a separate Northbridge and a Southbridge, i.e. the *Memory Controller*

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

Hub (MCH) and the *I/O Hub* (IOH), respectively.

Boskop is a customary desktop system, using the fastest SATA revision of all three infrastructures, i.e. SATA 3.0. Since Boskop's processor is an AMD, also the North- and Southbridge are AMD chips, in particular 890GX and SB850.

Also indicated in Figure 4, infrastructure property values were gathered from (manufacturer) specifications (*S*), by measurement (*M*), or, if not available otherwise, by assessment (*A*). Measurements were done by benchmarks or via system inspection, and focus on throughput of the system components (in *FLOPs/second* or *GByte/second*). Since bus throughputs are often provided in *Gigatransfers/second* (GT/s), we applied the following formula to obtain GByte/s: $cf/(8 * 2^{20})$, with c being the channel width in bits, and f the transfer frequency in Hz. FLOPs/second were estimated via the range of values given in GeekBench[3]. In some cases, measured values were preferred over manufacturer specifications – for example, a specific configuration setting on Boskop (possibly caused by a defect chip) caused the memory to be significantly slower than specified.

4) Scheduling: Implementing the provided interface of the applied I* model (cf. Section III-C), 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.

(i) System Workload, and (ii) Synapse Emulation

The first set of experiments establish (i) – support the assumption that an application's execution properties can well bound by deriving a SW – and (ii) – that an application instance can be truthfully represented via Synapse.

The SW is derived from the AW via W* transformations. As discussed in III-B, this work currently focuses on structural transformations, and on the derivation of resource consumption estimates for our modeling purposes. The only possible structural transformation on the AW given in figure 2 is to fix the number of worker tasks in the bag-of-tasks – in our experiments, that number is set manually for the purpose for creating a variable system load, to show that predictions and experiments are consistent for a range of conditions. The second transformation, the addition of resource consumption estimates, is aided by an automated profiling process: the application workload is used to extract sufficient information to run a single task of the bag-of-tasks, and that task is run under a set of profiling tools – the observed resource consumptions (number of CPU cycles etc) are considered representative for all tasks (tasks are uniform), and the system workload is created with those additional information.

To confirm that the profiling produces correct values, and thus the correctness of the SW, the resulting SW is executed by our MB implementation, and the same SW is also executed via Synapse, using the resource consumption estimates for the Synapse configuration, so that Synapse is tasked to consume the exact same amount of resources. Synapse itself is then also profiled.

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

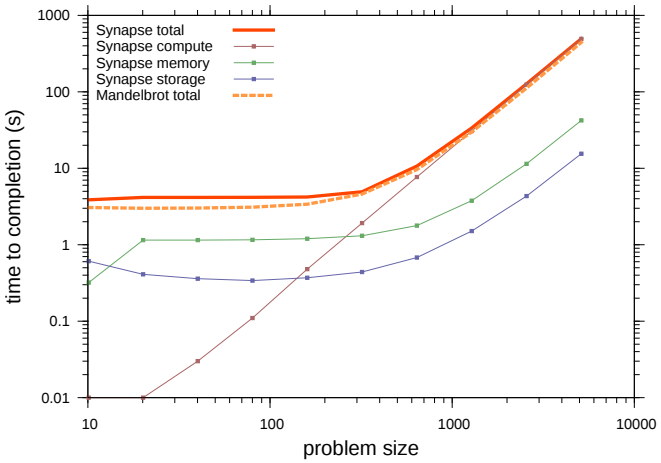

Fig. 5. Verifying Synapse: resource consumption metrics from a profiled application are executed as a SW by Synapse – resulting in the shown TTC relations: Synapse is able to truthfully reproduce the application's execution behavior over a wide range of parameters. The thin lines show the contributions of the individual workload types – compute, memory and storage – for the overall synapse TTC. This experiment was run on Boskop.

That procedure results in the following set of information:

1) TTC for MB application;
2) TTC for Synapse under the same configured load;
3) resource consumption profile for MB application;
4) resource consumption profile for Synapse run.

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 I).

That experiment is performed repeatedly while varying the dominant application parameter, the size of the complex plane for the MB algorithm – the resulting TTC values are shown in figure 5.

While the measured TTCs for small problem sizes, which result in application runtimes smaller than 10 seconds, show some deviations between the MB application runtime and the Synapse runtime, we see very consistent and compatible values for larger TTCs, confirming that the Synapse emulation of the application workload can truthfully reproduce the original application's runtime properties, for the scope targeted in this paper.

The deviations on smaller problem sizes were mainly caused by the granularity of the profiling tools – their measurement always added a small constant overhead, which is more significant for smaller overall resource consumptions.

B. (iii) Validating Model Predictions

Three weak scaling experiments (WS) for individual compute (WS.C), memory (WS.M) and storage loads (WS.S) have been performed on each infrastructure, with a varying number (1 to 64) of concurrent tasks, to show that the model predictions are compatible with the measurements along those axis.

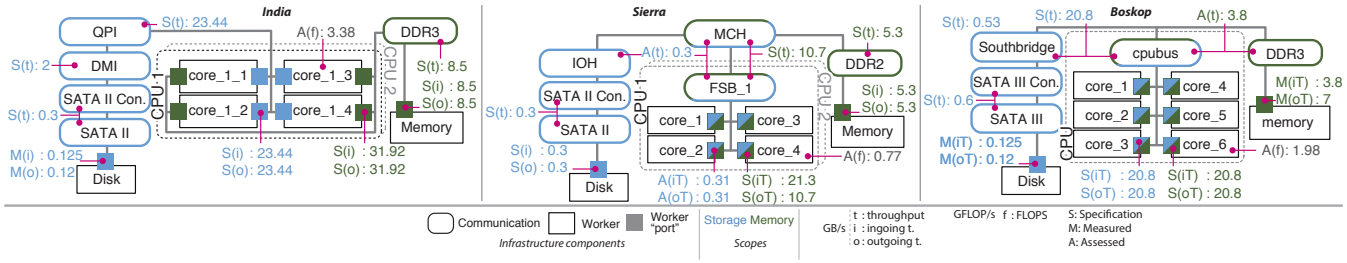

Fig. 4. Topology and property values of the infrastructures employed for experiments

Three additional experimental setup used a mixture of compute, memory and storage resource consumptions, where the initial values were chosen to be approximately balanced in their respective TTC on all tree infrastructures – that load was then varied in two dimensions: the number of concurrent tasks was varied (1 to 16); the load was consecutively biased toward higher compute (C.1-16), memory (M.1-16) and storage load (M.1-16), respectively.

The ranges of SW parameters used for the experiments are summarized in table II.

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 II

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.

All graphs below 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

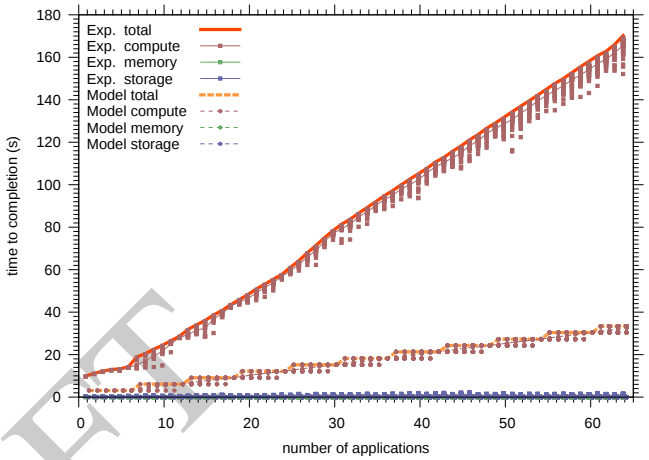

Fig. 6. **Weak Scaling of compute load on Boskop:** while both model and experiment agree on the overall trends in good detail, we observe an offset by a constant factor – for discussion, see text.

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.

Figures 7 shows 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), on Sierra. The graphs conveys the expected results: constant performance as long as application instances can be scheduled on an empty core, then a linear increase of TTC once all cores are used, etc (Sierra has 8 cores). The individual experiments (i.e. each data point) are running on a random node on Sierra, as

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 I

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

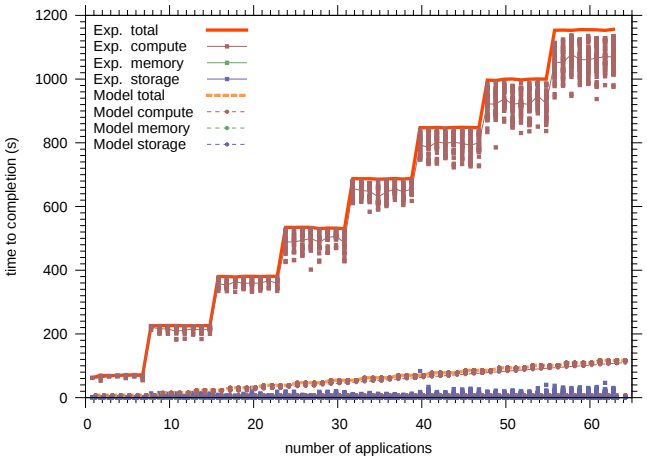

Fig. 7. **Weak Scaling of compute load on Sierra:** very similar to the Boskop results, we observe the same overall trends, and again the constant offset factor. Note the excellent reproduction of stepping caused by the 8-core CPU configuration.

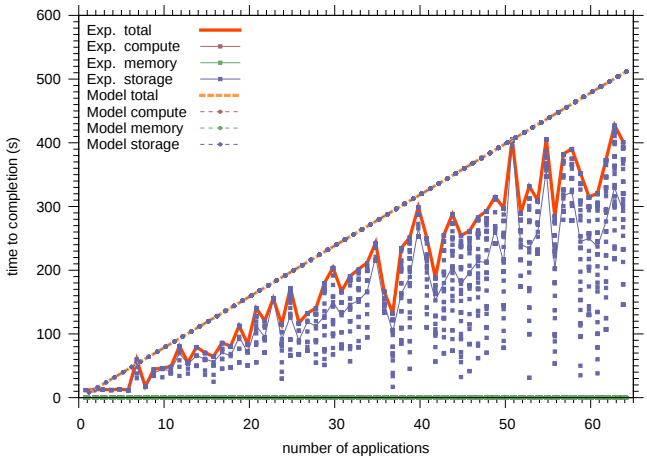

Fig. 8. **Weak Scaling of storage load on Boskop:** the results are in very good agreement with the model predictions – the resource contention due to the number of tasks leads to the expected increase in TTC. For more details, see text.

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.

While the qualitative model predictions are well reproduced, the measurements are quantitatively off by about a factor of 10. This is owed to the discrepancy between assumed CPU performance (FLOPs/second), and the *efficiency* with which the application code can actually utilize the CPU: we observe that the floating point pipeline remains idle for about 1/3rd of the cycles, and neither the application nor Synapse use SSE or AVX instructions – which result in performance boost factor between 4 and 10 (see [**?**], table 3). The future work discussion in section VI will discuss this issue.

The graphs show further 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 continued relocation of the application threads across the different cores), and to the background OS load (which is ignore by the modeled data).

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 represents a 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 shows a similar experiment, now focusing on storage scaling (here on Boskop). The predicted and observed TTCs are in generally good agreement – but it is obvious that the Synapse runs sometimes see much shorter times than predicted – we attribute that to the aggressive disk I/O caching of Linux. The overall TTC remains realistic though, as the total amount of data does not fit into the cache, so only some of

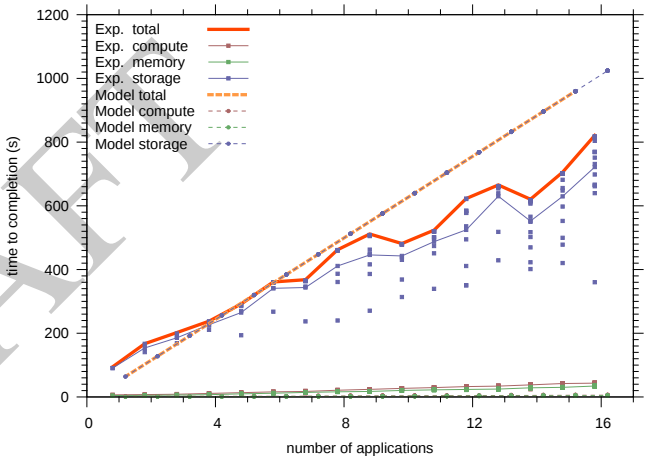

Fig. 9. Storage-heavy applications Boskop

the tasks from the uniform bag-of-tasks can benefit from the caches.

Figure 9 exemplarily shows the result of a mixed load 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.

(iv) Relevance for Distributed Applications

The ultimate goal for the presented work is to be able to reason about application distribution, by predicting application performance via the described model stack. While the paper at this stage does not include explicit experiments with distributed application instances, there are still a number of observations which are relevant for that topic, and which motivate future extensions.

The clusters india and sierra showed a noisier distribution of application runtimes than expected, it in all experiments which involve even smal amounts of disk I/O – the variation in disk performance over the cluster made the overall TTC prediction very difficult, due to very large noise and frequent failures. 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. That noise may not that obvious for other applications with less tightly controled resources requirements, but has the potential to influence distribution decisions – for example for tasks which need to run in lockstep.

Comparable offset factors between model predictions and measurements have been observed on all three infrastructures – which seems to indicate that both model and execution results are indeed comparable between infrastructures.

V. Related and Relevant Work

There is a set of approaches addressing the prediction of application behavior, e.g., performance prediction [**?**]. All these approaches have in common that they are not pursuing an integrated approach, i.e. they do not cover the entire stack comprising the application, infrastructure, and task mapping. Covering the entire stack is mandatory to enable reasoning about application execution, especially distribution and infrastructure configuration.

To the best of our knowledge, there is no approach covering the entire stack and hence, we are considering elements of our approach separately. In general, reasoning and prediction can be done by (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 run-time flaws, it is rather unsuitable for us, since we are aiming at integrating modeling of the entire stack (cf. Section II).

Empirical data can be retrieved by trace analysis or benchmark execution. Even if there are low-intrusive *trace analysis* approaches, like *Darshan* [**?**], and several tools for *benchmark execution*, both ways necessitate the physical execution of software at least once.

Our approach applies *model-based simulation*, which renders physical executions unnecessary and enables "what-if" analysis as illustrated by the conducted experiments and their parameters (cf. Table II). Several other approaches in the realm of model-based simulation are mature and time-tested, but they neither integrate nor interface with each other, which is required to cover the entire stack from the application down to the infrastructure (cf. Seciton II). Additionally, the majority of approaches is narrowed to a partition of our approach, e.g., *Dramsim2* [**?**] 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) [**?**] 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 do not provide interfaces for integration either, so adaption would be very costly and time-consuming.

In order to reduce modeling efforts, we investigated existing models and analyzed extension potentials to describe the layer elements, respectively (cf. Figure 1).

There are multiple models and tools to *describe applications* (A*), like *Triana* [**?**] or *Kepler* [**?**]. Mainly caused by their history, they focus on scientific domain-semantics and not primarily on supporting syntactic application decomposition, as required in Section III-A. Approaches related to transformation of AW to SW (W*), like *Pegasus* (*Planning for Execution in Grids*) [**?**], provide (sophisticated) mapping algorithms to place application tasks onto infrastructure components. Unfortunately, input is provided in extensive languages that are incompatible to our A* approach, which would cause a costly adaption and integration. To address their wide distribution, we are considering import functionality. For infrastructure description (I*), 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) [**?**], or the *Common Information Model* (CIM) [**?**], just to mention some of the most prominent ones. All analyzed approaches in common is their focus on technical details, like a disk's max block size, while omitting the mandatory infrastructure component attributes.

VI. Discussion

Although initial work, the comparision of our integrative models to actual experiments provides important validation of our approach of having well-defined models that support reasoning at every layer of the stack. Most obvious extension to this work will be to explicitly incoporate multiple resources that are concurrently utilized. In the future, we will extend our models to support well-defined functional *capabilities* based

upon which we will investigate how resources can be federated. This will provide the next step towards middleware that can support predictive services and reasoning about resource utilization. Last but not least, we think that this work has the potential to fundamentally alter how distributed resources are utilized, from a current best-effort ("execute wherever possible") but unplanned approach, to a more planned and reasoned approach to workload execution.

Acknowledgment – We would like to acknowledge Mark Santcroos' and Matteo Turilli's contribution to discussions on application and infrastructure modeling, respectively. Both contributed to initial scoping and towards better understanding of workloads (W*), which is still work in progress. **Author Contributions** – The experiments were designed by AM and SJ, in consultation with and input from CS. The primary experiments were performed by AM with important integration and implementation support from CS. Data was analyzed by AM, CS and SJ. SJ determined the scope and structure of the paper and wrote the introduction; AM and CS contributed equally to the writing of the paper, with editorial input from SJ and DK. The objective of the paper was collectively determined by all authors.