RADICAL-Pilot – A Scalable Implementation of the Pilot Abstraction

Andre Merzky, Mark Santcroos, Matteo Turilli and Shantenu Jha

June 30, 2015

Contents

Abstract

• **TODO:** SJ

The pilot abstraction embodies flexible resource management with user level control. The flexibility emerges from a combination of greater temporal and granular control. Even though Pilot Jobs were initially introduced as a mechanism to reduce queue waiting times, in recent years the pilot abstraction has emerged as an important approach to provide flexible resource management for dynamically varying resources as well as workloads. Although there are many pilot job implementations available, there is conflation between the resource management and workload management. In earlier work, we formulated the P* Model of Pilot abstractions, which provides a common minimal description of pilot-jobs. RADICAL Pilot builds upon the conceptual advances proffered by the P* Model, as well as experience in supporting many distinct science projects on a wide range of infrastructure from distributed computing infrastructure to the large-scale supercomputers. This paper provides the underlying rationale, objectives, design and implementation of RADICAL Pilot. It discusses the architectural and design decisions, as well as provides a detailed characterization of the performance. We develop a quantitative model of RPs performance, and discuss its response as a function of workload size, duration and characteristics. In addition, we also discuss a RP simulator that helps understand as well as predict the performance in response to different conditions/parameters/degrees-of-freedoms. We conclude with a section that captures our experience in supporting scalable science on production systems as well as designing tools, libraries and upper layers of middleware that use the pilot abstraction.

1 Introduction

• **TODO:** SJ

Introduction to Pilot-Jobs. More than beating & gaming the queue.

Pilot abstraction: Decoupling the workload placement from resource management, and in the process provide flexible execution modes and resource utilization.

Most often implemented in the context of jobs as Pilot Jobs, hence the name of the abstraction. But not confined to jobs, but also data (where the resource utilization is storage) and network.

The utility and power of pilot abstraction is not confined to the above. Much more as we will discuss but at the core of matters, it is a simple concept that helps overcomes the rigidity of traditional HPC systems and the execution modes they support. Also a way to overcome HTC(?) DCI limitation of collective resource utilization. The first two points are informally validated by the number of point-solutions (e.g., specific functionality) and isolated (e.g., on a given infrastructure) attempts that exist to overcome this limitation.

There is a need for consolidation: a consolidation of point-solutions/capabilities (on a given system) and consolidation of pilot-abstraction offerings across infrastructure.

1.1 Reference material from Pilot review paper

Mostly section 5.1 and bits of section 2

1.2 Why yet another pilot-job implementation?

- The very original motivation was practical:
  nothing we could use to support task-level parallelism for MPI tasks (functional) and with adequate performance (scalable)
- Consolidation of point-solutions, disconnected capabilities and infrastructure specific solutions. Conceptual simplicity with ease of use.
- improve MPI support, which we consider central for a pilot system to be able to run a large diversity of workloads.
- Imperfect state of connection between theory and practice
- Last but not least: Support research and production. Hitherto not possible.
  Alternative view: perish after publication vs premature development
- ...

1.3 Terminology and Definitions

***andre: I assume that definitions/terms are copied from the pilot review paper?

- **Application:**
- **Workload:**
- **Task:**
- **Compute Unit:**
- **Pilot:**
- **Pilot Agent:**
- **Resource:**
- **DCI:**

1.4 Structure

We structure this paper as follows: first, we will put our presented work into the context of existing PJ implementations (section 2), which will also provide further motivation to our approach. Section 3 we will discuss RADICAL-Pilot in more detail: we'll present and discuss its architecture from two different perspectives, and provide details about the RP implementation and feature set. Section **??** will present a performance characterization of RP, accompanied by a number of experiments which target specific RP components as well as integrated used cases. Sections **??** and **??** will discuss 2 different approaches to understand RP's performance characteristics ***andre: still not sure if both should be separate sections... . We will conclude the paper with section 6: a discussion of the various challenges and experiences made while implementing and using RP.

2 An overview of existing PJ implementations

- **TODO:** MS
  - UNDERSTANDING THE LANDSCAPE of Pilot Jobs
    OK to directly from Pilot-Review paper

- Redux of why yet another pilot-job implementation?
- Maybe even a bit of P* updated & refined..

3 Design of RADICAL-Pilot

RADICAL-Pilot (RP) is an implementation of a pilot system which aims to support the efficient execution of large scale workloads on distributed HPC and HTC infrastructures. RP is the upper layer of the RADICAL stack, a software stack developed in the RADICAL group at Rutgers University (see figure 1). 'RADICAL' stands for 'Research in Advanced Distributed Cyberinfrastructure and Applications Laboratory', and that software stack is designed to support that research context. RP's implementation is facilitated by RADICAL-SAGA [?], an implementation of OGF's SAGA API, which provides the required resource abstractions to keep RP itself portable and agnostic to details of resource access in distributed cyberinfrastructures. RP is prominently used by RADICAL's 'Ensemble-MD Toolkit', which is a collection of pattern based application abstractions, but RP is also directly accessible by other applications, libraries and frameworks (see section 6).

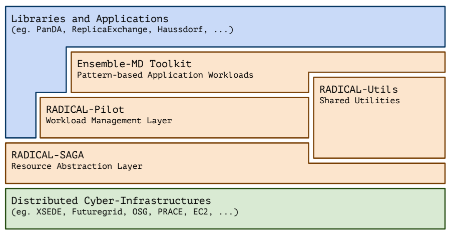
Libraries and Applications
(eg. PanDA, ReplicaExchange, Haussdorf, ...)
Ensemble-MD Toolkit
Pattern-based Application Workloads
RADICAL-Utils
Shared Utilities
RADICAL-Pilot
Workload Management Layer
RADICAL-SAGA
Resource Abstraction Layer
Distributed Cyber-Infrastructures
(eg. XSEDE, Futuregrid, OSG, PRACE, EC2, ...)

Figure 1: RADICAL-Cybertools is a software stack designed to support research on application scenarios in distributed cyberinfrastructures

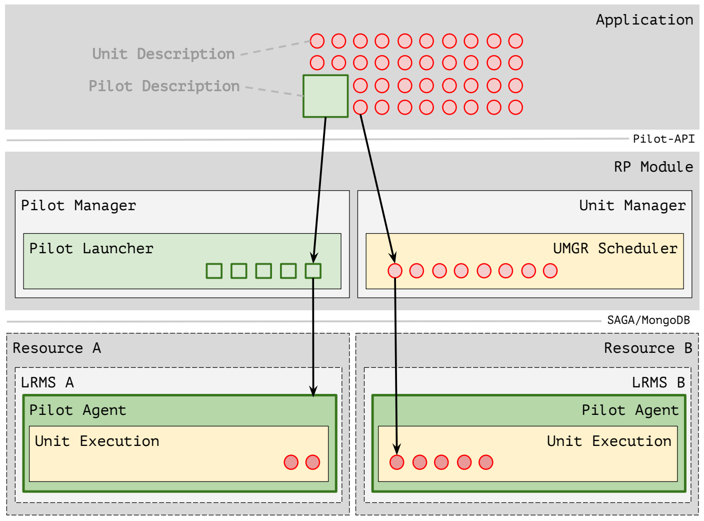

Figure 2: RADICAL-Pilot: Conceptual architecture

3.1 Design Objectives

RP is a pilot system [?], and on a high level perspective, it is conceptually very similar to other pilot-job approaches, such as those discussed in section **??**. The focus on distributed, heterogeneous infrastructures and on large scale, heterogeneous workloads though differentiates RP from other pilot systems:

- RP must be easily portable;
- RP must be able to utilize heterogeneous resources with different access types and policies;
- RP must be able to support different types of workloads (scalar, MPI, and OpenMP; small and large; short and long running; compute and data intensive, etc.);
- RP must be efficient and scalable (compute time on large resources is scarce);
- RP must support the programmatic specification of pilots and workloads;
- RP must be able to collectively use multiple pilots to execute a single workload – whether on the same physical resource or distributed distinct resources;
- RP must support experimentation into scheduling and research into distributed systems.

3.2 High-Level Design

RP is provided as a Python module: application, tools and frameworks can use the RP module API to (a) describe pilot jobs, and (b) to define workloads, which consist of compute units. The module exposes a pilot-manager (PMGR): it interprets pilot-job descriptions and uses RADICAL-SAGA to instantiates the respective pilot jobs on heterogeneous DCI resources; and a unit-manager (UMGR): interprets workload descriptions, instantiates the respective compute units, and schedules them onto the pilots for execution. By their nature, the pilots are entities which live on the target resources: the RP module communicates with active pilots via a central database (which simultaneously records global state for reconnect and recovery). Figure 2 illustrates that conceptual design of RADICAL-Pilot

The next subsections will discuss the RP architecture in more detail, and then describe the implementation of the individual components, including the discussion of several implementation choices. For each component, we will give a short overview over its performance characteristics – the overall RP performance is defined and discussed in much greater detail in section **??**.

However, before turning to architecture and implementation, we will describe and discuss the pilot and unit state models in RP, as they intrinsically relate to the RP component architecture.

***andre: make clearer how the design serves the objectives, above and in the sections below.

3.3 State Models

Both state models in RP are acyclic, ie. they have a finite set of initial states (specifically one: `'NEW'`), a finite set of final states, and state progression cannot form loops. The default state progression is in fact linear: under normal operation, an entity has exactly one state it is supposed to progress into at any point. A deviation of linearity is either an optimization (skipping a semantically empty intermediate state), or the entity reaches the `'FAILED'` or `'CANCELED'` final states.

Pilot State Model The pilot job state model (shown in figure **??**) is very similar to many state models used for jobs in distributed system – after all, the pilot job is exactly that, a job on a remote HPC resource. After construction of the pilot job instance (`NEW`), the pilot is submitted (`LAUNCHING`) to the target resource's LRMS (Local Resource Management System, which usually is a batch or queue management system). That LRMS will start the pilot job, thus moving it into the `ACTIVE` state where it will fetch and execute compute units. The pilot will finish either by explicit cancellation (`CANCELED`), by reaching the end of its assigned lifetime (`DONE`), or due to an error condition (`FAILED`). Cancellation and failure can happen on any state, successful completion (`DONE`) can only be reached from the `ACTIVE` state.

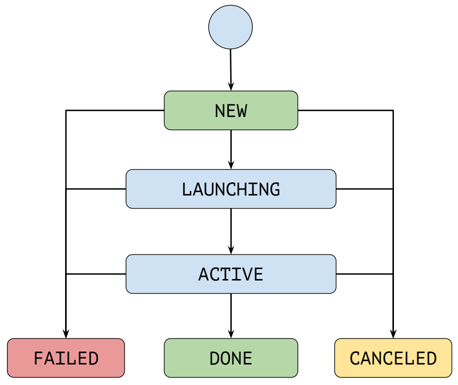

Figure 3: Pilot state model: the pilot is launched and eventually becomes active, ie. is able to execute compute units.

The pilot's `ACTIVE` state is the state where the pilot agent actively consumes resources on the target machine. The `ACTIVE` state has two internal phases: *bootstrapping* and *execution*. Those phases are not distinguished in the state model level, as we consider them to be an implementation detail. The *bootstrapping* phase prepares the execution environment for the pilot agent, and then start the second *execution* phase. It is and in that phase of the `ACTIVE` state that the pilot enacts progressions in the unit state models.

Unit State Model While the compute unit state model has a relatively large number of states, it is actually relatively simple designed: A unit gets created by the unit manager (`NEW`) and scheduled onto a pilot for execution (`UMGR_SCHEDULING`). Once scheduled, input data get staged to resource on which the assigned pilot runs (`STAGING_INPUT`). Once completed, the agent can place the unit onto a compute node it manages (`AGENT_SCHEDULING`) and execute it (`EXECUTING`). When the execution is completed, data are staged out (`STAGING_OUTPUT`), and the unit enters a final state (`DONE`). At any point, the unit can be canceled, and it then enters the `CANCELED` state. If any error is encountered, the unit will be moved into the `FAILED` state.

There exist two potential modifications to the state model in figure **??**. First, in some cases, RP can determine in

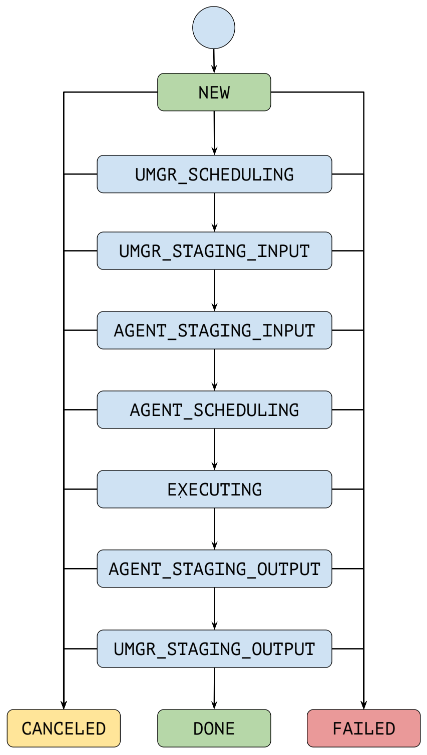

Figure 4: Unit state model: the unit manager schedules new units onto pilots, and performs part of the input data staging. The pilot agent manages data staging and sharing, the placement of units on the resource nodes, and their execution. After execution, the pilot performs some output data staging, then the unit manager picks the units up for final staging activi-

advance that a specific state will have no semantic meaning for a specific unit. For example, a unit which has no input data dependencies will not trigger any activity during its input staging states. RP could thus skip those states, and thus avoid the needless progression through the respective components. This optimization has been implemented for several states, but is not active for the experiments discussed in this paper: all units in those experiments will transition through all states.

Second, it is possible to implement failure recovery for compute units by moving them back to an earlier state. Specifically, a unit which fails when in control of a specific pilot can be rescheduled to a different pilot. The unit state moves back to the `UMGR_SCHEDULING`: this creates a cycle in the state mode which requires a termination condition. For example, such a condition could require to reschedule the unit at most n times, or to reschedule the unit at most once per pilot. This capability has been implemented (units can set a flag to signal they are eligible for re-scheduling on failure). Experiments in this paper are not using this feature, and we will continue to consider the state model as linear in the scope if this paper, if not explicitly mentioned otherwise.

`PENDING` States The pilot state model shown in figure **??** is somewhat simplified: before the pilot reaches the `LAUNCHING` and `ACTIVE` state, it will wait in a `LAUNCHING_PENDING` and `ACTIVE_PENDING` states, respectively. Those `PENDING` states signify that the pilot left the control of one software component, and waits for the enactment of the next state transition. For example, a pilot job submitted to a DCI resource's LRMS will typically be scheduled in a batch queue system, and will remain queued until the LRMS allocates resources to execute the pilot: only then will the pilot transition from `ACTIVE_PENDING` to `ACTIVE`. The full state model, including the `PENDING` states, is shown in figure **??**.

The same principle (each state is preceded by a `PENDING` state) also applies to the unit state model. We will discuss the implications of the `PENDING` states for the RP architecture and the RP implementation later on.

The complete unit state model, including all `PENDING` states, optimizing state transitions and rescheduling transition, can be found **FIXME:** either add as appendix or external reference.

3.4 RP Architecture

We will discuss the RP architecture on two different levels of granularity: (i) as macro-architecture, which defines the high level RP components in the context of distributed systems, and (ii) micro-architecture, which defines the set of functional entities and their relationships.

3.4.1 Macro Architecture

Pilot Systems are naturally distributed systems, as they necessarily feature one 'client' component which interacts with the application on the user's computer (laptop, department server, cluster headnode), and one or more 'pilot' components which execute the application workload on the target resources (eg. on cluster worker nodes). Some communication channel needs to be established between these components to communicate workload tasks and results. The high level architecture of RADICAL-Pilot can be characterized as having:

- the client component, 'RP Module', is rendered as a python module for pilot and compute unit management, with an application facing API;
- the 'Pilot-Agent Module' is a python program (agent) running on the resource, which executes and manages the compute units on that resource;
- communication between the client and pilot is established via a central database, into which the client registers compute units to be executed – the pilots eventually pull those requests from the database for execution. On its turn, the pilot use the database to communicate back the status and events of compute units to the client.

RP's macro-architecture (shown in figure 5) implements that naturally emergent set of components.

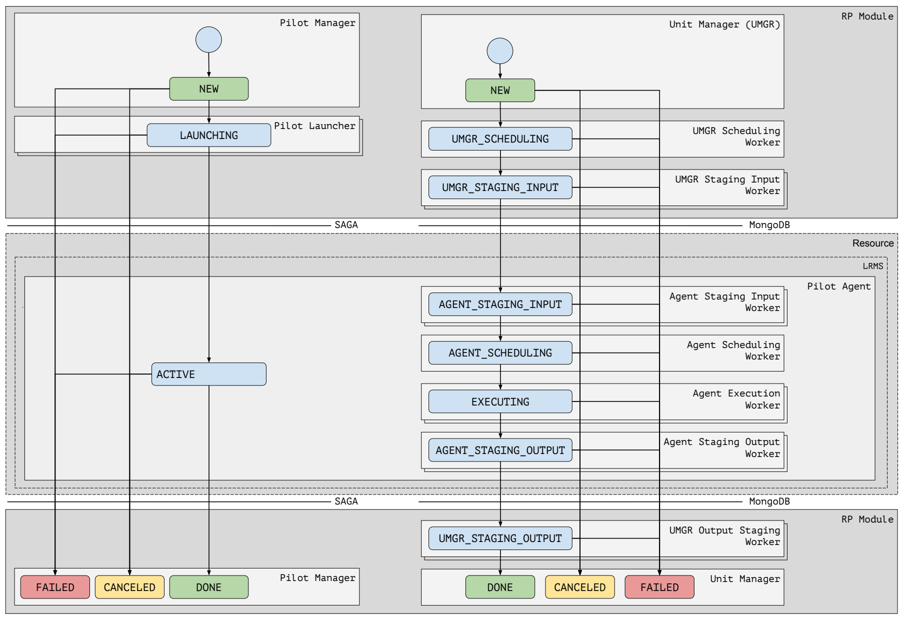

Figure 5: **Relation of Macro- and Micro-Architecture:** the 'RP Module' spawns the 'Pilot-Agent Module' on the target resource via SAGA, and then communicates with it via MongoDB. The worker components which implement pilot and unit state progressions (light grey) are contained in these modules, and communicate via queues. Communication channels represent valid state transitions.

Communication and Coordination on Macro Level The database-centric communication scheme has distinct advantages: it provides persistency for the global state of the pilot system and the workload execution (which for example allows the user to close the laptop after submission of work, and the system will continue to operate w/o interruption); and it provides connectivity in environments which are not favourable to direct network connections, ie. where NATs and firewalls on both ends (local user machine and remote target cluster / worker nodes) make it difficult to establish direct network communication. These advantageous have to be weighted against an potential performance tradeoff implied by the indirect communication between client and pilot components.

(Non-)Service Model The RP architecture does not include any *active* service component, i.e. no component which has a lifetime longer than the application itself (the database is longer living, but it functions as a *passive* data store, and does not perform any activities otherwise). This service-less architecture is very easy to deploy, as it operates fully in user space, thus reducing potential potential conflicts with the operational and security policies of the targeted resources. But this approach also implies that ongoing activities, such as workload scheduling, process monitoring and data staging, need to be performed by either the client component on the application end, or by the pilot agent component on the target resource. In particular the client component (living on the user's machine) is responsible for cross-site data staging and unit scheduling, and thus needs to be alive when those activities are required. That negates some of the advantages of the database-centric communication approach ***mark: I think that is unrelated to the database-centric approach, as it would have just as well applied to a direct communication channel , as it restricts the ability to disconnect the application after workload submission[1].

3.4.2 Micro Architecture

On a finer grained level, the RP micro-architecture is designed around the management of two different types of stateful entities: pilots and compute units. The respective state models have been discussed in section **??**. The RP micro-architecture defines

- the relation between the two state models;
- the individual components which implement the progressions in those state models;
- the relationships between those components; and
- the relation of these components to the RP macro-architecture.

Figure **??** shows how the RP micro-architecture embedded in the macro-architecture. For each state of the stateful objects, RP features one specific component which enacts that state. Communication channels connect these components: an existing communication channel represents a valid state transition. Where a communication channel crosses the boundaries of a macro-component, it is implemented either via SAGA or via MongoDB data exchange – otherwise communication channels are implemented as Python `Queue` objects.

That micro architecture maps to the macro architecture, in that the micro components are implemented and managed by either the client module or the pilot agent

[1] We consider to introduce some active service components in case that limitation turns out to be too restrictive for applications.

from the macro architecture. The `ACTIVE` state of the pilots ties the state models together: it is in that state that the micro-components owned by the pilot agent are operational and can enact unit state progressions.

Micro-components are (with some exceptions) designed to scale-out: several instances of a specific micro-component can exist in any given RP instance, for increased performance (throughput). For example, multiple instances of the `'AGENT_STAGING_INPUT'` component can coexist to perform data staging in parallel. We will discuss the effect (and the constraints) of this approach in section **??**.

The following subsections will describe the individual components of the macro- and micro-architecture in more detail, focusing on implementation choices which relate specifically to the RP design goals, see Section **??**: portability, support for heterogeneous resources, for heterogeneous workloads, efficiency, scalability, and programmatic use.

4 Implementation of RADICAL-Pilot

The RP macro-architecture discussed in section 3.4 implies a coarse grained structure of the RP implementation components, the RP micro architecture suggests a further subdivision into finer grained components which manage the pilot and unit state progressions. That describes indeed large parts of the overall code base (see figure **??**): the two macro-components are a Python module (RP Module), and a Python script (RP Agent). The RP module hosts two main components: the `radical.pilot.UnitManager`, and the `radical.pilot.PilotManager`, which host the respective micro-components for the module-side state progressions. The RP agent hosts the remaining micro-components enacting the unit state progressions on the target resource, specifically including the unit execution. We will first provide some details on the implementation of the macro components, before delving into the details and properties of the micro components.

4.1 The RADICAL-Pilot Module and its API

The RP Python module provides the RP API (in the `radical.pilot` namespace [**?**]), which is the single entry point for applications to interact with RP. The bulk of the API is provided by two classes: the `PilotManager` and `UnitManager`. Both classes provide the mechanisms to control the flow of pilots and units through their respective state models, ie. the application can (i) specify and manage pilot instances on different remote resources, (ii) submit and manage workloads (ie. sets of Compute Units – CUs) for execution on those pilots, and (iii) gather state and event information.

4.1.1 Pilot Manager

The `PilotManager` provides the pilot management API. The class accepts pilot instantiation requests (in the form of `PilotDescriptions`), which are instantiated as `NEW` pilots. The descriptions must contain a specification of the target resource – pilot placement decisions are out of scope for RP. Pilot instantiation requests are handled by the `PilotLauncher` class, which uses SAGA to interact with the Local Resource Management System (LRMS) of the target resource. In most cases, this will result in a batch queue submission of the RP agent script on the target resource.

The PilotManager additionally runs a separate thread which receives pilot state updates (via SAGA), and can issue asynchronous notifications about those state changes to the application. That thread also handles the transition of the pilots into any final state.

The code example shown in listing 1 shows an exemplary pilot instantiation, including a wait for the pilot to become ACTIVE, and an active cancellation.

```
import radical.pilot as rp

# create a description of a pilot to run
pdes = rp.ComputePilotDescription()
pdes.resource = 'xsede.stampede'
pdes.cores    = 1024
pdes.runtime  = 60         # minutes
pdes.project  = 'tg1234'   # allocation ID

# create a pilot manager instance, and ask it
# to run the described pilot
pmgr  = rp.PilotManager()
pilot = pmgr.submit_pilot(pdes)
print "pilot %s submitted (%s)" % (pilot.uid, pilot.state)

# wait for the pilot to become active,
# and immediately cancel it.
pilot.wait(rp.ACTIVE)
pilot.cancel()   # this is just a demo
print "pilot %s canceled (%s)" % (pilot.uid, pilot.state)
```

Listing 1: **FIXME:** caption

4.1.2 Pilot Manager: The Pilot Launcher Component

The pilot launcher component manages the LAUNCHING pilot state. It receives pilot descriptions via a Python queue which is fed by submit_pilot() calls from the PilotManager. Multiple instances of the pilot launcher component can exist, and share load by pulling from the same queue.

Before attempting to launch the pilot, the pilot launcher will perform a number of sanity checks (valid ranges of description attributes, presence of required attributes, etc), and will prepare the sandbox of the future pilot instance. The sandbox is a dedicated work directory which is created on the target resource via SAGA calls, and into which a number of prerequisites are staged, including the pilot agent script, the bootstrapper script (which is described in more detail in the pilot agent description), and module sources for the radical stack, if required.

Resource Config Files Resource specific config files are used to control the actions of the pilot launcher. Those config files include default values for some pilot description attributes and details on the system setup, as far as those are relevant to the pilot submission process. RP comes bundled with a resource configuration repository populated with a growing list of pre-configured DCIs. While RP tries to discover and auto-detect as much as possible, experience has taught us that on the level of detail that it matters to RP, there is a limit to what can be auto-detected and discovered. Therefore the inclusion of the resource configuration is a leading principle. This also means that every resource that is pre-configured has been tested by either someone of our development team, or by an external contributor. The obvious disadvantage of the model is that RP doesn't usually work out of the box, but we have put mechanisms that people can easily add custom configurations for new (or existing) DCIs.

The actual submission process relies completely on SAGA, which provides the required resource abstractions to submit the pilot agent as a `saga.job.Job` to the specific target resource (represented as `saga.job.Service`). Most configurations rely on ssh as access mechanism to the target resource, but any SAGA supported access mechanism is viable for pilot submission.

Performance Characterization **FIXME:** —— The relevant performance metrics for the Pilot Launcher component is (i) the throughput with which the component can enact pilot submission requests, and (ii) the scalability of the throughput with the number of pilots and backends. The plot in figure **??** shows the time needed by one pilot launcher instance to launch pilots on a number of target resources. Much of the required time is latency dominated, thus the secondary y-axis shows the ping latency to those target resources. The error bars document the variation of submission time over 10 pilot submissions.

The second plot, see figure **??**, shows the average time of submission over a range of pilot numbers, to a single resource. Multiple pilot launcher component instances can operate concurrently: the figure documents the respective scaling behaviour for 1, 2 and 4 instances.

In the waste majority of use cases, the number of pilots used for a single application instance will be small, ie. several orders of magnitudes smaller than the number of units submitted to those pilots. The goal for RP is thus to provide consistent performance for O(10) pilots, where each pilot submission needs O(10) seconds. The plots show that this goal has been reached and surpassed.

FIXME: ——

4.1.3 Pilot Manager: LRMS and the Pilot `ACTIVE` State

The pilot's `ACTIVE` state is thus a notable exception in the guiding principle of *'one RP component per state'*: that state is reached when the submitted SAGA job is in `saga.RUNNING` state on the target resource. As such, the state is actually managed by the target's LRMS system. Similarly, the pilot's `ACTIVE_PENDING` state is represented by the job waiting in the LRMS queue (when such a queue exists for that LRMS). The pilot manager uses SAGA's interface to the LRMS to keep (indirect) control over the pilot.

Performance Characterization The performance characteristics of this component is, by its very definition, dominated by the LRMS performance and thus out of scope for RP. The performance of the SAGA based interactions with the LRMS is discussed in [**?**]. The queue waiting time is, nevertheless, a significant contribution to the overall application TTC, and some approaches to handling queue waiting times in distributed, heterogeneous DCI will be discussed section **??**.

4.1.4 Unit Manager

The unit manager provides the unit management part of the RP API. Similar to the pilot manager, a `submit_units()` call consumes a set of unit descriptions, and thus starts their progression in the unit state model. The unit pass through a large number of states, which implies a larger semantic richness of the unit manager and its API. We will discuss details of that in the description of the individual micro components. Listing 2 shows, however, the fundamental parts of the interface: a unit manager instance gets created and configured with a set of pilots; units get created and get executed on the pilots.

```
import radical.pilot as rp

# create a UnitManager and give it
# two pilots to use for unit execution
umgr   = rp.UnitManager
umgr.add_pilots([pilot_1, pilot_2])

# create a set of unit descriptions...
descriptions = list()
for i in range(10):
  udes = rp.ComputeUnitDescription()
  udes.executable = 'mbr'
  udes.arguments  = ['-r']
  udes.cores      = 16
  descriptions.append(udes)

# ...and let the umgr execute them on its pilots
units = umgr.submit_units(descriptions)

# wait for the units to complete
for unit in units:
  unit.wait()
  print "unit %s finished (%s)" % (unit.uid, unit.state)
```

Listing 2: **FIXME:** caption

4.1.5 Unit Manager: The Unit Scheduler Component

The purpose of the unit scheduler is to bind submitted units to pilots for execution. RP uses a pluggable unit scheduler, so that, depending on workload type and application constraints, different ways to bind CUs to Pilots –under various levels of user control– can be used. A *Direct* scheduler allows the application itself to pick a specific pilot (and thus a specific resource) for execution; a *Round-Robin* scheduler does simple load balancing over available pilots; a *Backfilling* scheduler performs more fine grained (but also costly) load balancing based on actual workload of the pilots.

Performance Characteristics **FIXME:** —

The performance of the unit scheduler depends on a variety of parameters, such as the chosen scheduling algorithm, the number of available pilots, the sequence of unit submission, etc. Next to correctness (ie. the ability of the scheduler to define a scheduler which is correct and optimal under the chosen scheduling policy), the ultimately important metric is the throughput of the scheduler. The plot in figure **??** shows the throughput for a specific, but representative use case: the application defines 5 uniform pilots, and submits 10 batches of 1000 units, 1 batch per minute. Each pilot can concurrently execute 200 units. As we are only interested in the scheduler performance at this point, we will not actually execute the units, but assume they are immediately `DONE`, so that the pilots appear to be available for further units.

The unit scheduler is one of the few components which cannot use multiple instances for better scaling, due to the design of the scheduler plugins. The plots show, however, that the unit scheduler performance is consistent over the submitted batches, and shows a throughput in the range of **FIXME:** XX - YY units/second (depending on the selected algorithm).

FIXME: —–

4.1.6 Unit Manager: The Staging-Input Component

***andre: This paragraph could go into intro/scoping?
There exists a large body of research on data-compute co-scheduling, on data placement strategies, on optimizations for distributed data management, replica system etc – we will not discuss those research questions in this paper. Instead, please refer to [**?**, **?**, **?**] for a review of relevant techniques. In the context of this paper, we limit the discussion to the specific act of data staging, and describe the modes which are supported by RP, and characterize the performance of the implementation for some selected (but representative) cases.

Input staging is an essential capability for a pilot system: many applications define workloads where the compute units operate on specific data files, and since the pilot systems provide the means to run the units on a variety of target resources, it needs to be ensured that the units find the required data files when being executed. The *Unit Manager Staging Input* component [2] performs staging of data which are local to the client host (ie. local to the user). While the component processes one CU at a time, and ever only transfers one file at a time, several component instances can co-exist concurrently, thus resulting in parallel data transfers for the overall workload.

All transfer activities, including the creation of target directories etc, are handled via RADICAL-SAGA, and are thus platform and protocol independent. Globus-Online, scp and sftp are the dominating protocols used at the moment – but any other protocol is available if RADICAL-SAGA supports it.

Once the input staging is completed in the scope of the unit manager, the control over the unit is relinquished. The unit information are pushed to the database, where it can be picked up by the respective pilot for execution (see below).

Performance Characterization **FIXME:** —–

Transfer times for large files are usually bandwidth dominated, but in the scope of RP, which targets the execution of very large workloads with many units, the latency bound overhead becomes similarly important, specifically when many units require relatively small input files.

The plots in figure **??** show the throughput of the component, for numbers of files and file sizes. It also shows the effect of using concurrent instances of the staging component. The saturation of throughput for larger file sizes is caused by bandwidth saturation. Reaching that saturation early while keeping the latencies for small files as low as possible are the goals for the staging component performance.

FIXME: —–

[2]The capabilities we describe for this component do also apply for the *Unit Manager Staging Output* component, unless mentioned otherwise.

4.1.7 Unit Manager: The Staging-Output Component

After a compute unit passed through the unit manager's staging-input component, control is handed off to the pilot agent, and only once it was executed and the agent performed its part of the output staging, the control is handed back to the unit manager for final output staging activities and subsequent finalization. The output staging component is symmetric to the input staging, and has the same properties, and we will thus not discuss it in much detail.

4.2 The RADICAL-Pilot Agent

The pilot agent is naturally a central component RADICAL-Pilot. It is being launched on the target resource, where it, once it becomes active, actively manages a subset of that resource (a 'resource slice'). Several of the design objectives of RP depend directly on the implementation and operation of the pilot, and we will describe some relevant aspects in detail. After that, we will again look closer at the individual pilot components which are responsible for the unit state progression, focusing on the unit scheduling and execution.

4.2.1 Pilot Agent: Bootstrapping Process

The pilot agent is implemented as a Python process. To isolate all dependency setup from the agent implementation, we have implemented a bootstrapper shell script that creates all the conditions for a successful pilot agent startup. This includes the preparation of a suitable Python virtualenv, and the setup of network tunnels for communication with MongoDB and the application process.

Python Virtual Environments The pilot agent is written in Python. As RP is not (necessarily) software that is available on DCIs, the implementation of the bootstrapper is built on the premise that there are no requirements on pre-installed software other than a relatively modern Python installation[3]. This means that all required Python modules need to be installed by the bootstrapper. One of the building blocks we use is "virtualenv", a tool to create isolated Python environments. We use this to start with a relatively pristine environment. While installing additional libraries for Python on the surface may look simple and straightforward, our experience unfortunately tells us otherwise – we use resource specific config files to control the deployment process for specific target resources (new config files can be added by users for resources not packaged into the RP release, see section **??**).

For performance reasons, virtualenvs can be configured to be reused, to avoid the overhead of deployment for subsequent pilot submissions to the same resource. On-the-fly updates can also be triggered, to cater for newly released RP versions or for updates of dependencies. Additionally, to be able to support RP as a vehicle for research, we also put special care to make it easy to deploy development versions that lower the barrier to evaluation of experimental design.

Network access Once the agent is started, it needs to communicate with the central database to receive the workload to execute, and to report status and results. Because the bootstrapper runs in the context of a job on the DCI, there is a wide variety of connectivity constraints that RP has to handle. This ranges from no constraints

[3]Currently we require Python-2.6 or later.

at all, to only HTTP(s) traffic using proxies, to no outside network connectivity at all. We work around these constraints by dynamically setting up a network overlay to enable us to perform the downloads and communications that we require. Currently this overlay is build using SSH tunnels.

Performance Characteristics **FIXME:** —

Next to the queue waiting time, the pilot startup time is the dominating time between pilot `LAUNCHING` and `ACTIVE` state. While RP cannot influence the queue waiting time, it is is particular important to minimize the bootstrapping time, as the pilot's resource allocation is fully accounted for by the LRMS. The ability to re-use virtualenvs is crucial for minimizing that time span – it takes less than 10 seconds to prepare the pilot sandbox and to perform a sanity check of an existing virtualenv. Any required software update extents that period, depending on the size of the update and the network connectivity. Preparing a fresh virtualenv and installing the complete software stack can require O(1) minutes – that time is dominantly depending on the network connectivity.

• **TODO:** plots needed / useful?

FIXME: —

Agent Launch Once the python environment and network overlay are established, the bootstrapper will start the actual pilot agent process, which will spawn a number of components as threads or separate processes, and will set up message queues as means of communication between them. Once the outbound connection to the database has been made, the pilot is ready to work and can start pulling ComputeUnits for execution.

• **TODO:** add flow diagram?

***andre: include parts of the latex comment below

4.2.2 Pilot Agent: The Staging-Input Component

The staging-input component of the pilot agent complements the staging-input component of the unit manager (just as the staging-output components complement each other). The main difference really is that the agent staging happens in the execution environment of the pilot, which allows additional actions such as

- sharing data files between units via symbolic links;
- providing data files from the local pilot's staging area;
- using direct data transfers from some 3rd party data store to the target resource.

The application does not need to specify what input-staging component is activated for a specific staging directive ***andre: right? , but the choice is made dynamically, based on the data source and target locations, and on the requested action. For examples, staging via symlinks are always performed by the agent's staging component; data transfers to and from the application host are always performed by the unit manager's staging components.

The agent staging components are very similarly structured as those in the unit manager, and thus have very similar performance characteristics. We will not provide additional details at this point.

4.2.3 Pilot Agent: The Unit Scheduler Component

Once the LRMS and LMs are configured, and there are CUs available to be executed, the *Scheduler* will allocate resources for the CU and provide the optimal mapping of CUs to compute nodes and cores. The *Scheduler* is a component with multiple semi-resource specific implementations available. Currently two schedulers are available: the "Continuous" scheduler for flat interconnect layouts and the "Torus" scheduler for multi-dimensional interconnect layouts, e.g. for IBM BG/Q.

As just discussed, the *Scheduler* is responsible for the mapping CUs to computational resources, but it isn't responsible for the actual execution of the CU. This task lies with the *ExecWorker* component. The *ExecWorker* has an input queue where it receives CUs that are ready to be executed. The *ExecWorker* is also a component for which multiple implementations exist.These implementations are not (really) resource specific, and we also don't expect their number to grow too much, but their modularity is a great way to experiment with optimisation techniques for efficient task launching. The *ExecWorker* after execution of the CU also monitors the runtime and will report any error and is ultimately responsible for taking the CUs state to the next stage. As eloborated on earlier, the communication and coordination between the client and the pilot agent is through a (MongoDB) database. We have isolated all CU state update activity in the *UpdateWorker* component. ***mark: Asked Andre about some details as I actually dont understand the current implementation completely .

Performance Characteristics **FIXME:** ———

4.2.4 Pilot Agent: The Unit Execution Component

The execution of each compute unit requires a number of preparations. First, the units are examined for their resource requirements, and are placed (scheduled) into the set of cores managed by the pilot. A unique sandbox directory is created, then the pilot performs any remaining data staging operations, and in particular links shared data items from the global staging area into the unit sandboxes. Finally the pilot creates a shell script which represents the ComputeUnit workload, and executes it.

That shell script contains instruction for setup of the execution environment (loading system modules, setting environment variables, defining I/O redirection), and a specific command chain for the unit startup: single-core units are started as simple sub-processes, multi-core units are started via system specific versions or wrappers of 'mpirun' or 'mpiexec'.

RADICAL-Pilot supports a large range of those system specific startup mechanisms: different versions of ibrun, aprun and runjob**FIXME:** (?) which all wrap around mpirun or mpiexec (which are also called directly on some systems). Information about the execution environment, such as the number and topology of the managed nodes and cores, can be obtained under different resource management systems, such as SLURM, PBS Pro, TORQUE, SGE, DPLACE, etc.

- **TODO:** mention config files
- **TODO:** add details on OpenMP
- **TODO:** add flow diagram?

Performance Characterization ***andre: I would like to put Mark's experiments which focus *specifically* on the unit spawning into this section.

4.2.5 Pilot Agent: The Staging-Output Component

As mentioned earlier, the agent's staging-output component is complementary to the unit manager staging-output component: it performs output-staging actions in the execution context of the pilot.

The characteristics of this component is again very similar to those of all other RP staging components.

4.2.6 Pilot Agent: Additional Components

***andre: discuss heartbeat monitor, event handling for shutdown, async state updates

The *HeartBeatMonitor* component is an independent thread that monitors the database for commands destined for the pilot and also periodically verifies that all workers of this agent are still present. Lastly, it keeps track of the reaming wallclock time, and will actively shutdown the agent, even if the job might still be in the grade period of the LRMS.

FIXME: —— AM ——

4.3 Unique and distinctive features

4.3.1 MPI support

• **TODO:** iterate points from above about variety and flexibility of MPI support.
• **TODO:** point out that many other pilot systems miss proper MPI support

MPI is a standard for programming concurrent applications based on the message passing paradigm. However, we have experienced that there is little standardization on the launching of said applications. Most vendors and integrators, and even sites, have developed custom methods to launch MPI applications, in the addition to the variance that various batch queuing system add. Some of these customized methods are dictated by hardware (e.g. BG/Q), but not all of these customisations are equally justified. By providing an abstraction for launching MPI applications, and by implementing support for most common backend solutions, RP takes away the learning curve of submitting to new resources for end-users.

4.3.2 Data Management

Data Management, here narrowly defined as the staging of input and output files for the workload of the application is a first class citizen in the API and implementation of RP. File staging can be done on the level of the Pilot and on the Level of the Unit, whatever suits the workflow of the application better.

4.3.3 Range of Supported Systems and Workloads, Heterogeneity, Resilience

• **TODO:** SAGA as plumbing to get pilot onto heterogeneous systems Shell and Python as glue to seamlessly run on those systems modular and extensible agent to
• **TODO:** support variety of batch systems, schedulers, mpi versions, topologies, workloads, ...
• **TODO:** support for 'execution abstractions' ('application kernels'), so that ComputeUnits can be defined resource independently, which gets translated into resource

specific execution instructions after the scheduling to the target pilots (and thus target resources).
• **TODO:** different heterogeneous resources can be used simultaneously, for the same workload, w/o switching on application level
• **TODO:** different types of workloads are supported, from uniform independent bag of tasks to workloads which mix small and large ComputeUnits, which use scalar, MPI and OpenMP. Callbacks are used to support applications with dynamic workloads. Pilots can be added and removed on the fly. Failing pilots are gracefully handled (CUs are moved to a different pilot). Units can be restarted on failure or on pilots which time out / get canceled.
• **TODO:** more work on resilience on connectivity and data transfer problems is needed and planned.

4.3.4 Pluggable Schedulers for Multilevel Scheduling

• **TODO:** scheduling can be performed on application level (pipe CUs through DirectSubmission scheduler on RP level), or on RP level (unit schedulers, BF). Agent level scheduler does placement within scope of one pilot agent.
• **TODO:** unit schedulers pluggable, sensible default scheduler (BF), ensures distribution of work over pilots
• **TODO:** pilot level CU placement scheduler specific for target system layout, ensures that cores are kept busy
• **TODO:** data scheduling is approached, will follow similar approaches

4.3.5 Lightweight Deployment

• **TODO:** RP lives in user space, and does not require special setups on either user side nor on the target systems.
• **TODO:** It generally operates within the constraints and policies given by the target system.
• **TODO:** Shell and Python makes the agent very portable bootstrapping does not require any preparation or configuration on the target machines (apart from setting up password-less ssh login from headnode to compute nodes).
• **TODO:** support for new resources can be added via configuration files, either by the users or, on request, by the RP team. XSEDE, FutureGrid, some PRACE machines and several individual clusters are preconfigured and work out of the box.

4.3.6 Performance

• **TODO:** Jay!

4.3.7 Profiling Support

RADICAL-Pilot is continuously collecting performance information. Some of those information are available programmatically, during the runtime of an RADICAL-Pilot application. For example, time stamps for past state transitions are continuously appended to the respective object instances, and can be used to monitor progress of workload execution, and to react on certain events (such as Compute Units completing). To support the latter use case more efficiently, RADICAL-Pilot supports a callback mechanism which triggers application level callbacks on certain unit or pilot state transitions.

The complete state of RADICAL-Pilot sessions, including details as the set of used compute cores, timestamps on state transition and file transfers, is available as a set of PANDAS data frames, for automated analysis. Several utility methods and scripts are available to process those data for basic statistical information (TTC, average CU run time, average wait time, pilot utilization etc), and to provide graphical representations of the workload execution, see for example Figure **??**.

Other information, such as more fine-grained event monitoring on agent level, are available post-mortem: they are written to disk, and tools are available to fetch them, and to preprocess them, for example for conversion into PANDAS data frames.

***andre: show examples of logged / reported events?

4.3.8 Scalability

- **TODO:** out, across, above and beyond

5 RP Performance characterization

- **TODO:** AM, MS

We have been discussing the performance characteristics of the individual RP components in the previous sections – this section will focus on the integrated performance of RP for a number of different application classes. It will become clear that the overall performance is not simply the sum of its parts, specifically since RP supports different configurations, depending on resource layout and expected application workload. We will discuss the resulting trade-offs, and support that discussion with experiments.

To focus the discussion, we will first define a set of global performance metrics which we consider central to RP performance, and will discuss how those metrics relate to application performance. It will become apparent that unit execution throughput can be considered a very central metric in several aspects, and we will dedicate several experiments to that topic specifically. While we cannot possibly cover the complete range of supported applications, we will present and dissect a number of end-to-end application experiments, and discuss the relation between RP performance characteristics and application performance.

5.1 Performance Metrics

While total time-to-completion (TTC) seems the single performance metric which dominates user experience, there are other important metrics a pilot system must consider equally important. As a system which targets large scale compute resources, the utilization of those resources is very important, as that is a metric the resource operators usually try to optimize. In the same context, scalability is another important metric which emerges from the set of target use cases and target resources. And finally, while TTC can be rendered as a single number (for any given problem), it is determined by a number of internal performance characteristics, such as the minimal turnaround time for a single compute unit, the maximal concurrency of state transitions, and the like. This section will motivate what specific performance metrics RP attempts to optimize.

We separate two larger components of the performance discussion: the performance metrics which apply to the management of the pilot itself, and the metrics which apply to the management of the workload, ie. of the ComputeUnits.

5.1.1 Metrics of Pilot Management Performance

For the context of this discussion, a pilot agent is essentially a normal job on the target resource. As such, the performance of the job execution is determined by the target (batch or OS) system, once the job creation request is submitted. From an RP perspective, the remaining performance metrics are:

- t_{psub}: How quickly can a pilot creation request be submitted to a target system?
- t_{pstart}: How much time does it take for the pilot agent to become operational, to become ready to execute compute units, once the pilot job is started?
- t_{preq}: How quickly can the pilot react on management requests (such as shutdown)?
- How do the above metrics scale with the number of pilots, and the number of resources involved?

***andre: for the above metrics, we will present one plot each, varying over number of pilots and resources

The number of active pilots in the scope of a single application is limited, and not expected to be much larger than the number of resources involved – that makes the scalability issue lightweight on the pilot management layer (unlike on the workload management layer). The submission of the pilot creation request is performed via SAGA – so any performance boundaries on that metric arise from the SAGA layer. But since multiple pilot job submissions can be performed concurrently, both on the SAGA and the RP layer, and since the submission delay is usually tiny compared to queue waiting times, this is not a critical metric either.

The pilot startup performance is dominated by the creation of the Python virtualenv – see section **??** for details. That time can be greatly reduced by re-using virtualenvs, which is now the default for RP. Overall startup times are in the order of seconds**FIXME:** add measurement.

The time needed for a pilot agent to react on management requests is dominated by two contributions: the latency of the network connection to the central DB, and the frequency of how often commands are pulled from that DB. The latter in turn is determined by the load of the pilot agent: the main agent thread will need to forward all units received in the last batch to the follow-up components before being able to look for management requests. That time is dominating the responsiveness – but that schema also assures that no units remain in limbo when a management request is received.

5.1.2 Metrics of Workload Management Performance

Workload management is the core business of RADICAL-Pilot, and once the resource allocation is active (ie. the pilot agents are established and ready execute compute units), the workload management needs to ensure that those allocated resources are efficiently used for the workload execution – *resource utilization* is an important performance metric for RP. Resource utilization is defined as :

$$U = \frac{\sum_{i=0}^{\#units} ((t_i^{end} - t_i^{start}) * n_i^{cores})}{\sum_{j=0}^{\#pilots} ((t_j^{end} - t_j^{start}) * n_j^{cores})}$$

where $\#pilots/\#units$ are the total number of pilots and compute units, t_k^{start}/t_k^{end} are the pilot and unit execution's start and end times, respectively, and n_k^{cores} are the number of cores a pilot or unit allocate, respectively. U will be in a range from 0 to 1, where 0 signifies that resources are allocated but not used, and 1 signifies a complete usage of resources (no allocated cores are ever idle).

Several conditions need to be met to allow the pilot agent to achieve high resource utilization:

- when a pilot is active, it needs a sufficient number of units available for execution, ie. units for which execution preparation like assigning to the pilot, data staging etc, are completed;
- when units are available, the pilot needs to be able to spawn them quickly;
- finishing units need to be quickly replaced with new units – that implies that units which finish are detected quickly, and that cores which were used for their execution are quickly made available to other units.

Resource utilization is key to fast completion of the workload execution – where total time-to-completion, TTC, is defined as

$$T_c = \max_{i}^{\#units} t_i^{final} - \min_{i}^{\#units} t_i^{submit}$$

where t_i^{submit} is the time when a Compute Unit is submitted to RADICAL-Pilot, and t_i^{final} is the time when a Compute Unit reaches a final state. Those timestamps are different from t_i^{start} and t_i^{end} used earlier, which only refer to the execution stage of the Compute Units, when the units actively consume compute cores. For each unit, the total time spent in RADICAL-Pilot is the sum of the durations the Compute Unit spent in each state (state progression is linear and gapless in RADICAL-Pilot): ***andre: gapless, end-to-end, consecutive, uninterrupted, ... ?

$$t^{final} - t^{submit} = \sum_{j=0}^{\#states} (t_j^{end} - t_j^{start})$$

Significantly, the time a Compute Unit spends in RADICAL-Pilot includes the time it is waiting for a suitable pilot to become available (and thus potentially the pilot's queue time on the target batch system), and the time needed to stage data to and from the Compute Unit's working directory on the target resource.

High utilization is thus indeed a necessary condition for low TTC (which will be confirmed in the experiments in section **??**). However, the composition of the set of pilots, their size and distribution, strongly influences the performance metrics listed in section **??**, and thus also influences TTC. We have shown in **??** that those influences can be both more significant and more uncertain factors than pilot utilization alone.

***andre: For a single pilot, decisions and actions are local to the pilot in most cases (apart from staging). For multiple pilots, workload managements needs to plan and enact a distributed system. Explain the implications, such as information available for scheduling, delayed data staging, etc.

As the performance of task execution is so significant for pilot utilization U, which in turn is significant for TTC (T_c), we will specifically discuss execution performance, in more detail and wider context than in the presentation of the unit execution component in section **??**.

Figure 6: Details of ORTE Launch Method implementation.

5.2 Performance of Task Execution

Ultimately the execution of workloads boil down to the executions of the tasks it constitutes of. The execution (or launching) of tasks is the sole responsibility of the Pilot Agent. As discussed in the Agent Architecture section **??**, the Scheduler, ExecWorker(s), LRMS and LaunchMethods work together to execute tasks on the resources. ***mark: reintroduce agent architecture figure In this section we will investigate the isolated performance of task execution confined to the Agent only.

The following set of experiments are executed on Blue Waters (NCSA) ***mark: and Stampede (TACC)? . The experiments on Blue Waters, a Cray XXX, are performed using the Open Run-Time Environment (OpenRTE) as Launch Method (see fig6). OpenRTE [**?**] is a principle component of the Open MPI project.

5.2.1 Experiment 1: Task Duration

In these sets of experiments, we investigate the (relative) impact of the task duration (the duration of the payload of the task) on the TTC of a workload. The hypothesis is that task startup relies on the use of a shared resource and that therefore the dispersion of task launches has influence on the per-task overhead, and therefore on the total TTC. We execute a workload of 512 single core CUs on a pilot of 256 cores (8 nodes of 32 cores each) and we vary the duration of the individual task payload (/bin/sleep) for 0, 1, 10, 30, 60, 120, 300 and 600 seconds.

Figure 7 shows the TTC for the given workloads split up in actual payload execution and the time spend in the ExecWorker Queue, where the latter is a measure of the overhead to launch the tasks. Obviously the TTC grows with larger payload runtimes, but we also see that the cumulative overhead of the ExecWorker Queue shrinks.

Figure 8 zooms into the per-task ExecWorker Queue overhead and presents it as a mean value for the whole workload. The effect of increasing the dispersion seems to stabalize around 2 minutes, although this value is likely not independent of the total number of tasks to be executed.

Another perspective to look at this data is in Figure 9, where we show the core utilisation effiency for the given task durations. Although there is no normative correct value to be formulated for the effiency, it is safe to say that running single core tasks that run for less than 1 minute is not the best use of the resource. Still, this does depend on the science that would be enabled by doing this, we merely present it here as insight to make such decisions.

5.2.2 Experiment 2: Task Core Count

In these sets of experiments, we investigate the (relative) impact of the task core count (the number of cores used per task) on the TTC of a workload. The hypothesis is that because task startup relies on the use of a shared resource and that therefore the density of task launches has influence on the per-task overhead, and therefore on the total TTC.

We execute a workload of 60 second tasks on a pilot of 256 cores (8 nodes of 32 cores each) and we vary the number of cores for the individual task payload (/bin/sleep) for 1, 2, 4, 8, 16, 32, 64, 128, 256 cores. While the number of cores is the primary variable

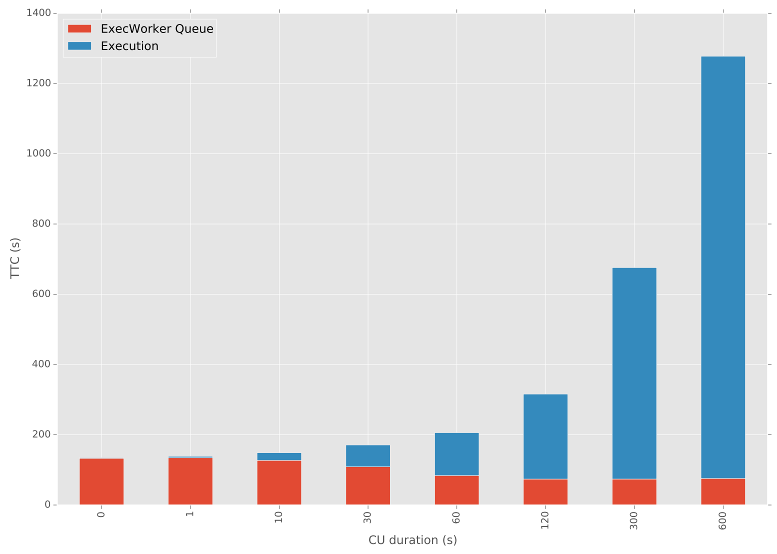

Figure 7: TTC (Y-axis) of a workload of 512 single core CUs. This figure shows the effect of varying the CU payload duration (X-axis) on the ExecWorker Queuing time (and naturally also on the CU execution time). Pilot size is 8 nodes of 32 cores each, meaning that 256 CUs can in principle run concurrently. The total time spent in the ExecWorker becomes shorter with longer running CUs, which indicates that the dispersion of CU launches alleviates the contention on the resources that are required to launch a CU. ***shantenu: the previous sentence needs attention. What is dispersion? I think a bit of mixing of cause and effect possibly? ***shantenu: We should speak to 512 CUs on 512 cores or 256 CUs on 256 Cores ***shantenu: all figures need to be done with Num. of CUs and Num. of cores at least 1 if not 2 orders of magnitude larger for publication grade! ***shantenu: What I want to see in the draft is: X-axis is number of core (or CUs), Y-axis TTC and multiple “lines” for different durations of the CU. When plotted to meet this requirement, this graph should have lines, not bars.

in this experiment, we also vary the number of CUs per workload, to keep the total consumption of cores (from a payload perspective) constant.

Figure 10 shows the mean per-task ExecWorker Queue overhead for executing the workloads with varying task core counts. While the total consumption of actual core time is constant, the overhead for smaller tasks is significant. Similar as with the task duration, it is impossible to give a normative answer to what is an acceptable use of resources.

Figure 11 presents the data from a core utilisation efficiency perspective to give a better view of the effect of the overhead on the utilisation of resources.

5.2.3 Experiment 3: Number of ExecWorkers

As discussed in the implementation/archicture of the Pilot Agent, many workers are implemented so that there can be similar concurrent instances working on elements in

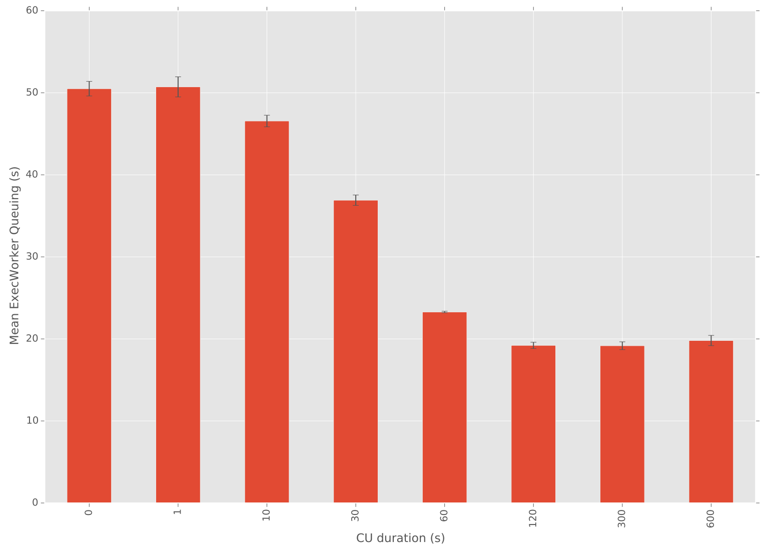

Figure 8: Mean time in ExecWorker Queue(Y-axis) with a workload of 512 single core CUs. This figure shows in more detail the effect of varying the CU payload duration (X-axis) on the ExecWorker Queuing time. Pilot size is 8 nodes of 32 cores each, meaning that 256 CUs can in principle run concurrently. The time spend in the ExecWorker Queue becomes shorter with longer running CUs, which indicates that the dispersion of CU launches alleviates the contention on the resources that are required to launch a CU. ***shantenu: I don't see any more detail than the previous figure. Why is the value different than the previous figure as the experimental configuration seems to be the same? also, last sentence is the same as previous figure and needs fixing

a queue. The ExecWorker is responsible for launching tasks and is also configurable to be multi-instance. There will still be a single Queue where the scheduler puts Tasks that are ready to be executed into, but then there will be multiple workers who work out of that queue to parallelize the actual startup. The hypothesis in these experiments it that because of the synchronous nature of task execution, launching multiple tasks concurrently can have an effect on the throughput. As hypothesis and validated in experiments 1 and 2, given that ultimately there are shared resources that are used for this, there will be limit to the benefit obtainable from paralellising the launching of tasks. The workload for these experiments is a set of 512 single core tasks that have a payload (/bin/sleep) of 0 seconds, where we vary the number of concurrent ExecWorkers (threads) from 1 to 8.

***mark: Need more repetitions once BW is back to draw conclusions.

Figure 12 shows the TTC of the workload while varying the number of workers. While there is clear improvement to the TTC, there are also rather large error margins for the higher number of workers. Additionally it needs to be realized that these workers run on a shared (with other users) resource where the additional workers put more strain on the resource than benefits for the TTC justify.

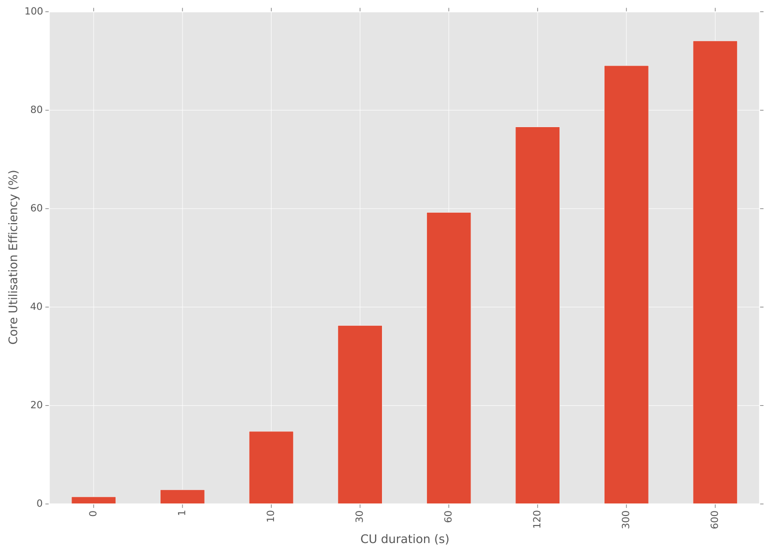

Figure 9: Core utilisation efficiency (Y-axis) of a workload of 512 single core CUs. This figure shows the result of time spent in the ExecWorker Queue for varying number (X-axis) on the core utilisation of a pilot. Pilot size is 8 nodes of 32 cores each, meaning that 256 CUs can in principle run concurrently. The percentage is calculated by the fraction of time spent in executing the actual payload as part of the total resources required to complete the full workload.

***mark: todo: if time permits try with processes instead of threads

Figure 13 shows the impact on a per task basis.

More intuitively, Figure 14 shows us the actual launch rate of tasks for the given worker configurations.

To get an idea of the progression over time, Figure 15 shows the ExecWorker Queueing time for individual tasks ordered chronologically on launch time. At the first instance, the task scheduler knows that all resources of the pilot are available and schedules 256 tasks into the Qeueu. The ExecWorker(s) process these tasks from the Queue as fast as they can. Given the ordered processing of these tasks from the queue, the ExecWorker Queue overhead for the first 256 tasks gradually grows, until it reaches steady state for the execution of the remaining 256 tasks.

5.3 Integrated Experiments

FIXME:

Having defined the relevant performance metrics metrics (U, T_c, scaling with units and resources/pilots/heterogeneity), we will show experiments for specific application cases which show how those metrics behave.

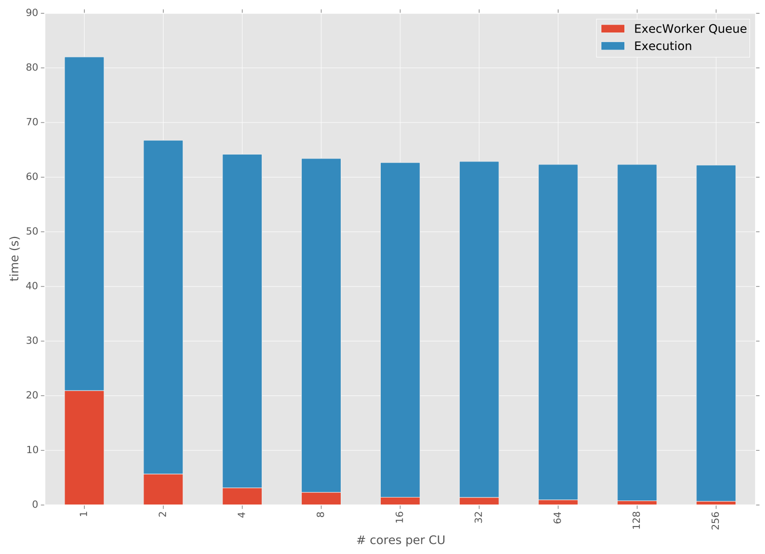

Figure 10: Mean time in ExecWorker Queue and Execution (Y-axis) of a workload with a varying number of CUs with a payload of 60 seconds and varying CU count. This figure shows the effect of varying the number of cores per CU (X-axis) on the ExecWorker Queuing time. The number of CUs executed is inversely proportional to the number of cores per CU, to keep the total demand on the cores equal. The Pilot size is 8 nodes of 32 cores each. The total time spend in the ExecWorker Queue becomes shorter with larger CUs, which indicates that the density of CU launches alleviates the contention on the resources that are required to launch a CU. ***mark: add errorbars ***mark: possibly remove task duration?

5.3.1 Application 1:

Gromacs BoT, for a small and a large science problem.

Plot 1: U, T_c with varying unit and pilot numbers.

Plot 2: global state progression of pilot and units, which will identify the bottlenecks (and shows off the plotting for app analysis... ;)

5.3.2 Application 2:

One of the Ensemble-MD patters with data dependencies,

Plot 1: U, T_c depending on problem size (scale) and data size

Plot 2: influence of data staging strategies (CU based vs. pilot based; early binding/staging vs. late binding/staging)

5.3.3 Application 3:

AIMES, Matteo's experiments and plots.

***shantenu: Possibly don't reproduce data to avoid duplication...

FIXME: ——————————————

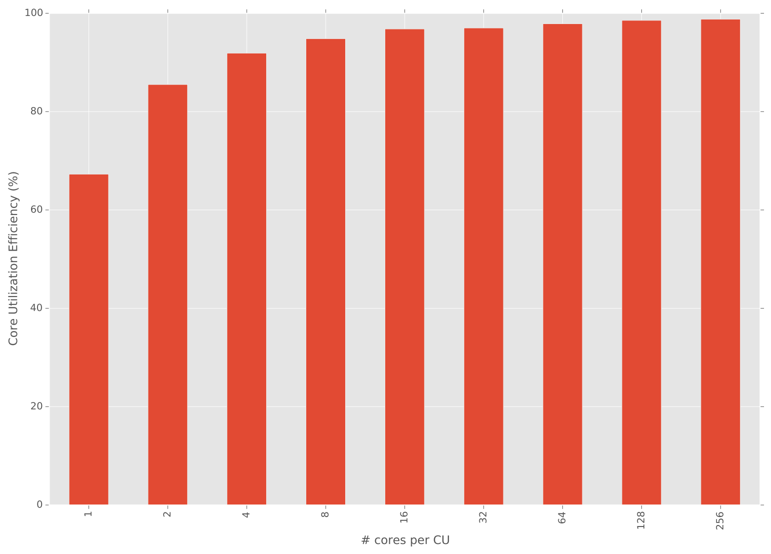

Figure 11: Core utilisation efficiency (Y-axis) of a workload with a varying number of CUs with a payload of 60 seconds and varying CU count. This figure shows the result of time spent in the ExecWorker Queue for varying number of cores per CU (X-axis) on the core utilisation of a pilot. Pilot size is 8 nodes of 32 cores each. The percentage is calculated by the fraction of time spent in executing the actual payload as part of the total resources required to complete the full workload.

6 Experience

6.1 Challenges

6.2 From distributed resources to high-end machine

- **TODO:** Providing an uniform execution model
- **TODO:** Challenges in supporting both HPC and DC/HTC

6.3 RP usage exemplars

- **TODO:** External Users (??)
- **TODO:** AIMES (MT)

RADICAL-Pilot has been adopted by the AIMES project [ref] to develop and characterize the concept of execution strategy for a workload/workflow. Defined as a set of temporally ordered decisions, an execution strategies characterizes and parametrizes the procedure by which the requirements of the given workload are matched to the properties of target resources. Typical decisions composing an execution strategy concern the number of pilots required to execute the workload, their size and duration, the pool of resources on which these pilots should be scheduled but also where, how,

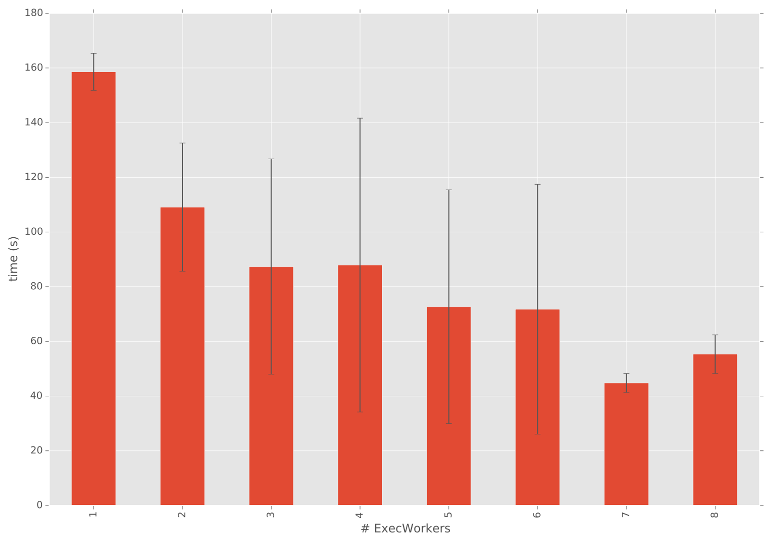

Figure 12: TTC (Y-axis) of a workload of 512 single core CUs with a varying number of ExecWorkers. This figure shows the effect of varying the number of ExecWorkers (X-axis) that enables the parallel launching of CUs on the TTC. Pilot size is 8 nodes of 32 cores each, meaning that 256 CUs can in principle run concurrently. ***mark: Should probably use the "split" version for this plot. ***shantenu: scale needs to be increased by 2 orders of magnitude

and when the task of the workload should be executed, and how their data should be handled.

RADICAL-Pilot has been used to isolate and measure the effect of every decision of an execution strategy on the overall time to completion of the given workload. One experiment for each decision was devised so to test its effects on the overall execution of workloads with different requirements. Workloads were devised with varies number of tasks, various distribution of their durations, different data requirements. Furthermore, single or multiple resources were targeted across multiple administrative domains. The execution of each experiment run was timed and statistical methods were applied so to understand the correlation between the specific decision of the adopted execution strategy and the overall time to completion of the workload.

The first experiment focused on . . . .

The second experiment focused on . . . .

- **TODO:** EnMD-TK (VB)

Author Contributions

AM, MS and SJ planned this paper, and it was written primarily by AM with contributions from MS and SJ. Experiments core to this paper were designed, executed and

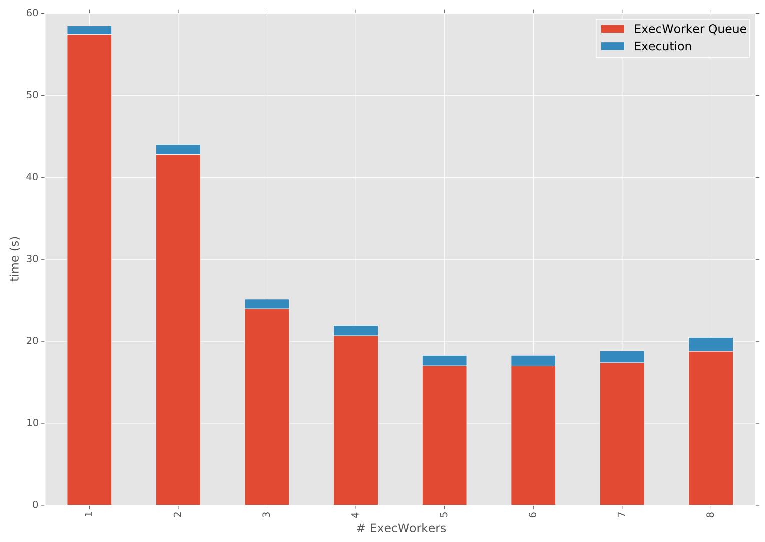

Figure 13: Mean time in ExecWorker Queue and Execution (Y-axis) of a workload with 512 single core CUs with a payload of 0 seconds and varying the number of ExecWorkers (X-axis). This figure shows the effect of varying the number of ExecWorkers on the ExecWorker Queuing time per CU. The Pilot size is 8 nodes of 32 cores each. ***mark: add errorbars

analyzed by AM and MS.

Acknowledgements

This work is funded by NSF ACI-1440677 (RADICAL Cybertools: Scalable, Interoperable and Sustainable Tools for Science) and DOE ASCR grants DE-FG02-12ER26115 (AIMES). We thank Ole Weidner for preliminary prototyping and useful discussions. We thank Vivek Balasubramanian, Ioannis Paraskevakos, Nikhil Shenoy, Antons Trekalis who have been invaluable in the testing and bug-reporting of RADICAL-Pilot. We also thank Antons Trekalis for useful discussion related to the performance characterization of RADICAL-Pilot. We acknowledge support of computational time on FutureGrid, XSEDE, Amazon-EC2 grant.

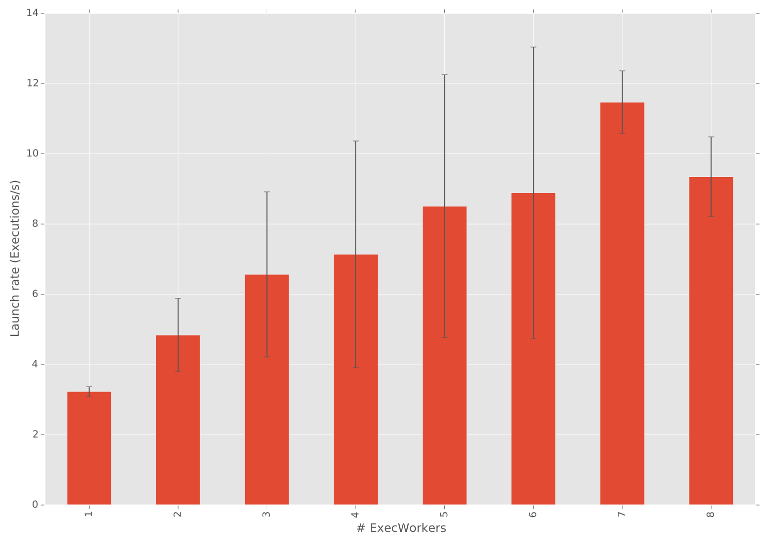

Figure 14: Rate of launching CUs (Y-axis) with a workload of 512 single core CUs with a payload of 0 seconds with a varying number of ExecWorkers (X-axis). The Pilot size is 8 nodes of 32 cores each.

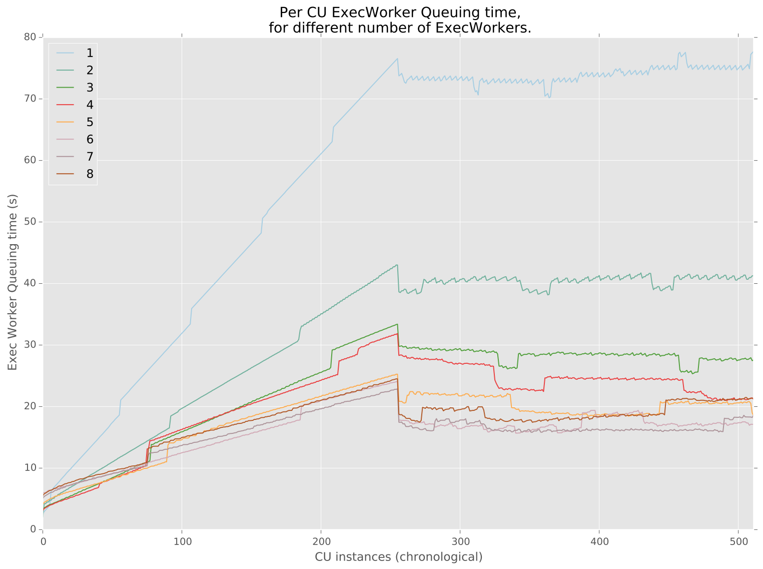

Figure 15: ExecWorker Queuing time (Y-axis) with a varying number of ExecWorker (lines) for consecutive CU launches (X-axis) with a workload of 512 single core CUs with a payload of 0 seconds. The Pilot size is 8 nodes of 32 cores each. At T=0, the scheduler assumes all cores to be available, and schedules 256 CUs onto those cores. Once all CUs are occupied (at CU=256), a plateau level is reached.