RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers

Andre Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha

RADICAL Laboratory, ECE,
Rutgers University, New Brunswick, NJ, USA

Abstract. Traditionally high-performance computing (HPC) systems have been optimized to support mostly monolithic workloads. The workload of many important scientific applications however, is comprised of spatially and temporally heterogeneous tasks that are often dynamically inter-related. These workloads can benefit from being executed at scale on HPC resources but a tension exists between their resource utilization requirements and the capabilities of HPC system software and HPC usage policies. Pilot systems have successfully been used to address this tension. In this paper we introduce RADICAL-Pilot (RP), a scalable and interoperable pilot system that faithfully implements the Pilot abstraction. We describe its design and characterize the performance of its components, as well as its performance on multiple heterogeneous HPC systems. Specifically, we characterize RP's task execution component (the RP Agent), which is engineered for optimal resource utilization while maintaining the full generality of the Pilot abstraction.

1 Introduction

Supercomputers have traditionally been designed to support applications comprised of monolithic workloads. However, applications using supercomputing resources have expanded to include workloads with spatially and temporally heterogeneous tasks that are dynamically inter-related [1,2]. Applications with such *non-traditional* workloads account for a relevant fraction of utilization [3,4], are likely to grow in importance, and they could benefit from better execution support on HPC resources.

These applications require a middleware system that can efficiently execute combinations of small and large tasks (serial vs. MPI), and combinations of short and long tasks (seconds vs. hours). Furthermore, the middleware must support dynamic relationships between tasks; allow tasks to be defined during the execution (dynamic vs. static workload); support variable tasks characteristics by utilizing heterogeneous resources.

There is a tension between the resource requirements of non-traditional workloads and the capabilities of the legacy HPC system software and its usage policies. Application-level heterogeneity and dependences requires flexible scheduling while maximization of overall HPC system utilization requires balanced resource partitioning across jobs and users. Consequently, the ability to support the resource requirements of these non-traditional workloads without compromising traditional capabilities needs careful software ecosystem design.

We propose as an effective starting point the Pilot abstraction [5]. The pilot abstraction generalizes the common concept of a resource placeholder and decouples the workload specification from the management of its execution. Pilot systems submit a placeholder job (i.e., pilot) to the scheduler of a resource. Once active, the pilot accepts and executes tasks directly submitted to it by the application. Tasks are thus executed within the time and space boundaries set by the resource's scheduler but scheduled by the application. In this way, pilot systems lower task scheduling overhead and mitigate queuing delays when executing workloads with multiple, heterogeneous and dependent tasks.

In this paper we introduce RADICAL-Pilot (RP) and provide a characterization of its performance. RP has two distinguishing features: it is a faithful implementation of the theoretically grounded concepts and capabilities underlying the Pilot abstraction as presented in Ref. [5,6]; and it supports the execution of non-traditional workloads over a wide range of HPC (and non-HPC) resources while providing both well-defined capabilities and performance.

In Section 2 we motivate RP's architecture and design. Section 3 presents a set of experiments that characterize different aspects of the performance and overheads of RP. In Section 3.5 we understand the performance of RP in light of its design. We also discuss future RP developments and optimization, based upon an understanding of the primary barriers to scaling which are derived from experiments. In Section 4 we compare and contrast our efforts with related efforts and highlight distinguishing features of RP.

2 Architecture and Design

In order to appreciate the architecture and design of RP, we present a minimal number of definitions consistent with Ref. [5].

Compute Unit / Unit	task of an application workload
Pilot-job / Pilot	placeholder job executed on a target resource
Agent	Pilot component which manages / executes Units

RP is architected (Fig. 1a) as a distributed system with a Client Module, and a set of Agents. In turn, the Client Module (or 'Client') consists of two main components, the `PilotManager` and the `UnitManager`, that manage Pilots and Units respectively. The `PilotManager` provisions Pilots on (remote) resources through interaction with the resource's Local Resource Manager (LRM). The `UnitManager` dispatches Units to Pilots for execution. Once each Pilot is bootstrapped on the target resource, it instantiates the Agent, which then receives its Units and executes them on the resource slices it holds and manages.

The design of RP is based on the central notion of Units and Pilots as stateful entities: both have a well defined state model and life cycle. RP is designed as a set of components which manage the individual states of these entities and their state transitions. In the next subsection, we describe the Unit and Pilot state models, and from that we derive the design of RP. We will then discuss how the design supports the efficient execution of application workloads.

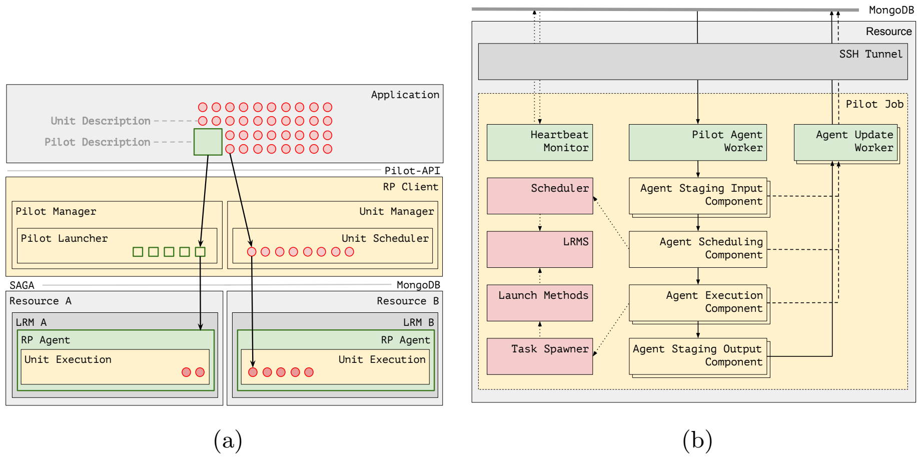


Fig. 1. **(a)** Overview of RP: an application uses the Pilot-API to create and interact with the RP Client Module. Units are sent to Pilots on target resources for execution. **(b)** RP Agent architecture showing components for communication and coordination (green), resource abstraction (red), and Unit states progressions (yellow). Solid lines show the normal state progressions, dashed lines show abnormal ones (toward `FAILED` or `CANCELED`). Dotted lines show interactions unrelated to unit state progression. Note that RP adheres to the architecture pattern for Pilot systems as defined in Ref. [5].

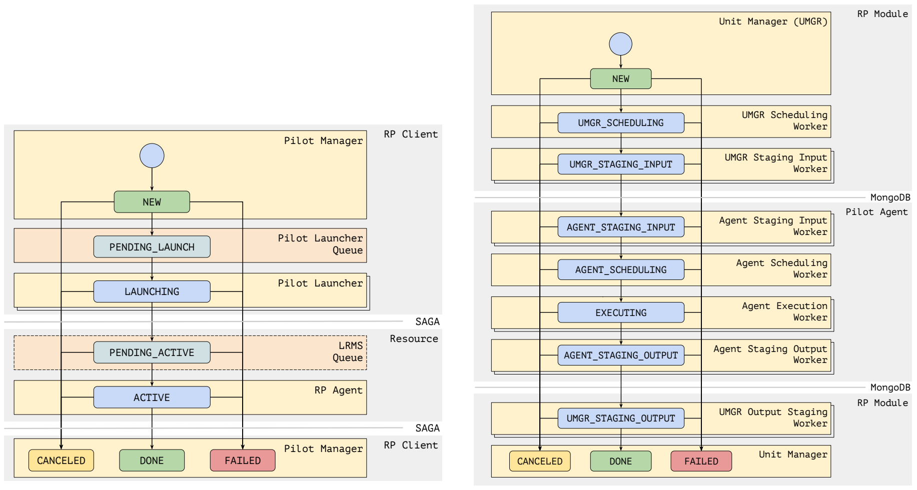
2.1 State Models

State Model for Pilots: The state model for Pilots is described in Figure 2a. When Pilots are instantiated via the `PilotManager` component, their initial state is `NEW`. The state then transitions to `LAUNCHING_PENDING`, which means the Pilots are passed on to the `PMGRLaunching` component. When that component finds a Pilot waiting to be launched, it updates the state to `LAUNCHING`.

The component interacts with the target resource's LRM via SAGA [7] to submit the Pilot for execution. Once the job submission to the LRM is successful (ie. once the Pilot is in the batch queue of the LRM) the Pilot state is updated to `PENDING_ACTIVE`. At some point in time the LRM will start the job: the RP Agent is instantiated and updates the Pilot state to `ACTIVE`. The Pilot remains in that state until it reaches a finale state: (`DONE`) when it reaches the end of its lifetime; (`CANCELED`) when the Agent is interrupted; or (`FAILED`) when the Agent experiences an error.

State Model for Units: Units progress through a base state model similar to the one of Pilots: Units are created in a `NEW` state; they are dispatched to a Pilot to be executed (in state `EXECUTING`); and they terminate in one of the final states (`DONE`, `FAILED` or `CANCELED`). The Unit base state model is expanded for two reasons: (i) Units need to be placed (scheduled) onto Pilots, and thus onto the resource slice managed by each individual Pilot, and (ii) Units require input and output staging of data.

Figure 2b shows the resulting expanded state model: a Unit is submitted via the `UnitManager` into a `NEW` state, and gets scheduled onto one of the available Pilots (`UMGR_SCHEDULING`). Once scheduled, the `UnitManager` performs Client



(a) (b)

Fig. 2. State Models: RP manages two kinds of stateful entities: Pilots (a) and Units (b). Each yellow box corresponds to a component. The transitions between states is strictly sequential. The Unit state model is shown without `'..._PENDING'` states. `PENDING` states (darker) signal that a Pilot or Unit are being communicated between RP components, and are waiting to be picked up again.

side staging of input data (`UMGR_STAGING_INPUT`), before passing control to the Agent. The Agent performs resource side input staging (`AGENT_STAGING_INPUT`), and then places the Unit onto the set of managed resources (`AGENT_SCHEDULING`). Once a suitable resource is found, the Unit is executed (`EXECUTING`). When finished, the output data staging is again split between Agent and `UnitManager` (`AGENT_STAGING_OUTPUT` and `UMGR_STAGING_OUTPUT`, respectively), before the `UnitManager` places the Unit in one of its final states. For simplicity the many `PENDING` states are left out in Figure 2b.

2.2 RP Component Design

The individual states of Pilots and Units are managed by different RP components, where each component is responsible for managing entities in one of the Unit or Pilot states. The state models diagrams in Figure 2 include the component boxes, and thus relate the state model to the design of the RP components. The components are loosely coupled, and communicate the managed entities over MongoDB (between the Client and the Agent), or via ZeroMQ [8] within the Client or Agent.

A state-model driven and component oriented design has multiple advantages: (i) the individual components have a well defined and limited semantic scope, and thus limited implementation complexity (which also benefits performance); (ii) second, the limited semantic scope makes components exchangeable, which supports the adaptation of the system towards different use cases and environments (e.g., making it easy to use a resource or application specific Unit scheduling algorithm); (iii) multiple concurrent instances can be used for each component, which then operate on multiple entities concurrently, resulting in scalable throughput; (iv) the individual components are essentially stateless, and thus provide some amount of fail safety – a failing component instance does not influence the other instances, and (v) much of the component code, specifically for communication and control flows, can be reused between different components, leading to simpler overall code and well defined optimization points.

The component based design however, incurs a component management overhead, and the bootstrap and tear-down procedures for the component network adds complexity. This cost is balanced out by the simplified code structure and, more importantly, by the scalable throughput which we will discuss in more details in Section 3.

2.3 RP Agent Design

The Agent efficiently manages and executes Units with very different characteristics, on a range of hardware and software platforms. It provides a number of abstractions which are used by the Agent components (see Fig. 1b), to keep them functioning mostly independent of the specific Unit and resource configurations. Figure 1b shows the primary abstractions (red): `LRM`, `Scheduler`, `LaunchMethod`, and `TaskSpawner`. These abstractions enable RP to run on heterogeneous infrastructures, and make it easily portable to new ones.

LRM The `LRM` abstraction enables the interaction of the Pilot with the resource's batch queue system. This abstraction provides information about the set of cores the Pilot can utilize: their number and distribution, their connectivity, etc. The currently supported systems are: `TORQUE`, `PBSPro`, `SLURM`, `SGE`, `LSF`, `LoadLeveler`, `CCM` (a cluster abstraction layer for CRAYs), and `Fork`.

Scheduler The Agent scheduler places incoming Units on the set of available cores, and keeps track of the core utilization. Cores are reclaimed when Units finish, and then reallocated to the next set of Units. MPI units spanning multiple nodes are placed on topologically close nodes to minimize intra-Unit communication times. Multithreaded (non MPI) Units are confined to a single multicore node. The schedulers currently supported are `CONTINUOUS`, which considers the set of available cores to be a continuous set, and `TORUS`, which handles nodes laid out in an n-dimensional torus, as done for example by IBM BG/Q machines.

LaunchMethod Each system offers specific and multiple mechanisms to launch executables on allocated nodes, often different for MPI and non-MPI codes. On many systems, administrators install their own, subtly different tool for launching processes, often motivated by the integration of custom accounting mechanisms. That layer is abstracted in RP by a set of `LaunchMethod` classes. These classes produce the required command for process launching for each given Unit and core allocation. RP supports the following LaunchMethods: `MPIRUN`, `MPIEXEC`, `APRUN`, `CCMRUN`, `MPIRUNCCMRUN`, `RUNJOB`, `DPLACE`, `MPIRUNRSH`, `MPIRUNDPLACE`, `IBRUN`, `ORTE`, `SSH`, `POE`, and `FORK`. Additional launch methods required to support new platforms and resources can be easily added.

TaskSpawner The command produced by the `LaunchMethod` is issued when the core allocation for a Unit becomes active and the application workload needs to be executed. The `TaskSpawner` abstraction of RP supports different mechanisms to issue launch commands, each with specific performance and scalability properties. RP supports `POPEN` and `SHELL` based spawning mechanisms. The former is based on Python's process management capabilities, the latter uses `/bin/sh` as process manager. The `TaskSpawner` is a point of tight interaction with the compute node's operating system: the `TaskSpawner` establishes the process environment for the Unit executables, and watches their execution. Once Unit execution finalizes, the `TaskSpawner` collects the exit code and communicates the freed cores as 'available' to the scheduler.

RP Configuration: In order to balance performance requirements with the ability to manage heterogeneity of platforms and workloads, the exact set and cardinality of Agent components, the specific implementation of the abstractions, are configured at runtime, and can vary from Pilot to Pilot. The configuration is managed via configuration files which RP provides for our most commonly used resources (e.g. belonging to XSEDE, NCSA, NERSC, ORNL). Configurations for other resources can be added by users. Each configuration can also be changed at runtime on a per-application or per-pilot basis. This couples system expertise with application level flexibility.

The Pilot configuration includes the ability to specify the number, and also the placement, of the Agent components (see Fig. 1b). Depending on the target resource architecture, the RP Pilot components can be placed on cluster head nodes, mom nodes, compute nodes, or any combination thereof. We use the notion of *Sub-Agents* to refer to a set of components running on one specific node. A set of ZeroMQ communication bridges ensures that the Units transition through the resulting distributed component network. Section 3 will discuss the implications of different configuration options for the resulting RP performance.

3 Experiments to Characterize Performance

Experiments are designed to profile RP's performance and characterize its dependence on different static and runtime configuration variables, and thus provide qualitative validation of the architecture and design of section 2.2. We investigate the performance of RP at three levels: (i) **Component Level**, where we use micro-benchmarks to measure the performance of *individual* and *isolated* RP components when under stress; (ii) **Agent Level**, where we measure how the Agent components perform *collectively* under a synthetic workload (the observed performance is then related to individual and aggregated measurements obtained at Component Level); and (iii) **Integrated Level**, where we measure the complete RP system performance as perceived by end-user applications.

The experiments focus on characterizing the Agent and its components (see yellow boxes in Figure 1b) which are crucial for the overall workload execution performance. Outside of the Agent, data transfer times, Client-Agent communication delays and batch queue times for the Pilot jobs are most relevant to the overall RP performance. Those depend on external factors such as network bandwidth and latency, resource configuration and usage policies, and are not directly under RP control[1].

Our experiments are performed on three distinct HPC resources, which are representative of the range and type of resources available to the computational science community:

– ***Comet*:** a 2 PFLOP cluster at SDSC, with 24 Haswell cores 128GB RAM per node (6,400 nodes), Infiniband, Lustre shared filesystem (FS).
– ***Stampede*:** a 10 PFLOP cluster at TACC, with 16 Sandy Bridge cores / 32GB RAM per node (1,944 nodes), Infiniband, Lustre shared FS.
– ***Blue Waters*** a multi-PFLOP Cray at NCSA, with 32 Interlago cores / 50GB RAM per node (26,864 nodes), Cray Gemini, Lustre shared FS.

RP is instrumented with profiling probes, which record event times for all RP's operations. The events are recorded and are written to disk; utility methods are used to fetch and analyze them. Like most system profiling mechanisms, RP's profiling is designed to have minimal effect on the runtime, keep the code footprint and complexity small. Due to space constraints, we will not present experiments to verify the claim, but provide circumstantial verification via a benchmark that has been run with and without profiling. In its smallest and simplest configuration this benchmark measured $144.7 \pm 19.2s$ and $157.1 \pm 8.3s$ when running with and without profiling, respectively.

3.1 Performance Metrics and Overheads

We use two metrics to characterize RP's performance: a subset of the total time to completion (*ttc*) we call ttc_a, and Resource Utilization. The former is a measure of temporal efficiency, the later of spatial efficiency.

ttc_a is the subset of *ttc* spent by the Agent to manage and execute Units. ttc_a is the time span between the first Units entering `AGENT_STAGING_INPUT` state, and the last Unit leaving `AGENT_STAGING_OUTPUT` state (Fig. 2b). ttc_a isolates the elements of *ttc* that depend exclusively on RP. The other elements or *ttc* depends instead on resource or application properties like queue time, core/FS speed, Unit execution time, Unit ordering, or Pilot layout and placement.

Resource Utilization is the amount of resource (cores) used at any point in time during workload execution by RP. Resource here indicates the slice of a machine held by a Pilot and workload execution refers to executing Units. Resource Utilization is thus a function of how many Units enter the state `EXECUTING`, how many Units enter a final state, and how many Units are in the `EXECUTING` state at any point in time during the Pilot life time (Fig. 2b). We name these: Unit Startup Throughput, Unit Collection Throughput, and Unit Concurrency.

[1] Note that while we ignore data *transfer* times, the data *staging* components are still investigated for their base performance (creation of working directories, management of standard output and error files, etc).

Ideally, the Agent would be able to immediately use all cores, keep all cores busy until all Units are done, and then immediately free all cores. In practice, system and implementation bottlenecks cause cores to be under-utilized. Component Level micro-benchmarks identify those bottlenecks while the Integrated and Agent Level experiments measure how they influence Resource Utilization.

Due to space constraints, our experiments will not investigate all involved overheads. Specifically, we will not discuss Pilot submission delays, Pilot bootstrapping time, Unit submission delays, etc. In short, the experiments assume that the Agent is `ACTIVE` and has a sufficient part of the workload available for execution.

3.2 Micro-benchmarks

Micro-benchmarks measure the performance of individual RP components *in isolation*. To do so, experiments are carefully designed to capture the performance of exactly one component under a non-perturbative workload. Although RP is instantiated as normal, including running a complete Pilot on the target resource, a single, light-weight Unit is pushed through the system as opposed to a set of Units of a regular workload. The incoming Unit is cloned a specified number of times (10k if not mentioned otherwise), and operated on by the component under investigation. All clones are dropped immediately when they leave that component, ensuring that downstream components remain idle. The resultant effect is that any specific component can be stressed in isolation and is investigated without influence of other RP components, under load conditions that are locally full and realistic.

This approach has two side effects: RP components and communication channels do not compete for shared system resources; and RP components cannot be starved due to a performance bottleneck arising upstream. The micro-benchmark thus measures an *upper bound* for individual component performances, and verify that the individual components does not limit the overall throughput of the RP system. Limitations from competition for system resources between different components are measured by the Agent and Integrated benchmarks later on.

We perform micro-benchmarks for all Agent components: `AgentStagingInputComponent`, `AgentSchedulingComponent`, `AgentExecutingComponent`, and `AgentStagingOutputComponent`. We compare the performance of these components for three different resources (*Stampede*, *Blue Waters*, and *Comet*), and for several load-sharing setups (1, 2, 4, 8 component instances per Sub-Agent (CI); 1, 2, 4, 8 Sub-Agent instances (SA)). These configurations span a large parameter space for the experiment, thus it is not possible to present the full set of results; we will focus on those which best illustrate the overall behavior of RP, for the full set of profiling data and plots, please refer to [9].

We investigate how quickly the components reach steady throughput, what that throughput is, how stable it is, and how it depends on the experiment configuration. Specifically, we discuss the strong scaling behavior of the individual components under an increasing number of concurrent component instances, and the dependency of that scaling on component placement.

To put the observed component performance in relation to the RP implementation, we note that an empty component, i.e. a component which only advances unit states without performing any additional work, can operate on approximately 1 000 *units/s*. That performance scales linearly with the number of concurrent component instances until the ZeroMQ bridges' throughput is saturated, at about 8 000 *units/s*.

Agent Scheduling Performance Currently, RP has only one instance of the `AgentSchedulingComponent`. The Agent scheduling is compute and communication bound: the algorithm searches repeatedly through the list of managed cores; core allocation and de-allocation are handled in separate, message driven threads. Figure 3 shows how the component performs on three resources: the scheduler throughput stabilizes very quickly in all three cases, but the absolute values differ significantly, presumably due to differences in system performance (the RP configurations are identical).

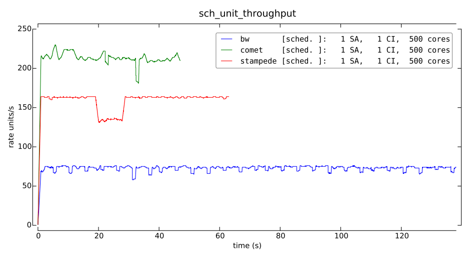


Fig. 3. Scheduler throughput is stable over time, but differs per resource.

Agent Input-Staging Performance The `AgentStagingInputComponent` has to perform several activities in preparation for the Unit execution: it creates the workdir, copies input data to that workdir if needed, or creates symlinks for shared data. We motivated in Section 3 the absence of any data transfer in our experiments, thus the functionality of the component reduces to the creation of the Unit's working directory.

Figure 4a shows the throughput of one component instance on the three resources. While the average throughput is again reached very quickly, both *Comet* and *Blue Waters* show significant noise in their performance. We consider this to be an artifact of the Lustre shared FS behavior: the performance of repeated `'mkdir'` calls on command line shows similar jitter.

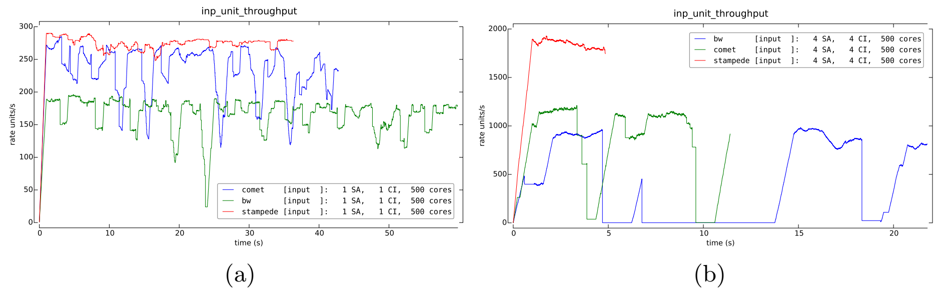
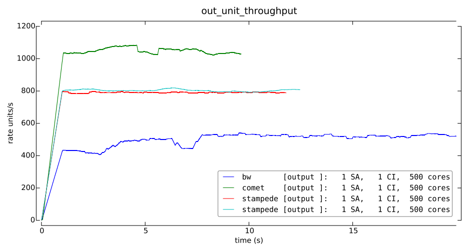
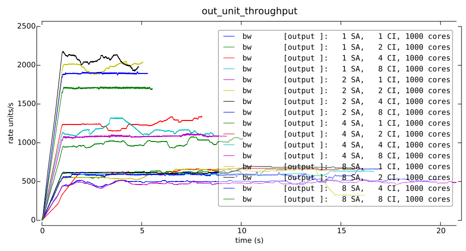


Fig. 4. (a) The throughput of a `AgentStagingInputComponent` instance is mostly limited by FS operations (e.g. `mkdir`). It varies over different resources, both in jitter and absolute value. **(b)** The scaling of input staging operations with number of Sub-Agents and components differs significantly over resources. *Stampede* shows some scaling while *Blue Waters* and *Comet* 'choke' on large numbers of components (shown here: 16).



(a)



(b)

Fig. 5. (a) The output staging performance is less jittery than for input staging, presumably because of FS level data caching. The baseline throughput differs again for different resources. **(b)** For *Blue Waters*, output staging scales nicely with the number of staging components, independent mostly independent if those components are on the same, or distributed over different Sub-Agents (and thus compute nodes).

Figure 4b shows how the throughput of the component changes when we use four Sub-Agents (which run on 4 different compute nodes), and each of the Sub-Agents uses 4 instances of the `AgentStagingInputComponent`.

While the throughput does not scale linearly with the total number of component instances (16), it *does* show a significantly increased maximum throughput, which in the case of *Stampede* is also sustained for the duration of the experiment. For *Comet* and *Blue Waters*, the maximum rate is only achieved in certain time intervals, where otherwise the throughput falls sharply to zero, indicating an overload of the FS due to the concurrent operations.

Agent Output-Staging Performance The `AgentStagingOutputComponent` is expected to behave similarly to the `AgentStagingInputComponent`, but where the latter is constrained by the performance of *writing* data to the shared FS, the former is expected to be constrained by the *read* performance of the FS. In our experiments, which exclude actual data transfers, the task of the component is only to read very small `stdout` and `stderr` files from the FS. We are therefore stressing the metadata component of the FS more than its actual read performance.

Figure 5(a) shows the throughput of one component instance on the three resources. We observe much less jitter than on writing to the FS, and also constantly higher throughput for all three machines. Read caches are much easier to achieve on reading from a FS, and we suspect that most of the throughput increase can indeed be accounted to good FS level caching.

Figure 5(b) focuses on the specific scaling behavior on *Blue Waters* when varying the number of Sub-Agents and staging components. For 1 and two Sub-Agents, the throughput does not vary significantly, and is not at all dependent on the number of components used. When using 4 or 8 Sub-Agents, we do observe a good scaling of throughput, indicating that the shared FS is then able to load balance between compute nodes. Specifically for 8 Sub-Agents, we see a *negative* scaling with the number of staging components: more components decrease the overall throughput, indicating a saturation of the FS on the individual nodes.

Similar scaling behavior is observed for *Comet* and *Stampede*, although the negative scaling over the component numbers is much less prominent there.

Unit Execution Performance The Unit execution performance, i.e. the process of spawning the application tasks, is where the resources are observed to differ significantly, both in their basic performance numbers, and also in the scaling behavior. Figure 6(a) shows the throughput for 1 Sub-Agent and 1 component instance. *Blue Waters* is observed to have a very consistent, but low rate of about 15 Units/s. *Comet* shows a significantly higher rate, which again varies significantly over time. *Stampede* has a relatively high rate of more than 150 Units/s, but with less jitter than *Comet*.

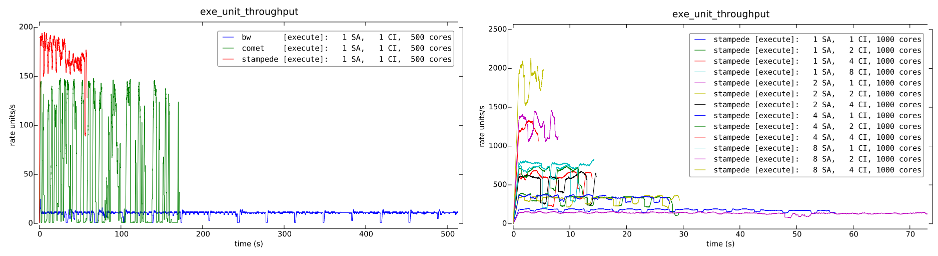
Figure 6(b) shows the scaling behavior for *Stampede*: the throughput scales with both the number of Sub-Agents and the number of components per Sub-Agent. Specifically, the combination of 8 Sub-Agents with 2 components performs similar to the combination of 4 Sub-Agents with 4 components, suggesting that the scaling is independent of the component placement. The 8 Sub-Agents / 8 components configuration achieves up to $2\,000 units/s$ throughput, but at that point the jitter begins to increase compared to the smaller configurations.

We investigated the scaling of throughput with Sub-Agents and components on *Blue Waters* (no figure shown); the jitter increases very quickly and the average throughput increases by up to a factor of 2.5.

3.3 Agent Performance

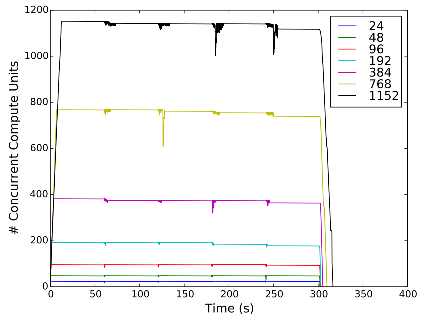
While the micro-benchmarks describe the specific performance and scaling of individual RP components, they do not describe the overall Agent behavior. In principle, the Agent performance upper bound is the performance of its slowest component but, in practice, the Agent performance is lower. This is due to at least three reasons: (i) the micro-benchmarks neglect the influence of communication between components; (ii) the concurrent operation of multiple components introduces competition for shared system resources (e.g., both input and output staging compete for FS resources); and (iii) the Agent performance can be limited by slower RP Components or system resources *outside* of the Agent scope.

The set of Agent level experiments discussed in this subsection investigate the contributions of (i) and (ii). To neutralize (iii), we ensure that the Agent

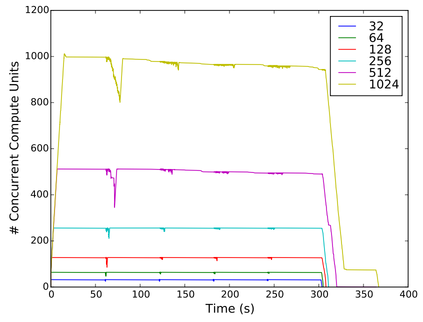


(a) (b)

Fig. 6. **(a)** The throughput of a single execution component shows very large variation over resources – both in jitter and absolute values. **(b)** For *Stampede*, execution throughput again scales with the number of component instances, independent of their placement.



(a)



(b)

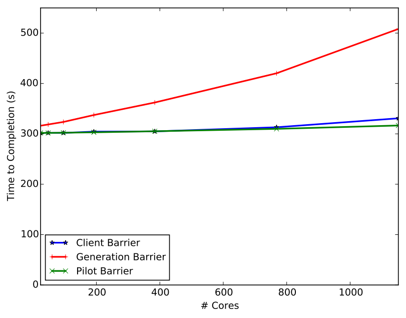
Fig. 7. Results showing the number of concurrent Units that the Agent can sustain (y-axis), and the resulting ttc_a (x-axis). Results are shown for varying pilot sizes on *Comet* (a) and *Blue Waters* (b). Utilization deteriorates somewhat for large pilots.

operates in isolation, i.e. it is independent of the performance of the Client side RP components and system. Specifically, we ensure that the Agent receives sufficient work to fully utilize the Pilot's resource slice, by introducing a startup barrier in the Agent: it only starts to process Units once the Client has pushed the complete workload into the database.

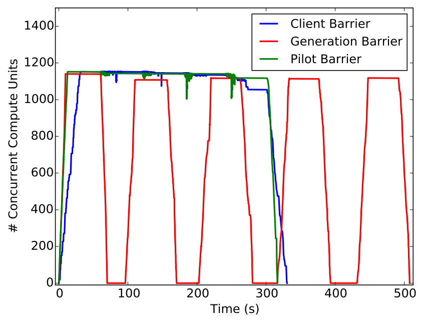
For these experiments we run a workload with single core, 60 second tasks with 5 generations (meaning it is 5 times the size that would fit concurrently on the resource) and we vary the number of cores allocated to the Pilot. The optimal ttc_a for all these workloads would be 300 seconds (5 generations x 60 seconds). The configuration of the Agent is, except for the number of cores available to the tasks, constant: it uses 10 Sub-Agents on individual worker nodes, each with a single Executing Component. All other components are co-located with the main Agent instance.

The x-axis of Figures 7a and 7b shows that the optimal ttc_a value of 300 seconds is achieved for up to about 384 and 256 cores on Comet and *Blue Waters* respectively. For higher core counts, *Blue Waters* introduces increasingly significant overhead. The y-axis of the same figures shows that the Pilot's ability to achieve optimal task concurrency on the allocated resources decreases with scale. As the core count increases, the Agent has increasing trouble keeping all cores continuously busy, especially at the transition from one generation to the next.

While the Agent seems to recover from the increased load of generation transitions on *Comet* quickly (Figure 7a), *prima facie* that is not the case on *Blue Waters* (Figure 7b). We have seen in the micro-benchmarks, that the `AgentExecutingComponent` throughput on Cray's does not scale very well: indeed the observed maximum value of about 40 $units/s$ limits the ability of the Agent to achieve a quick exchange of the Unit generations. Starting at 512 cores, the Agent can not collect finished Units quickly enough, thus is not able to re-use the cores for the next generation in time. This effectively results in an additional 'generation' to be executed.



(a)



(b)

Fig. 8. (a) ttc_a of a workload consisting of 5 generations on a varying number of cores (24, 48, 96, 192, 386, 768, 1152) for the 3 barrier scenarios. **(b)** Unit concurrency of the 3 scenarios for 1152 cores: the Generation barrier introduces communication delays.

3.4 Integrated Performance

In the following experiments which are designed to investigate the characteristics of RP as perceived by a end-user, we change the way in which the workload reaches the Agent: rather than using an Agent-side barrier to isolate the Agent components from the influence of the Client-side components and the Client-Agent communication layer, we look into different types of Client-side barriers, which more closely represent that real application submit Units distribution over time. For each experiment, we vary the pilot sizes to reach at least 1K cores, while keeping the total numbers of Units constant for a given pilot size.

In the first scenario we re-use the configuration from the Agent level experiments, i.e. all workload (of all generations) is available at Agent start (*'Pilot-barrier'*). In the second scenario, that order is reversed, i.e. the Agent is started first and then the Client starts to feed the workload to the Agent (*Client-barrier*). In the third and last scenario, the Client creates a barrier after every generation, and it does not start to feed the next generation to the Agent before all Units of the previous generation are completed (*Generation-barrier*). We use the same workload as in the Agent-level experiments, thus the optimal performance for this workload would again be $300s$.

Figure 8a shows that the performance difference between the Pilot-barrier and Client-barrier are negligible for smaller core counts, but becomes noticeable for more than 1k cores. When we focus on the 1152 core experiment in Figure 8b we see that the offset is primarily caused by the different unit startup rates, which is explained by the fact that in the Pilot-barrier scenario all workload is ready to run at the Agent side while in the Client-barrier scenario the workload still needs to be communicated to the Agent.

The performance of the Generation-barrier, that already shows considerable overhead with the smaller experiments too. The detailed plot in Figure 8b shows prolonged periods of core idleness between the generations. During this period the unit statuses are communicated back to the Client and the workload of the next generation is submitted to the Agent, and the communication delay is what causes the gaps. The communication load increases with the number of units, and thus with the number of cores, which explains the growth of the overhead at increasing core counts. The startup rate of the Generation-barrier is consistent with the Client-barrier though, indicating that, once notified about completed units, the Client is able to communicate the next generation's work quickly enough to keep the Agent components busy.

3.5 Discussion

The micro-benchmarks offered detailed insight into the performance characteristics of the individual RP components on the target resources, and their dependencies on the different Agent configuration options. They provide a theoretical upper-bound on the performance of RP. We find unit throughput is system dependent, and is determined by system performance limitations. All micro-benchmarks yielded limits that were well below the values measured for an empty component, confirming that the component based implementation concept does not impose a fundamental performance limitation at this point.

The Agent level benchmarks discern performance when RP components are used in orchestration, and how the performance measured in the micro-benchmarks relate to the overall Agent performance. The availability of profile data allow us to identify bottlenecks and viable scaling approaches, and thus to optimize the Agent layout (component number and distribution) for each individual target resource. They indicate where the components can and should be distributed over Sub-Agents, and how many component instances are useful for a specific resource.

Both micro-benchmarks and Agent-level benchmarks show that RP's performance and scaling properties will be resource dependent. We plan to address this in two ways. We have seen that the FS interactions imply significant throughput limitations. To address this, we will reduce the dependency on shared FS by moving some FS interactions to local, temporary storage or to memory. Our experiments also suggest that process spawning and management overhead is a significant bottleneck, specifically on the Cray machines. We plan to replace the centrally managed process spawning with a more distributed approach, which replaces at least some of the processes forks with library calls.

The integrated benchmarks show that the Agent performance translates into application performance, but they also show the influence of communication delays on the overall system utilization. This reiterates the need for applications to communicate workload scheduling decisions to the target resources efficiently.

Put together, these experiments suggest several improvements in implementation: First, the current implementation uses an inefficient protocol to communicate Unit state changes to the application, which is partially responsible for the inter-generation delays seen in the integrated experiments. Second, there is scope for distributed scheduling algorithms for common workload types, to free the application from the need to re-implement the (non-trivial) scheduling and communication schemes. This is possible due to the modularity of the RP design which allows the spread of the agent scheduling algorithm between the Client module, i.e. the `UnitManager` Scheduler, and the Agent, i.e. the `AgentScheduler`.

4 Related Work and Contributions

Falkon [10] ("Fast and Light-weight tasK executiON framework") is an early example of a pilot system for HPC environments. The Falkon architecture consists of Executors on every compute node and a central Dispatcher that manages the tasks. Falkon's design and performance are optimized to support serial/single node applications [10]. A single Falkon executor is measured to be able to manage between 7 and 28 *units/s*, and scales up to about 400 *units/s* for 50 executors.

The pilot system in Fireworks [11] is called Rockets, which has a logical equivalent to Falkon's Executor. The performance for a single executor ('Firework') is is about 10 *units/s*, and it operates at the level of a node. An intrinsic limitation of the one executor per node granularity used by Falkon and Fireworks is the inability to manage multinode MPI Units.

In contrast, RP supports any size of Units as long as they fit into the Pilot. RP components are more fine grained enabling load balancing per Unit state. While individual RP components seem to have comparable throughput in isolation, we observe a wider range of scaling behavior, depending on the specific combination of component, resource, and Agent configuration. Data show that parts of the observed RP scaling is indeed related to competition for system resources. In the absence of control experiments on the same hardware platform however, it is difficult to estimate the precise reasons underpinning the performance difference between Falkon, Rockets and RP.

Another system designed to execute many small tasks on HPC is CRAM[2]. CRAM enables the packing of many concurrent and independent MPI jobs into a single job to be executed on an IBM BG/Q system. Even though CRAM has impressive performance numbers, it is limited to a specific system (IBM) and requires the workload to be known in advance, i.e., the job packaging is static.

Pilots have proven very successful at supporting distributed applications. For example, the Open Science Grid (OSG) [12] deploys HTCondor and a Pilot system named "Glidein" to access 700 million CPU hours a year for applications requiring high-throughput of single-core tasks. Pilots are also used by several user-facing tools to execute workflows. Swift implements pilots in a subsystem named "Coasters" [13] and Pegasus uses Glidein via providers like "Corral" [14]. With the possible exception of Falkon and Rockets, most pilot systems were originally developed in a distributed systems context optimized for typical distributed workloads. As such, existing pilot systems require significant engineering in order to flexibly execute workloads on diverse HPC platforms.

RP's unique contribution arises from being an implementation of a well-defined pilot abstraction [5,6] and capable of supporting a wide set of application on resource types ranging from distributed to HPC platforms. We recapitulate how RP is designed to meet these design objectives.

MPI execution abstraction: RP supports all MPI startup versions, such as `aprun`, `ibrun`, `mpirun` (different implementations), `mpiexec`, `orte-submit`. It can be extended to support other task startup mechanisms via plugins.
Heterogeneous and Distributed Resources: RP uses the SAGA [7] resource interoperability layer to access a variety of resources, data management mechanisms and protocols. SAGA supports resource access via `ssh` and `gsissh`, and data transfer via `scp`, `sftp`, and `GlobusOnline`. SAGA also supports resources with a variety of batch and job management systems, such as `PBS`, `Torque`, `Slurm`, `SGE`, `Condor`. RP also supports non-HPC resources, such as Condor and AWS. The resource heterogeneity is localized to the SAGA layer, can be extended to additional/new resources via SAGA plug-ins without refactoring of the other RP components.
Modularity: all semantic components of RP are stateless and pluggable, and can be transparently exchanged. This makes RP adaptable to different workload and resource types. Units for a given workload can vary over task sizes and types. Tasks can be scalar, MPI, OpenMP, multi-process, multi-threading support, have shared or non-shared data.
User-level bootstrapping: A fundamental design decision is to to make RP operate completely at the user-level; no system level services need to be installed or configured. This enables the use of diverse resources with minimal requirements on the resources' software stack.
Application Level Scheduling: Pilot systems support dynamic workload execution; RP also supports custom scheduling algorithms, both for workload (Unit) execution and pilot agent management.

Arguably, this paper provides a first attempt to characterize the performance of a pilot system and use a semi-empirical discrete event approach to engineer and optimize a pilot system.

5 Conclusion

Given the diversity of current [1] and future workloads that will utilize HPC systems (see Ref. [15] for a recent analysis on NERSC systems), the efficient, flexible and scalable execution of spatially and temporally heterogeneous interrelated tasks is a critical requirement. *Prima facie*, a system implementing the Pilot abstraction [5,6] provides the conceptual and functional capabilities needed to support the execution of such workloads. The impact of an abstraction however, is only as good as its best implementation. Whereas there are multiple pilot systems, they are mostly partial implementations of the Pilot abstraction or implementations geared towards specific functionality and platforms.

Against this backdrop, RADICAL-Pilot brings together recent conceptual advances [5,6] with advances in systems & software engineering, and open source and community best practices. In addition to describing the architecture and implementation of RADICAL-Pilot (Sec. 2 and Fig. 1a), this paper profiles and characterizes its performance on several distinct HPC platforms (Sec.3).

Although the focus of this paper has been on the direct execution of workloads on HPC machines, RADICAL-Pilot also serves as a vehicle for research in distributed [16] and data-intensive scientific computing [17]. RP also forms the middleware system for a range of high-performance application libraries [18,19,20,21] many of which are already in production use.

RADICAL-Pilot is available for immediate use on most contemporary platforms [22]. RADICAL-Pilot source is accompanied with extensive documentation and an active developer-user community.

Acknowledgements: This work is supported by NSF "CAREER" ACI-1253644, NSF ACI-1440677 RADICAL-Cybertools and DOE Award de-sc0008651. We acknowledge access to computational facilities on XSEDE resources via TG-MCB090174, and support of NSF-1516469 which provides access to NSF's Blue Waters. We thank members of the RADICAL group for useful comments, feedback and testing. Many thanks to the OpenMPI developers for discussions and support.

Author Contribution: Andre Merzky is the lead developer of RADICAL Pilot, designed/performed the micro-benchmark experiments as well as wrote the bulk of this paper. Mark Santcroos is senior developer of RADICAL Pilot, is responsible for the implementation and optimization of the Agent abstraction layers, including the ORTE access layer. He designed and executed the Agent level and Integrated experiments. Matteo Turilli has played an important role in the testing and design discussions of RADICAL Pilot. Shantenu Jha is the project lead.

References

1. Lavanya Ramakrishnan Jay Srinivasan, Richard Shane Canon. My Cray can do that? Supporting Diverse Workloads on the Cray XE-6. https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap157.pdf.
2. J. Gyllenhaal, T. Gamblin, A. Bertsch, and R. Musselman. Enabling High Job Throughput for Uncertainty Quantification on BG/Q. In IBM HPC S ystems Scientific Computing User Group (ScicomP14), Chicago, IL, 2014.
3. Brian Austin, Tina Butler, Richard Gerber, Cary Whitney, Nicholas Wright, Woo-Sun Yang, and Zhengji Zhao. *Hopper Workload Analysis.* May 2014.
4. Thomas R. Furlani, Barry L. Schneider, Matthew D. Jones, John Towns, David L. Hart, Steven M. Gallo, Robert L. DeLeon, Charng-Da Lu, Amin Ghadersohi, Ryan J. Gentner, Abani K. Patra, Gregor von Laszewski, Fugang Wang, Jeffrey T. Palmer, and Nikolay Simakov. Using xdmod to facilitate xsede operations, planning and analysis. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, XSEDE '13, pages 46:1–46:8, New York, NY, USA, 2013. ACM.
5. Matteo Turilli, Mark Santcroos, and Shantenu Jha. A Comprehensive Perspective on Pilot-Abstraction, 2015. `http://arxiv.org/abs/1508.04180`.
6. Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. P*: A model of pilot-abstractions. *2012 IEEE 8th International Conference on E-Science*, pages 1–10, 2012. http://doi.ieeecomputersociety.org/10.1109/eScience.2012.6404423.

7. Andre Merzky, Ole Weidner, and Shantenu Jha. SAGA: A standardized access layer to heterogeneous distributed computing infrastructure˙ *Software-X*. DOI: 10.1016/j.softx.2015.03.001.
8. ZeroMQ – Distributed Messaging, `https://www.zeromq.org/`.
9. RP Experiment Repository, Micro Benchmarks, `https://github.com/radical-experiments/rp-paper-2015-micro/`.
10. Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of the 8th ACM/IEEE conference on Supercomputing*, page 43. ACM, 2007.
11. Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. FireWorks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 2015.
12. Ruth Pordes et al. The open science grid. *J. Phys.: Conf. Ser.*, 78(1):012057, 2007.
13. Mihael Hategan, Justin Wozniak, and Ketan Maheshwari. Coasters: uniform resource provisioning and access for clouds and grids. In *Proc. 4th IEEE Int. Conf. on Utility and Cloud Computing*, pages 114–121, 2011.
14. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahl, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
15. Gonzalo Pedro Rodrigo Álvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Hpc system lifetime story: Workload characterization and evolutionary analyses on nersc systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 57–60, New York, NY, USA, 2015. ACM.
16. Integrating Abstractions to Enhance the Execution of Distributed Applications, Matteo Turilli, Feng (Francis)Liu, Zhao Zhang, Andre Merzky, Michael Wilde, Jon Weissman, Daniel S. Katz, Shantenu Jha, accepted for IPD PS'16 (Chicago), available at arXiv:1504.04720 `http://arxiv.org/abs/1504.04720`.
17. SC15 Tutorial: Data-Intensive Applications on HPC Using Hadoop, Spark and RADICAL-Cybertools. `https://github.com/radical-cybertools/supercomputing2015-tutorial/wiki`.
18. ExTASY Project. `http://www.extasy-project.org/`.
19. Brian K. Radak, Melissa Romanus, Tai-Sung Lee, Haoyuan Chen, Ming Huang, Antons Treikalis, Vivekanandan Balasubramanian, Shantenu Jha, and Darrin M. York. Characterization of the Three-Dimensional Free Energy Manifold for the Uracil Ribonucleoside from Asynchronous Replica Exchange Simulations. *Journal of Chemical Theory and Computation*, 11(2):373–377, 2015.
20. Replica-Exchange Framework. `https://github.com/radical-cybertools/radical.repex`.
21. SPIDAL Project. `http://www.spidal.org/`.
22. RADICAL-Pilot Github Project. `https://github.com/radical-cybertools/radical.pilot`.