2016 IEEE International Conference on Robotics and Automation (ICRA)
Stockholm, Sweden, May 16-21, 2016

RoboBench: Towards Sustainable Robotics System Benchmarking

Jonathan Weisz, Yipeng Huang, Florian Lier, Simha Sethumadhavan, and Peter Allen

***Abstract*— We present RoboBench, a novel platform for sharing robot full-system simulations for benchmarking. The creation of this platform and benchmark suite is motivated by a need for reproducible research. A challenge in creating a full-system benchmarks are incompatibilities in software created by different groups and the difficulty of reproducing software environments. We solve this problem by using software containers, an emerging virtualization technology. RoboBench enables sharing robot software in a runnable state, capturing the software behavior of robots carrying out missions. These simulations make clear the performance impact and resource usage of programs and algorithms relative to other software involved in the mission. These containers are integrated with the CITK platform for reproducible research, which automates generation and publishing of the containers. We present an overview of the system, a description of our prototype set of benchmark missions, along with a validation study comparing the computational load profile of a mission performed on a real and simulated robot. Additionally, we present preliminary results of an overall analysis of the benchmarks in the RoboBench suite, showing where computational work is expended in robotics common robotics tasks. RoboBench is extensible, and is the first step toward a robust, quantitative approach to engineering computationally-efficient robots.**

I. Introduction

Benchmarking has been central to improvements in a number of computer science disciplines, by making possible repeatable measurements to quantify improvements. Currently, comparative research in robotics has focused on competitions between systems [3][4], and on benchmarking *capabilities* [15][19], both of which have shortcomings in their ability to advance research. Competitions such as RoboCup lead to robot systems that excel at certain missions, and teams are required to share binaries and/or source code. But vital information such as detailed build instructions, and the system's required library versions are insufficiently shared to allow replication of results long after the competition. In terms of capabilities benchmarks, input datasets and comparison tools have helped comparing algorithms within domains, leading to immense improvements in individual subsystems—indeed, this is the current definition of benchmarking in robotics. However, benchmarks for subsystems do not test software running as part of a whole robot, and are unable to shed light on how to optimize systems as a whole.

We advocate for a study of robot *systems*, which takes into account the cost of computing over the course of robot missions, optimizing robotics algorithms from the perspective of covering the most used functions first, while paying attention to costs incurred in infrastructure such as ROS and networks. To make repeatable, quantitative studies of systems possible, we need an ability to share fully built systems that are ready to run missions.

This work has been supported by National Science Foundation grants CNS 1239134, IIS 1208153, and a fellowship from the Alfred P. Sloan Foundation. J. Weisz, Y. Huang, S. Sethumadhavan and P. Allen are with the Department of Computer Science, Columbia University, E-mail: jweisz, yipeng, simha, allen@cs.columbia.edu. F. Lier is with CITEC, Bielefeld University. E-mail: flier@cit-ec.uni-bielefeld.de.

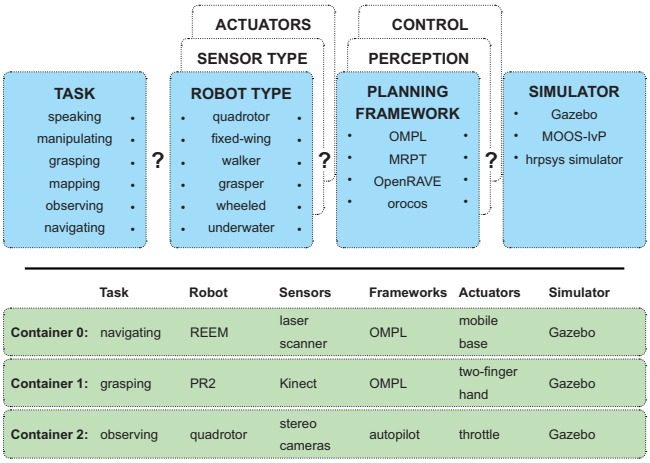

Fig. 1. Containers capture a fixed combination of software that form a simulation. In the top part of this figure, we see mission tasks, robot subsystems, and simulators form combinatorially many designs, allowing for high flexibility for researchers. But the open-ended framework presents a high entry barrier for peers to replicate systems for benchmarking. Bottom, successfully working simulations with a fixed set of software are distributed as containers, which are guaranteed to run on other users' computers.

Researchers need a robotics software benchmark suite, consisting of fully built robot simulations, that can be shared with non-experts in robotics to study the overall properties of its software. These properties include a breakdown of algorithms involved, and an analysis of the hardware requirements of the software. Studying such as benchmark suite would guide research in what software needs to be optimized first, and what specialized hardware may be useful for supporting robotics.

We present RoboBench, a benchmarking platform integrated with the CITK open research sharing platform [13] for sharing robotics systems research. The key enabling technology for bringing together disparate simulations from different projects is Linux containers, a form of virtual machine images. With containerization, we are able to package simulations with all their dependencies, forming an extensible benchmark suite.

In order to achieve our goal of creating a sustainable benchmark suite, we make the following contributions in this paper.

- A novel tool for procedurally creating robotics simula-

tions containers, which are automatically adapted and deployed to the users' host machine.
- A representative set of containerized missions, capturing a variety of software environments commonly used in robotics.
- A case study in using software profiling tools inside containers. The measurement results obtained from containers are validated against results from a real physical robot.
- An overview of observations garnered from profiling the reference set of benchmarks.

As a case study using the RoboBench suite, we analyze the amount of time spent in applications and routines during missions. A surprisingly large amount of CPU cycles go into software for launching and coordinating software submodules. Our data motivate a need for an efficient framework for communication between software modules. Among the remaining compute cycles, we find sensory software (*e.g.*, vision, point clouds), and continuously running routines (*e.g.*, collision detection) occupy a large portion of CPU time. These algorithms may be better supported by specialized hardware, which may include GPUs and field programmable gate arrays (FPGAs).

The rest of this paper is organized as follows: Section II motivates and describes the benefits of using containers; Section III presents a framework for making RoboBench extensible; Section IV describes the initial set of simulations in RoboBench; Section V demonstrates using RoboBench to determine how robot energy is spent in computing; Section VI covers existing benchmark suites in various domains; and Section VII concludes.

II. A Case for Containers for Robots

Where simulations reduce the hardware cost for robotics research, virtualization could further reduce the software maintenance cost of simulators. In order to create a sustainable benchmark of whole robot systems, we distribute packages of all software dependencies for individual simulations, along with the scripted recipe for creating the simulation.

Our approach runs contrary to common robotics software development, where software packages provide general interfaces to support many robot designs and missions, as depicted in the top half of Figure 1. This conventional approach to assembling software is demonstrated in various simulations that are publicly available in robotics repositories such as ROS [18] and robotpkg[1]. But in setting out to create a suite of such simulations, we found it impossible to maintain several system simulations in a environment at once. Different simulations depend on conflicting libraries, and sometimes depend on obsolete libraries—this is a common problem in *all* software, including robotics. These difficulties prevent researchers from replicating results derived from simulations.

Containers are an emerging feature in Linux for creating lightweight virtual machine images capable of solving aforementioned problems. We use Docker [2], the leading application for using Linux containers, to create, package, and distribute robotics simulations, as shown in the bottom half of Figure 1. Simulation containers come with inputs that specify mission goals, a description of the robot, all software libraries to support mission tasks, and a simulated environment.

This approach provides several benefits: *1*) *Guaranteed deployment:* similar to VM images, containers come with installations of libraries and their own Linux environment. *2*) *Ability to host multiple simulations:* each simulation gets a dedicated file system root directory, allowing multiple simulations to reside on a host, even if they require conflicting dependencies. *3*) *Lightweight:* unlike full virtualization, containers do not incur the overhead of a guest kernel or OS; container images have a tractable size that can be hosted on public repositories for containers such as Dockerhub. *4*) *Native hardware access:* with proper configuration, the container can access the GPU for rendering and computation with near native machine performance.

Due to this ability to extend the longevity of research software, Docker and Linux containers have attracted attention for enabling reproducible research in other fields [7]. This approach may also assist sharing of software and experiments at conferences and journals.

In addition to creating the containers for simulations, our work overcomes the following problems in using containers for robotics research: *1*) *User adoption:* adoption of the containerization would require robotics researchers to track yet another complex technology, which would limit the adoption of this approach. *2*) *Metadata and documentation:* while functionally complete, containers still need to be associated with documentation for its construction, purpose, and expected behavior. *3*) *Diversity of host machines:* configuring a container to use host resources (e.g., graphics and sound hardware) requires complex command line options specific to each host machine. These issues were overcome in part by extending the CITK platform through the *citman* application described in the next section.

III. Integration of RoboBench and the CITK Platform

We present *citman*, a tool for automatically generating containers from existing build scripts (also known as *recipes*). The *citman* tool interacts with the CITK[2] project to enable automatic documentation and testing of the generated containers. Using *citman*, researchers can build, run, and publish simulation containers similar to those in RoboBench. As CITK recipes are added by other researchers, they are automatically available through *citman*, and can be profiled with minor alterations. CITK recipes which demonstrate

[1]http://robotpkg.openrobots.org/

[2]CITK is a platform for improving experiment replication and publication for robotics software. CITK does so by providing a content management system for publishing metadata associated with simulations, which are themselves assured to compile from available sources using a continuous integration (CI) framework.

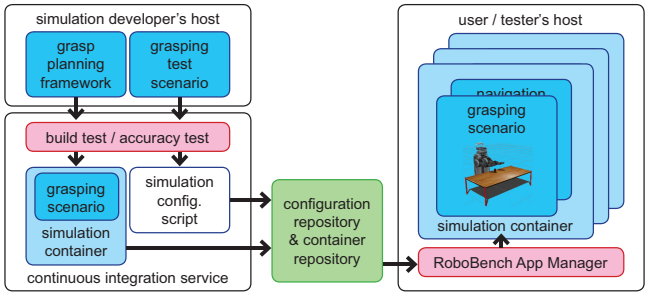

Fig. 2. Our proposed RoboBench infrastructure for system benchmarking. We build upon existing continuous integration services, which check for successful compilation and accurate program outputs, to create containers of simulations. The containerized simulations are then shared via public repositories. Users obtain and configure simulations to run on new machines with the RoboBench Application Manager.

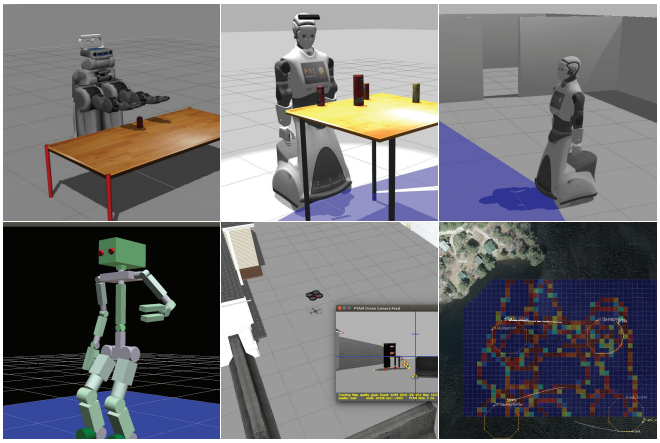

Fig. 3. Six of the robots in RoboBench carrying out tasks in simulation. From left to right, top row: PR2 grasping, REEM grasping, REEM navigation; bottom row: humanoid walking, quadrotor localization, AUV swarm.

interesting workloads may be selected to become part of the benchmark suite.

Developing the recipes for complex full system simulations is challenging. The CITK platform provides a language for building and testing full systems, along with a repository of modules and datasets useful in systems. By extending CITK's modular recipe framework, RoboBench seeks to create an appealing platform for developers to build full system tests for their own development needs.

The CITK repository currently has some full-mission simulations, but it still is missing missions that would include important robotics domains and frameworks. As part of creating RoboBench, we have added recipes that include important frameworks used in a more diverse set of robotics domains. In these domains we have implemented sample missions which demonstrate use of these frameworks and validated that our containerization and benchmarking approach works.

As the name suggests, continuous integration (CI) testing is a software engineering practice where code committed by developers is frequently tested to ensure they produce correct results. CI testing can encompass testing of multiple programs working together as a robotic system. By requirement, a body of code and dependencies that is able to pass CI testing is able to compile and run without human invention, and therefore ready to be captured in a container.

In theory, CI alone (without using containers) should guarantee preservation and delivery of artifacts. The ongoing tests and maintenance by developers should ensure that a running system always be available. In reality, the required libraries beyond the control of the system developer may become obsolete, at which point a working system simulation ceases to be available. Containerization mends this shortcoming of CI by capturing the binaries, libraries, and environment when it is presently working, ensuring that the simulations will run some time in the future.

In particular, *citman* extends the CI capabilities provided by the CITK project, which are in turn built on Jenkins CI[3], a widely used CI framework. We chose to extend this project because of its focus on CI for robotic systems, and for the additional documentation and metadata sharing services it provides.

Our integration of RoboBench's *citman* tool with CITK provides an additional benefit of having CITK's testing capabilities available *inside* containers. CITK provides a way to test system simulations using finite state machine testing (FSMT), which checks the command line output of programs for correct results at specified times. This model of testing is ideal for checking for successful start-up of all the programs and satisfactory termination of the simulation, and for launching analysis tools such as code profilers at specified times during a robotics mission.

By facilitating creation of containers from existing and inherently useful tests, our framework may increase the number of simulations captured in RoboBench, forming a larger benchmark suite. Further details for integrating tests in the RoboBench / CITK platform can be found on the RoboBench portal of the CITK server, found at *http://robobench.net*, described further in Section VII. The reader is encouraged to visit this site for detailed instructions on how to download and install the *citman* application to reproduce the results found in the following sections.

IV. Selection of Benchmark Simulations

Table I shows the seven simulations currently included in the benchmark suite, which include simulations of *1) PR2 grasping:* a robot segmenting a point cloud scene with a bottle, followed by grasping it; *2) PR2 navigation:* a robot finding a path around a map; *3) REEM grasping, 4) REEM navigation:* a REEM robot performing equivalent tasks as the PR2[4]; *5) humanoid walking:* a two legged robot maintaining balance and walking; *6) quadrotor localization:* an aerial reconnaissance drone using vision-based localization, flying to preset waypoints; *7) AUV swarm:* a simulation of multiple autonomous underwater vehicles performing collision

[3]https://jenkins-ci.org/

[4]PR2 and REEM are both grasping robots used in research settings.

TABLE I

SUMMARY OF ROBOTS, SIMULATION FRAMEWORKS, MISSION TASK, AND CAPABILITIES USED FOR TASK

Simulation	Robot	Dependencies	Mission task	Capabilities used
PR2 grasping	PR2 Interactive Manipulation	Ubuntu 12.04, ROS (Groovy), Gazebo	Segment image from Kinect, identify a bottle and grab it	Point cloud clustering, grasp planning
PR2 navigation	PR2 Gazebo	Ubuntu 12.04, ROS (Groovy), Gazebo	By searching a preloaded map, find a way between two points	Search based path planning
REEM grasping	REEM	Ubuntu 12.04, ROS (Hydro), Gazebo	Identify bottle and grab it	Grasp planning
REEM navigation	REEM	Ubuntu 12.04, ROS (Hydro), Gazebo	Navigate through doorway to adjacent room	Search based path planning
Humanoid walking	Model of humanoid walker under real time control	ROS (Hydro), HRPSYS Simulator	Maintain balance, walk in straight line	Forward dynamics modeling, inverse dynamics control
Quadrotor localization	TUM quadrotor AR drone	Ubuntu 12.04, ROS (Hydro), TUM Simulator	Navigate to a set of waypoints	Detect video stream features, Kalman filter sensor fusion
AUV swarm	Swarm of AUVs under control of pHelmIvP autopilot	MOOS IVP simulator	Navigate to a set of waypoints	Collision avoidance by calculating closest point of approach between vessels

avoidance while navigating to waypoints.

While these robots do not form a comprehensive set of robot designs, these robots were selected to form a *representative* set of different operating environments, including aerial, terrestrial, and aquatic robots, which come equipped with wide variety of sensory input, levels of autonomy, and actuator designs. The robots' behaviors are supported by a set of middleware frameworks including ROS, and planning frameworks including MoveIt![5]. These robots receive sensory input from simulators such as Gazebo [12], HRPSys[6], and MOOS-IVP [5], and their activity can be viewed through visualization user interfaces such as Rviz[7].

V. CASE STUDY: ROBOBENCH ENERGY CHARACTERIZATION

As mobile robots become more autonomous and take on longer mission durations, the energy efficiency of their subsystems becomes a greater concern. An often overlooked component of energy consumption is that of computation, which we forecast will take up a greater fraction of the total energy budget in future robots. Significant research effort has been devoted to improving robot capabilities, in terms of features and accuracy, along with the efficiency of individual algorithms. But despite the immense gains in individual subsystems, it is not clear how these optimizations form an energy-efficient computer architecture for mobile robotics.

In this section we provide motivation for creating a system-wide benchmark suite, starting by making a case that robot computation efficiency is increasingly important. Using the amount of CPU time a software spends as an approximation of its energy consumption, we reveal what pieces of software incur the most energy cost over the course of a mission. This helps researchers evaluate the overall impact of individual algorithm optimizations, taking into account what programs ultimately take the most time. This kind of study of the computational demands of the collection of robotics software is only made possible by the infrastructure provided by RoboBench.

[5]http://moveit.ros.org/support/

[6]http://wiki.ros.org/hrpsys

[7]http://wiki.ros.org/rviz

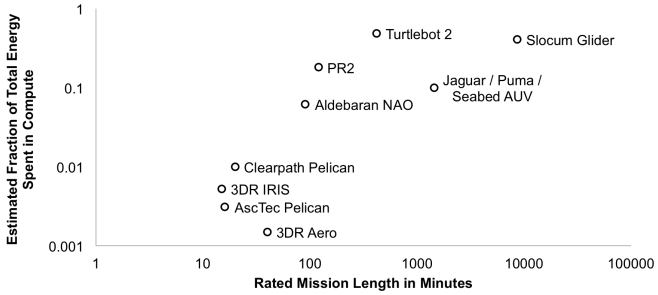

Fig. 4. Fraction of robot battery energy spent on computation.

A. Brain vs. Brawn

Robots, as mobile computers with wheels and limbs, have the unique energy demands of having to power motors and sensors. Lighter materials, efficient actuators, and capacious batteries all drive improvements in robot mission endurance. But as robots are required to operate with increasing levels of autonomy while subject to more varied environments, mobile robots are acquiring more advanced sensors (e.g., stereo cameras and laser scanners), and actuators (hands), both of which present greater computational demands.

Figure 4 shows an estimate of the percentage of battery energy spent on computation for nine robots, using data obtained from public specification sheets. The compute energy is estimated via the total dissipated power (TDP) of the CPU, which overestimates the processor load but excludes energy consumed by memory and cooling. We find later in this section that sensor processing and safety monitoring run constantly, even when the motors are idle, such that the rated TDP is a reasonable approximation for compute energy. The energy expenditure outside of compute (sensors and motors) is obtained from specification sheets, or is derived by dividing total energy stores by the rated endurance and the subtracting the TDP, yielding the most conservative estimate. The values are plotted against endurance as a measure of

mobility and autonomy. Logarithmic axes are used to fit the immense range of robot designs on the same chart.

We see that autonomous underwater vehicles (AUVs) spend 10-40% energy on computation, due to their ability to loiter for long periods while neutrally buoyant; when they do move, AUVs use hydrofoils and control surfaces to glide through the water while moderating their buoyancy, achieving highly efficient movement. Modest improvements in computational efficiency could significantly enhance AUV endurance. This is in contrast to the rotorcraft fliers, which spend all their energy just to stay aloft. Therefore, compute is not the key consideration when optimizing for quadrotor endurance.

The most computationally capable robots, the humanoid Nao, mobile manipulator PR2, and the Turtlebot 2, with its netbook payload, spend 8-12% energy on computation. These terrestrials make use of compute power for seeing, planning, and/or grasping, and increasingly for interacting with humans through facial, speech recognition and voice synthesis; all of these tasks place increasing computational load on robot systems.

Robots that operate tethered to an external power source, such as Baxter or Atlas, or mobile robots that derive energy from fuel, such as the BigDog or Cheetah robots, are excluded from this study.

B. Robot Software Profiling and Instrumentation

The relative importance of programs and functions in a system's workload can be measured via sampling-based profiling, which entails occasionally recording the processor stack while a workload is in progress. We use OProfile[8] to measure what percentage time is spent in applications and libraries. Where possible, we either obtained debug symbol libraries or compiled binaries with profiling symbols to expose function names and parameter signatures, allowing us to obtain a highly detailed, function-level, breakdown of where CPU cycles are spent. This allows analysis of the amount of CPU load presented by all software, including sensory processing, planning, control, and any middleware frameworks such as ROS.

Sampling based software profiling is not well supported inside virtual environments. In our experience OProfile most reliably identifies the source code for function call samples. The competing Linux perf tool supports sampling within KVM based virtual machines, but not in container-based machines such as Docker containers. When using perf within containers, perf delays resolving source code information to after sampling is complete, looking in file system locations which may have been removed by Docker's union file system; therefore, perf does not reliably generate source-level information.

We use GNU Gprof and Google perftools[9] to further analyze programs taking most time, in particular to plot their internal function call dependencies.

[8]http://oprofile.sourceforge.net/about/

[9]https://github.com/gperftools/gperftools

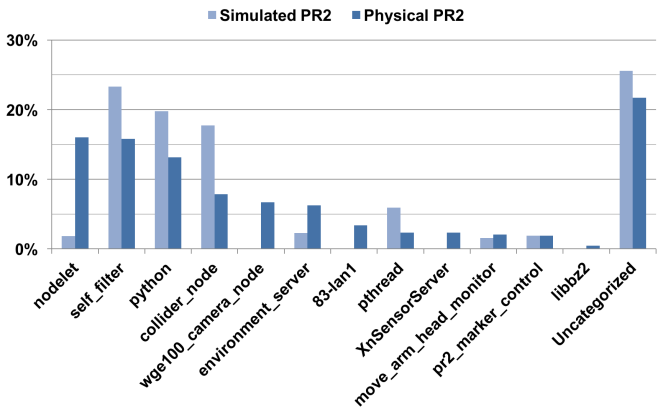

Fig. 5. Distribution of time spent in programs and routines in a PR2 during a grasping task.

For each simulation we identified the processes that only exist because the robot is running in simulation; these include any samples occurring in setting up the virtual world, such as Gazebo's core linear complementarity solvers for physics simulation, UI software such as Rviz, graphics drivers, and our profiling tools. These are removed from the analysis.

Due to space constraints, we present analysis of three example robots; the other simulations are available as part of the RoboBench suite.

C. Analysis of PR2 Grasping Simulation

We analyze the system profile of the PR2 robot during a mission where the robot segments a point cloud, identifies a bottle, and plans and carries out a grasp. We compare the system profile of this simulation to that of a physical robot doing the same task. Figure 5 shows the relative amount of CPU cycles spent by programs in PR2, showing the top categories that account for 75% of the samples during the mission.

In some aspects the simulated robot differs from the physical measurement: activity from `nodelet` appears only in the physical robot because the simulated robot does not compress and decompress sensor data streams; likewise, this simulated robot lacks activity in driving camera and network stacks. These discrepancies are caused by the simulator's limited ability to model distributed computing resources in the robot. Outside of these differences, the two activity traces follow the same trends.

This measurement provides a number of interesting findings: *1) A large amount of computing energy is spent in infrastructure code:* Python routines for setting up main programs and logging are one of the largest consumers of CPU time. Also prominent are networking routines and pthread waits. The systems we profiled all rely on middleware frameworks, such as ROS, in order to support a plug-and-play model consisting of many different modules. Often these frameworks themselves require significant computational power. *2) Collision detection, sensory input run constantly:* outside of frameworks, the largest consumers of CPU time that do computational work are `nodelet` binary

TABLE II

FUNCTION CALLS USING THE MOST CYCLES IN SIMULATED PR2, EXCLUDING CYCLES SPENT IN CALLEE FUNCTIONS

Program	Source code file name	Function signature	Function description	% mission CPU cycles
self_filter_color	bodies.cpp	ConvexMesh::intersectsRay()	raytracing a point in point cloud to see if it is from robot itself	11.2%
collider_node	collider.cpp	Collider::degradeSingleSpeckles()	denoising octree entries, remove errant data	3.4%
collider_node	collider.h	Collider::getOccupiedPoints()	octree lookup	3.2%
collider_node	OcTreeDataNode.hxx	OcTreeDataNode::hasChildren()	octree traversal	1.4%

for compressing point clouds, `self_filter` for clustering points in the point cloud and identifying the robot's arms, and `collider_node` for processing laser scanner input and collision detection. These are routines that are called from several sources and run constantly, regardless of the robot activity. *3*) *Complex robotics algorithms may represent relatively little computing energy:* As a corollary, various algorithms that are of high research interest comprise a small portion of the runtime; these include vision routines that run occasionally, such as image segmentation, and motion planning to find a grasp. *4*) *The most expensive functions include three dimensional loops:* the function calls that show up most frequently during the simulation all have nested loop structures accessing a three dimensional data structure, Table II shows the signatures of these function calls. *5*) *The workload is highly multiprogrammed:* numerous programs are needed to support the task, with over 300 processes showing activity at some point in the mission.

D. Analysis of Humanoid Walking Simulation

The humanoid robot walking simulation has a robot in an environment generated in an OpenHRP simulator. The robot is balanced with an OpenRTM real time controller. 70% of the robot compute cost occurs in calls to `libhrpModel`, which is the library that provides forward and inverse dynamics routines. Within this library, it appears the majority of the time, 40% of the cycles, is spent in recursive calls to `calcInverseDynamics`. These functions are in turn backed up by the Eigen linear algebra library. This confirms intuition that inverse dynamics is computationally more taxing than forward dynamics, along with forward and inverse kinematics combined.

E. Analysis of Quadrotor Localization Simulation

We simulate a quadrotor flying in a street scene, navigating to preset waypoints. The quadrotor must use input from a camera, gyroscope, and accelerometer to localize itself. The system profile shows that the binary for SLAM, `drone_stateestimation`, takes 27% of the CPU time, the remaining time is spent in the binaries `state_publisher`, `drone_autopilot`, Python, `message_to_tf`, and `rosout` each taking 13%-15% of cycles each. Further analysis of the function calls in `drone_stateestimation` shows sensor fusion is the bulk of computation.

Case Study Summary: The applications, functions, and algorithms identified in this type of study may be ideal candidates for optimization. More interesting, however, is the fact no algorithms overwhelmingly occupy this set of robots' computers as they complete their missions. The largest improvements in computing efficiency may come from changes to robot software infrastructure, along with possible computer architecture and hardware support for select algorithms.

TABLE III

SUMMARY OF ROBOTICS BENCHMARKS

	Problem domain	Benchmark
Sensor	vision	SD-VBS [20]
	mobile system vision	MEVBench [9]
	SLAM	SLAMBench [16]
	visual servoing	ViSP [14]
Autonomy	physics simulation	Bullet benchmark [1]
	2d motion planning	MoVeMA [8]
Actuator	arm motion planning	Planner Arena [15]
	grasp planning	OpenGRASP [19]

TABLE IV

SUMMARY OF ARCHITECTURE BENCHMARKS

Hardware system class	Benchmark
Microcontroller embedded systems	EEMBC [17]
Single threaded CPU	SPEC CPU2006 [11]
Multithreaded CPU	PARSEC [6]
Cloud and datacenter workloads	CloudSuite [10]

VI. RELATED WORK

Benchmarking is has been central to the systematic development of several computer science fields. Within robotics much research has been done in creating benchmarks organized around specific problem domains. Table III shows a number of such benchmarks. These benchmark suites help answer questions such as *"how accurately or how efficiently does an application solve a problem?"* As an additional benefit, the benchmark suites shown in Table III concisely share with non-experts emerging problems and state-of-the-art applications througout robotics.

RoboBench presents a class of benchmarks different from any shown in Table III. The benchmark simulations that can be created by the RoboBench framework capture the software behavior of an entire software system, and help answer questions such as *"what applications and algorithms are running in a robot?"*. RoboBench benchmarks concisely share with non-experts in robotics implementations of robotic software systems, similar to how benchmarks in Table IV capture the software behavior of various classes of computers.

VII. Conclusion & Future Work

In many computer science fields, reproducible benchmarking is a cornerstone of evaluating designs and the peer review process. The robotics community has increasingly recognized the need for benchmarks in systematic research. However, the lack of a platform for distributing whole systems has limited benchmarks to evaluating isolated components. In this paper, we have presented a framework for creating system benchmarks, which are important for a system-wide view of performance and efficiency. This view is a first step for answering important questions about robot designs, such as the computational needs of a system designed to perform a particular mission.

Some projects in the recent years, such as ROS and robotpkg, have addressed the distribution problem of software system engineering, through package repositories; however, this has not provided adequate stability to keep large software projects available without significant maintenance.

Emerging technologies (e.g., containerization, CI) overcome these challenges by assuring the availability of running systems, long after they were developed. In this paper we presented how to leverage these technologies to create a sustainable, extensible robotics benchmark suite.

As a case study of how system-wide benchmarking is useful, we measured the relative cost of different software components of a few mobile robots carrying out missions. Insights such as these are crucial for guiding further research and development, such as designing efficient custom hardware for robotics. The availability of such a suite of benchmarks enables researchers from outside the robotics community to analyze the benchmarks and take part in related research activities.

Our measurements from the simulations, though secondary to the main contribution of the paper, are in themselves surprising. Background tasks that persist the whole life of the mission, such as sensor processing for obstacle avoidance and intermodule communication overhead, may consume more CPU cycles than short-lived but complex tasks such as motion planning, therefore limiting the impact of software optimizations targeted at individual modules on overall system efficiency.

The instructions for replicating the experiments presented in this paper, the descriptions for all the simulated systems and their subcomponents, and the data that comprises the graphs shown in this paper are available at `www.robobench.net`.

With this paper we introduce just one possible use case for a benchmark simulation suite. At the webpage above we provide instructions on how to install and use *citman*, and on how to submit new simulations for inclusion. We hope this infrastructure will promote system-level benchmarking, while maintaining the transparency necessary to enable reproducible scientific research.

References

[1] "Bullet physics library," http://bulletphysics.org/wordpress/.

[2] "Docker: An open platform for distributed applications for developers and sysadmins." https://www.docker.com/, accessed: 2015-03-02.

[3] A. Ahmad, I. Awaad, F. Amigoni, J. Berghofer, R. Bischoff, A. Bonarini, R. Dwiputra, G. Fontana, F. Hegger, N. Hochgeschwender, *et al.*, "Specification of general features of scenarios and robots for benchmarking through competitions," *RoCKIn Deliverable D*, vol. 1, 2013.

[4] J. Anderson, J. Baltes, and C. T. Cheng, "Robotics competitions as benchmarks for AI research," *The Knowledge Engineering Review*, vol. 26, no. 01, pp. 11–17, 2011.

[5] M. R. Benjamin, H. Schmidt, P. M. Newman, and J. J. Leonard, "Nested Autonomy for Unmanned Marine Vehicles with MOOS-IvP," *Journal of Field Robotics*, vol. 27, no. 6, pp. 834–875, November/December 2010.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[7] C. Boettiger, "An introduction to Docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015.

[8] D. Calisi, L. Iocchi, and D. Nardi, "A unified benchmark framework for autonomous mobile robots and vehicles motion algorithms (MoVeMA benchmarks)," in *Workshop on experimental methodology and benchmarking in robotics research (RSS 2008)*, 2008.

[9] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "MEVBench: A mobile computer vision benchmarking suite," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, Nov 2011, pp. 91–102.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, Mar. 2012.

[11] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sept. 2006.

[12] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, Sep 2004, pp. 2149–2154.

[13] F. Lier, J. Wienke, A. Nordmann, S. Wachsmuth, and S. Wrede, "The cognitive interaction toolkit improving reproducibility of robotic systems experiments," in *Simulation, Modeling, and Programming for Autonomous Robots*, D. Brugali, J. Broenink, T. Kroeger, and B. MacDonald, Eds. Springer International Publishing, 2014, vol. 8810, pp. 400–411.

[14] E. Marchand, F. Spindler, and F. Chaumette, "ViSP for visual servoing: a generic software platform with a wide class of robot control skills," *Robotics Automation Magazine, IEEE*, vol. 12, no. 4, pp. 40–52, Dec 2005.

[15] M. Moll, I. A. Sucan, and L. E. Kavraki, "An extensible benchmarking infrastructure for motion planning algorithms," *CoRR*, vol. abs/1412.6673, 2014. [Online]. Available: http://arxiv.org/abs/1412.6673

[16] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. D. Riley, N. Topham, and S. Furber, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," *CoRR*, vol. abs/1410.2167, 2014. [Online]. Available: http://arxiv.org/abs/1410.2167

[17] J. Poovey, T. Conte, M. Levy, and S. Gal-On, "A benchmark characterization of the EEMBC benchmark suite," *Micro, IEEE*, vol. 29, no. 5, pp. 18–29, Sept 2009.

[18] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[19] S. Ulbrich, D. Kappler, T. Asfour, N. Vahrenkamp, A. Bierbaum, M. Przybylski, and R. Dillmann, "The OpenGRASP benchmarking suite: An environment for the comparative analysis of grasping and dexterous manipulation," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 1761–1767.

[20] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The san diego vision benchmark suite," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 55–64.