2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Macau, China, November 4-8, 2019

Benchmarking and Workload Analysis of Robot Dynamics Algorithms

Sabrina M. Neuman, Twan Koolen, Jules Drean, Jason E. Miller, and Srinivas Devadas

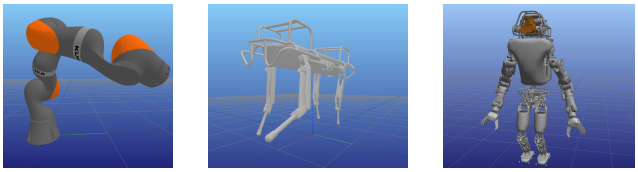
***Abstract*— Rigid body dynamics calculations are needed for many tasks in robotics, including online control. While there currently exist several competing software implementations that are sufficient for use in traditional control approaches, emerging sophisticated motion control techniques such as nonlinear model predictive control demand orders of magnitude more frequent dynamics calculations. Current software solutions are not fast enough to meet that demand for complex robots. The goal of this work is to examine the performance of current dynamics software libraries in detail. In this paper, we (i) survey current state-of-the-art software implementations of the key rigid body dynamics algorithms (RBDL, Pinocchio, RigidBodyDynamics.jl, and RobCoGen), (ii) establish a methodology for benchmarking these algorithms, and (iii) characterize their performance through real measurements taken on a modern hardware platform. With this analysis, we aim to provide direction for future improvements that will need to be made to enable emerging techniques for real-time robot motion control. To this end, we are also releasing our suite of benchmarks to enable others to help contribute to this important task.**

I. Introduction

Modern robotics relies heavily on rigid body dynamics software for tasks such as simulation, online control, trajectory optimization, and system identification. But while early robots performed simple, repetitive tasks in constrained environments, robots are increasingly expected to perform complex operations in dynamic, unstructured, and unpredictable environments, ranging from non-standard manipulation tasks [1] to disaster response [2]. These new challenges will require robots to adapt to their environments in real-time, which in turn will require more complex control algorithms that place a greater burden on the rigid body dynamics implementations that drive them.

One trend in the effort to improve the adaptability of robots to their environment is the increased use of nonlinear model-based control (MPC) [3], which moves trajectory optimization, traditionally an off-line task, into the realm of online control. Differential dynamic programming (DDP) techniques, including the iterative linear quadratic regulator (iLQR) algorithm, are being deployed on more complex robots [4], [5]. These techniques require simulating into the future at every control time step, and also require gradients of the dynamics. Where traditional control approaches might only evaluate dynamical quantities once per time step, MPC requires many evaluations, with the quality of the control policy improving with longer time horizons. Another trend is the use of machine learning algorithms trained on data obtained from simulation. The performance of these algorithms typically improves with larger quantities of input data [6]. Thus, speeding up the production of simulation samples may enable improved performance and lower training costs.

S. M. Neuman, T. Koolen, J. Drean, J. E. Miller, and S. Devadas are with the Computer Science and Artificial Intelligence Laboratory at MIT, Cambridge, MA, USA. {sneuman, tkoolen, drean, jasonm, devadas}@mit.edu



(a) LBR iiwa (b) HyQ (c) Atlas

Fig. 1. 3D visualizations of the robot models used as benchmark cases. From left to right, LBR iiwa (KUKA AG), HyQ (DLS Lab at the Italian Institute of Technology), and Atlas (Boston Dynamics, version used during the DARPA Robotics Challenge).

Much work has gone into creating high-performance rigid body dynamics libraries [7]–[12]. However, existing implementations still do not have the performance necessary to satisfactorily run algorithms like iLQR for complex robots on off-the-shelf hardware [13]. To help improve this situation, we present benchmark results and an associated benchmark suite for rigid body dynamics libraries. We include representative examples of three different robot categories (see Fig. 1): a manipulator (LBR iiwa), a quadruped (HyQ), and a humanoid (Atlas). These benchmarks are aimed at helping control engineers and library authors understand differences and similarities between the libraries and identifying possible areas for optimization. We perform a workload analysis in this paper, and are also releasing our suite for use by the larger community.

This paper makes several important contributions:

1) An open-source benchmark suite (`http://github.com/rbd-benchmarks/rbd-benchmarks`);
2) Direct comparison of four state-of-the-art rigid body dynamics libraries;
3) Consistent comparison of each of the libraries, using the same inputs and ensuring that their outputs match;
4) Low-level performance statistics collected on modern hardware using hardware performance counters;
5) An analysis of trends and differences across the distinct combinations of algorithm and implementation, to provide insight for future work in optimization and acceleration of this workload.

To our knowledge, this paper provides the most comprehensive analysis of the rigid body dynamics workload to date. Previous work analyzing various software implementations focused only on overall performance [7]–[11], but not a full

978-1-7281-4004-9/19/$31.00 ©2019 IEEE

workload analysis from a microarchitectural perspective. To our knowledge, the only evaluation to analyze additional performance measurements was in [9], where the authors used the profiling tool Valgrind to report instruction counts, cache misses, and branch misprediction rates on several hardware platforms. However, this study was much more limited in scope – only two software libraries and two dynamics algorithms were measured. By contrast, our suite includes three key dynamics algorithms implemented by four software libraries, and we present measurements taken from hardware performance counters, including instruction counts, cache misses, stall cycles, floating-point vector operations, and instruction mix.

We begin with a brief review of the rigid body dynamics problems under study. We will then survey current, state-of-the-art software implementations of these algorithms and characterize their performance and use of resources on modern hardware. Finally, we will examine the commonalities and differences between these implementations to motivate future work on the acceleration of these algorithms, an important step towards satisfying the future computational needs of real-time robot motion control.

II. Dynamics Algorithms

Subject to the standard rigid body assumptions, the dynamics of a general robot without kinematic loops can be described using the well-known equations of motion,

$$M(q)\dot{v} + c(q, v) = \tau, \tag{1}$$

where:

- $q \in \mathbb{R}^n$ is the joint configuration (or position) vector;
- $v \in \mathbb{R}^m$ is the joint velocity vector;
- $\tau \in \mathbb{R}^m$ is the joint torque (or effort/force) vector, which may include motor torques determined by a controller as well as friction torques and other external disturbances;
- $M(q) \in \mathbb{R}^{m \times m}$ is the positive definite (joint space) mass or inertia matrix;
- $c(q, v) \in \mathbb{R}^m$ includes velocity-dependent terms and terms due to gravity, and is referred to as the generalized bias force [14] or nonlinear effects.

In this paper, we will focus on three standard problems related to the equations of motion:

1) *forward dynamics*: computing $\dot{v}$ given q, v, and τ;
2) *inverse dynamics*: computing τ given q, v, and $\dot{v}$;
3) *mass matrix*: computing $M(q)$.

Algorithms that solve these standard problems are based on traversing the kinematic tree of the robot in several passes, either outward from the root (fixed world) to the leaves (*e.g.,* the extremities of a humanoid robot) or inward from the leaves to the root. While the presented algorithms are often described as being recursive, in practice the kinematic tree is topologically sorted, so that recursions are transformed into simple loops, and data associated with joints and bodies can be laid out flat in memory. Data and intermediate computation results related to each of the joints and bodies are represented using small fixed-size vectors and matrices (up to 6×6).

The dominant inverse dynamics algorithm is the Recursive Newton-Euler Algorithm (RNEA) [15]. It has $O(N)$ time complexity, where N is the number of bodies. The Composite Rigid Body Algorithm (CRBA) [16] is the dominant algorithm for computing the mass matrix. It has $O(Nd)$ time complexity, where d is the depth of the kinematic tree. For forward dynamics, the $O(N)$ articulated body algorithm (ABA) is often used [17]. Alternatively, one of the libraries under study computes $M(q)$ using the CRBA and $c(q, v)$ using the RNEA, and then solves the remaining linear equation for $\dot{v}$ using a Cholesky decomposition of $M(q)$. Though this approach is $O(N^3)$, shared intermediate computation results between the RNEA and CRBA and the use of a highly optimized Cholesky decomposition algorithm could make this approach competitive for small N. Each of the algorithms is described in detail in [14]. To bound the scope of this work, we focus on dynamics without contact.

III. Survey of Software Libraries

We briefly introduce the software libraries under study and highlight some of their commonalities and key differences. All of the libraries have been developed fairly recently, and are actively maintained, open source, and provided free of charge. The libraries can all load the layout and properties of a robot from the common ROS URDF [18] file format, or a file format to which URDF can be converted using automated tools. A well-known library missing from this study is SD-FAST [12]. We chose not to include this library because it is somewhat older, proprietary, and the authors of several libraries included in the study have published benchmarks that show significant improvements over SD-FAST. The libraries are summarized in Table I.

In contrast to the other libraries, RobCoGen is a Java tool that generates C++ source code specific to a given robot. RigidBodyDynamics.jl (referred to from this point on as RBD.jl) is unique in that it is implemented in Julia [19]. It is also the only library that doesn't use the ABA for forward dynamics, instead solving for the joint accelerations using a Cholesky decomposition of the mass matrix. Furthermore, RBD.jl annotates the typical spatial vector and matrix types (*e.g.,* wrenches) with an additional integer describing the coordinate frame in which the quantity is expressed, for user convenience and to enable frame mismatch checks (disabled for our benchmarks).

All of the C++ libraries use Eigen [20] as a linear algebra backend. Eigen provides fixed-size matrix and vector types that can be allocated on the stack, thereby avoiding any overhead due to dynamic memory allocation. Operations involving these types can use vector (SIMD) instructions, if available. RBD.jl, uses StaticArrays.jl for similar fixed-size array functionality, in addition to using OpenBLAS through Julia's LinearAlgebra standard library for operations on dynamically-sized matrices (mainly, the Cholesky decomposition of the mass matrix).

A key difference between the libraries lies in how different joint types (*e.g.,* revolute or floating) are handled. While a naive implementation might rely on inheritance and virtual

TABLE I
DYNAMICS LIBRARIES EVALUATED

Library	Language	Linear Algebra Backends	Coordinate Choice	Release Date
RBDL 2.6.0 [10]	C++	Eigen 3.3.7	Body Coordinates	02 May 2018
Pinocchio 2.0.0 [8]	C++	Eigen 3.3.7	Body Coordinates	10 Jan 2019
RigidBodyDynamics.jl 1.4.0 [7]	Julia	StaticArrays.jl 0.10.2, OpenBLAS 0.3.3	World Coordinates	02 Feb 2019
RobCoGen 0.5.1 [11]	C++	Eigen 3.3.7	Body Coordinates	07 Dec 2018

methods, all of the libraries have avoided this in various ways to improve performance. RBDL enumerates the different joint types, and uses run-time branching based on the joint type in the main algorithm loops to implement joint-specific functionality. RBD.jl's non-standard choice of implementing the algorithms in world coordinates allows joint-specific computations to be performed out-of-order: data for all joints of the same type are stored in separate vectors and are iterated over separately, avoiding if-statements in tight loops. Pinocchio handles different joint types using a visitor pattern based on the Boost C++ library. RobCoGen's generated code unrolls all of the loops in the dynamics algorithms, replacing them with explicit repetition of their contents and thus avoiding the overhead of program control flow (*e.g.,* branches and address calculations) in general, including in the implementation of joint-type-specific behavior.

A peculiarity of RobCoGen is that the authors have opted to implement a hybrid dynamics algorithm for floating-base robots, instead of regular RNEA. In essence, this hybrid dynamics algorithm does forward dynamics for the floating base joint, while implementing standard RNEA for the non-floating (revolute) joints. Although the inputs and outputs of this algorithm are not the same as for the 'regular' inverse dynamics algorithms implemented by the other libraries, we still chose to include the results, as we deemed the algorithm to still be similar enough.

Both Pinocchio and RBD.jl implement the algorithms in a generic (parameterized or templated) way. This enables, for example, automatic differentiation of the dynamics using non-standard special scalar types and function overloading.

IV. METHODOLOGY

This section details our method for collecting timing results and measurements from performance counters on a hardware platform.

A. Hardware Measurement Setup

Evaluation of the software implementations was performed on a modern desktop machine with a quad-core processor. Key hardware parameters are shown in Table II. This machine was selected for the workload measurements because quad-core Intel i7 processors are a common choice for state-of-the-art robots, including many featured in the 2015 DARPA Robotics Challenge [2], such as Atlas [21], Valkyrie [22], and CHIMP [23].

To isolate our measurements from the non-deterministic effects of changing clock frequencies and thread migrations, TurboBoost and HyperThreading were disabled in the BIOS.

TABLE II
HARDWARE SYSTEM CONFIGURATION

Feature	Value
Processor / Frequency	Intel i7-7700, 4 Cores / 3.6GHz
Private L1 / L2 Cache per Core	8-way, 32kB / 4-way, 256kB
L3 Cache / DRAM Channels	16-way, 2MB / 2 Channels

The clock frequency of all four cores was fixed at 3.6GHz. To measure timing, we used the Linux system call `clock_gettime()`, with `CLOCK_MONOTONIC` as the source.

To collect measurements of architectural statistics, we used the hardware performance counter registers [24] provided by the processor, which can be configured to monitor a given set of hardware events (*e.g.,* cache misses, instructions retired). We accessed these registers using LIKWID [25], a C library interface for reading hardware performance counters. In our testbed code, calls to the dynamics libraries' routines were instrumented with LIKWID routines, in order to carefully measure only activity during the regions of interest.

B. Software Environment

Our hardware measurement platform ran Ubuntu 16.04. For RBD.jl, we used version 1.1.0 of Julia [19] with flags `-O3` and `--check-bounds=no`. All C/C++ code was compiled using Clang 6.0, which we chose because both Clang 6.0 and Julia 1.1.0 use LLVM 6 as their backend. The RBDL, Pinocchio, and RobCoGen C++ libraries were compiled with the "Release" CMake build type option. For RobCoGen, we had to add the `EIGEN_DONT_ALIGN_STATICALLY` flag to the default flags in the generated CMake file, to avoid crashes due to memory alignment issues.

While Julia is JIT-compiled by default, for this study we statically compiled the relevant RBD.jl functions to a C-compatible dynamic library ahead of time using the PackageCompiler.jl Julia package [26], so as to avoid benchmark artifacts due to JIT compilation and to enable interoperation with measurement tools. As a further precaution against observing JIT overhead, we called all RBD.jl routines once before starting measurements.

C. Inputs and Cross-Library Verification of Results

To compare the dynamics libraries, we used three different robot models as inputs: iiwa, a robot arm with 7 revolute joints [27], HyQ, a quadruped robot with 12 revolute joints and a floating joint [28], and Atlas, a humanoid robot with 30 revolute joints and a floating joint [29] (fingers not modeled).

For verification, it was important to ensure that all libraries were manipulating the exact same representations of the robot models. This was a non-trivial task because RBDL, Pinocchio, and RBD.jl all take as input a URDF file describing the robot model, whereas RobCoGen uses a custom file format, KinDSL. We started with a set of reference URDF files for each robot. To generate the KinDSL files, we used a file conversion tool created by the authors of RobCoGen, URDF2KinDSL [30]. This tool performs not only a change of syntax in the robot representation, but also a number of less-trivial transformations, *e.g.,* from extrinsic to intrinsic rotations. Unfortunately, URDF2KinDSL did not produce a KinDSL file that exactly matched the URDF for Atlas, so we opted to convert each of the KinDSL files back to URDF using RobCoGen's built-in conversion tool. It was these back-and-forth-converted URDF files that were ultimately used as the inputs for RBDL, Pinocchio, and RBD.jl. While there was still a somewhat larger mismatch between RobCoGen's outputs and those of the URDF-capable libraries, these could be attributed to the rotation transformations.

To simulate the effect of running many dynamics calculations during the operation of a robot with time-varying input data, we executed each dynamics calculation 100,000 times in a row, using different inputs each time. The libraries each expect their inputs in a different format, for example as a result of depth-first or breadth-first topological ordering of the kinematic tree. To enable cross-library verification, we used RBD.jl to generate 100,000 random inputs for each of the robot models (in RBD.jl's format), after which the inputs were transformed into the format that each of the libraries expects. Similarly, corresponding expected output values were computed using RBD.jl, and transformed to the representation expected from each of the other libraries. These expected results were used to verify that all libraries indeed performed the same computation. For the inverse and forward dynamics algorithms, gravitational acceleration and external forces (an optional input) were set to zero to facilitate direct comparison of the libraries.

For each combination of algorithm and robot model, statistics were measured for the entire set of inputs and then these numbers were divided by 100,000 to calculate the average per calculation. To further insulate our measurements from potential background noise, each experiment run of 100,000 inputs was performed 10 non-consecutive times, and the results across those 10 experiments were averaged.

V. Evaluation

This section presents the results of running the four different software implementations of the robot dynamics algorithms for the three different robot models (Section IV). The average execution times of the dynamics algorithms are shown in Fig. 2. Each cluster of bars shows the results for all of the implementations on a particular robot model.

The runtime values shown are averaged across multiple experimental runs of $100,000$ inputs each (as described in Section IV-C). We examined the standard deviation, σ, of the execution time of a single dynamics calculation from a single input value, and we found that for all of our data collected, $0.2\% < \sigma < 1.3\%$ of the overall mean runtimes. This suggests that for these implementations of the dynamics algorithms, the performance is not sensitive to differences in the input data, as we expected.

The top performing libraries overall were RobCoGen and Pinocchio. These results will be explored in more detail in Sections V-A, V-B, and V-C, which examine the data from each algorithm separately. We will also comment on the memory usage of the algorithms, and the effects of various types of parallelism on their performance.

A. Forward Dynamics Results

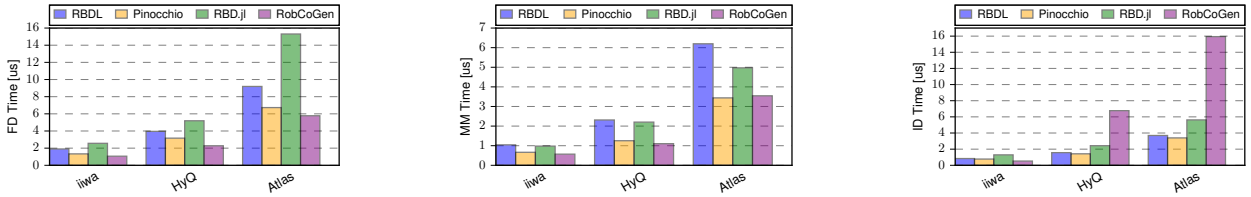
RobCoGen gave the fastest runtimes for forward dynamics for all robot models, at 1.1μs, 2.4μs, and 5.9μs for iiwa, HyQ, and Atlas, respectively (Fig. 2a). The key to RobCoGen's performance on this algorithm is indicated by its instruction count data (Fig. 4a). The total number of instructions retired by RobCoGen is much lower than the total instructions retired by all of the other implementations. This remarkable reduction can be attributed to RobCoGen's technique of using explicit loop unrolling. The unrolled loops can perform the same calculation with far lower instruction count overhead because they eliminate branches (Fig. 4a) and calculation of branch conditions, and can reuse address calculation temporaries. For the limited number of links in the robot models evaluated, loop unrolling is an effective strategy for performance (though it should be noted this may not necessarily be the case for robots with many more links). RobCoGen gives the fastest performance because it does so few instructions overall, despite having a somewhat lower rate of instruction throughput, measured in instructions per cycle (IPC) (Fig. 3). (Note that the maximum rate of IPC per core for our testbed hardware platform is 4 [31].)

The other three libraries, which do not perform explicit loop unrolling, all have similar rates of instruction throughput (Fig. 3), so it is not surprising that their relative runtimes directly correspond to the total instructions retired by each. Recall that RBD.jl, which generates the most instructions (Fig. 4a), also uses a different algorithm for forward dynamics than the other libraries (see Section II), which requires significantly more memory accesses (loads and stores).

These results indicate that reducing the overall amount of instructions to be performed and avoiding extra work (*e.g.,* branch calculations) that does not directly contribute to the algorithm is a clear path to performance success for this algorithm. For RobCoGen, this reduction came from aggressive loop unrolling, leading to a significant reduction in the number of instructions.

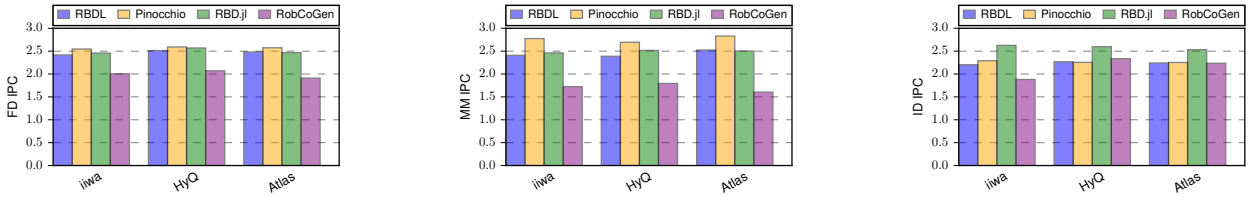
B. Mass Matrix Results

For the mass matrix calculation, RobCoGen was fastest for iiwa and HyQ (at 0.6μs and 1.1μs, respectively), but Pinocchio was slightly faster for the Atlas robot, at 3.5μs (Fig. 2b). RobCoGen's impressive performance on this algorithm has the same explanation as its superior performance on the forward dynamics algorithm (Section V-A): RobCoGen's



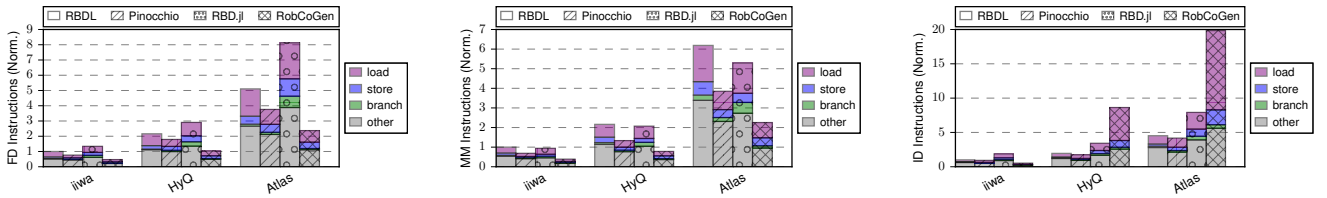
(a) Forward Dynamics Runtime (b) Mass Matrix Runtime (c) Inverse Dynamics Runtime

Fig. 2. Runtime in microseconds. Shorter runtimes indicate better performance. Within each cluster, the bars are in the same order as the top legend.



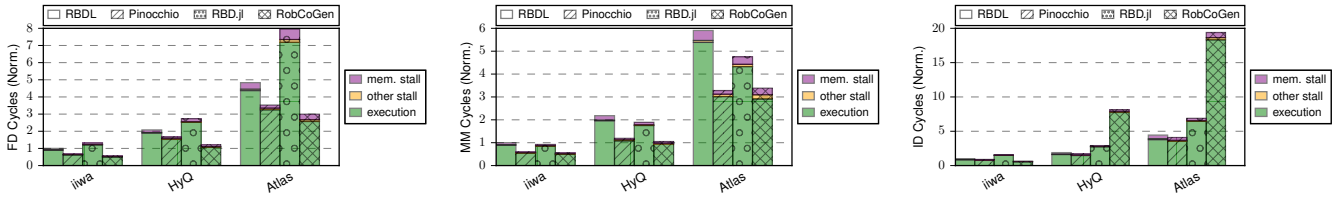
(a) Forward Dynamics Instructions Per Cycle (b) Mass Matrix Instructions Per Cycle (c) Inverse Dynamics Instructions Per Cycle

Fig. 3. Instructions per cycle (IPC). Higher IPC indicates better instruction throughput.



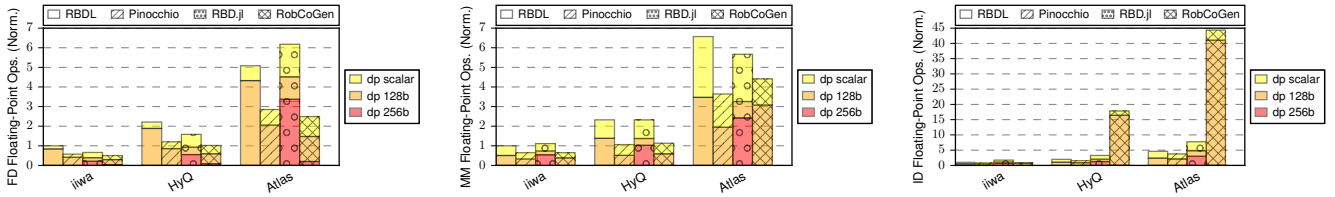
(a) Forward Dynamics Instructions (b) Mass Matrix Instructions (c) Inverse Dynamics Instructions

Fig. 4. Total instructions retired by the processor, categorized into memory accesses (loads and stores), branches, and other instructions. Results normalized to RBDL on iiwa.



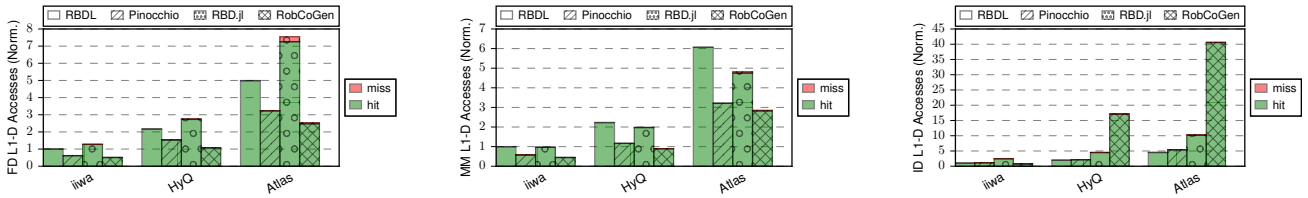
(a) Forward Dynamics Clock Cycles (b) Mass Matrix Clock Cycles (c) Inverse Dynamics Clock Cycles

Fig. 5. Total clock cycles, categorized into memory stall cycles, non-memory "other" stall cycles, and non-stall cycles. Results normalized to RBDL on iiwa.



(a) Forward Dynamics Floating-Point Operations (b) Mass Matrix Floating-Point Operations (c) Inverse Dynamics Floating-Point Operations

Fig. 6. Total floating-point operations, categorized into double precision operations with scalar, 128-bit packed vector, and 256-bit packed vector operands. Results normalized to RBDL on iiwa.



(a) Forward Dynamics L1 Data Cache Accesses (b) Mass Matrix L1 Data Cache Accesses (c) Inverse Dynamics L1 Data Cache Accesses

Fig. 7. L1 data cache accesses, categorized into hits and misses. Results normalized to RBDL on iiwa.

optimized code greatly reduces the number of superfluous instructions (Fig. 4b) that are not related to the core calculations of the algorithm. Once again, RobCoGen's unrolled loops cut down on its total number of branch instructions (Fig. 4b) and L1 data cache accesses (Fig. 7b) compared to the other implementations.

Pinocchio's good performance on this algorithm corresponds with an increased IPC throughput (Fig. 3b) relative to the other libraries.

C. Inverse Dynamics Results

For inverse dynamics, RobCoGen was the fastest library for iiwa (at 0.6μs). However, for the floating base robots, RobCoGen was the slowest library by far, and Pinocchio gave the fastest runtimes, at 1.5μs and 3.5μs for HyQ and Atlas, respectively (Fig. 2a). This is due to the much larger number of instructions executed. In these cases, RobCoGen executes many more loads and stores (Fig. 4c) than the other libraries, and it also performs many more floating-point operations (Fig. 6c).

To understand this degradation of performance, recall that RobCoGen in fact implements a hybrid dynamics algorithm for floating base robots (see Section III). For more detail, we profiled RobCoGen with the software profiling tool Valgrind [32]. We found that there were a handful of function calls in RobCoGen's inverse dynamics routine that were generating a clear majority of the function calls, as well as the resulting high numbers of branch instructions and L1 data cache accesses observed in the data from the hardware performance counters (Figs. 4c and 7c). Upon reviewing the code, we found that this is likely caused by coordinate-frame transformation of composite rigid body inertias, which are intermediate result of the hybrid dynamics algorithm that RobCoGen uses for floating-base robots. We suspect that this operation can be easily and significantly optimized, as RobCoGen's mass matrix algorithm already uses a more efficient implementation of the same coordinate transformation. In addition, this coordinate transformation is only needed as a result of RobCoGen's choice to use a hybrid dynamics algorithm for floating-base robots; the baseline recursive Newton-Euler algorithm used by other libraries does not require this step. For the iiwa robot, this effect does not come into play because RobCoGen does use 'regular' (non-hybrid) recursive Newton-Euler for fixed-base robots.

The runtimes for Pinocchio and RBDL are fairly close for inverse dynamics, with Pinocchio giving the shortest runtimes for the HyQ and Atlas robots. RBD.jl executes significantly more instructions overall than Pinocchio and RBDL, so it has a longer runtime than those libraries.

D. Memory Usage

All of the software implementations spent the clear majority of their cycles on execution (Fig. 5) rather than waiting for memory (stall cycles), so they can all be considered to be compute-bound, not memory-bound. In fact, most of the software implementations suffered almost no misses in the L1 data cache (Fig. 7), with average miss rates of all computations $< 1.4\%$. From this, we can see that for most of the implementations of these algorithms, the working set fits comfortably in the 32kB L1 data cache on this machine.

For inverse dynamics on HyQ and Atlas, RobCoGen has a much higher number of L1 memory accesses (Fig. 7c). This is caused by non-optimal access patterns related to the coordinate-frame transformation described in Section V-C. Again, this high number of memory accesses corresponds with a higher total number of instructions executed (Fig. 2c) and greatly increases the runtime of RobCoGen for inverse dynamics on floating-base robots.

A likely cause for RBD.jl's comparatively high number of memory accesses is its use of the integer frame annotations mentioned in Section III, which need to be copied along with each intermediate computation result.

E. Sources of Parallelism

All of the studied implementations are written sequentially at the top level, largely because they implement a set of recursive algorithms. As a result, there is currently no task-level parallelism exploited by any of the libraries. However, some future opportunities for task-level parallelism are within reach. For example, RBD.jl uses a world frame implementation (see Section II), which enables a loop interchange where joints of the same type can be stored in separate vectors and iterated over not in topological order. This presents an opportunity for task-level parallelism, however, it is not currently implemented in a parallel manner.

There was also data-level parallelism present in the floating-point workload (see Fig. 6). Interestingly, RBD.jl was the only implementation whose linear algebra library made significant use of the widest vector operations available, the 256-bit packed double precision floating-point instructions. The main source of these densely packed operations is Julia's use of the superword-level parallelism (SLP) vectorizer LLVM compiler pass in combination with unrolled code for small linear algebra operations generated by StaticArrays.jl. While RBD.jl took a performance hit by generating many more instructions overall than the other libraries (Fig. 4), a combination of instruction codebase efficiency and vectorization together could result in increased performance.

F. Sensitivity to Compiler Choice

All results presented in this section were taken from software compiled with Clang 6.0.1, but we performed some additional experiments to see the effect of compiling the non-Julia libraries (RBDL, Pinocchio, and RobCoGen) with GCC/G++ 7.4, released the same year as Clang 6.0.1. For RBDL and Pinocchio, results with GCC demonstrated degraded performance (*e.g.,* runtime increased by 54% for forward dynamics with Pinocchio). For RobCoGen, performance was roughly the same for forward dynamics and the mass matrix. The only case where performance improved at all from using GCC was with RobCoGen on the inverse (or rather, hybrid) dynamics benchmark for HyQ and Atlas (both

runtimes decreased by about 40%). However, these times are still far behind the best times observed for inverse dynamics using Clang (see Fig. 2). From these additional experiments, it is clear that compiler choice can have a large impact on performance for these applications.

VI. Discussion

In this section, we note several trends that span the different algorithms and implementations and describe some possible strategies for improving performance in future work. We also briefly speculate on the implications that our findings for the dynamics workloads might have for another related set of computations, the dynamics gradients.

A. Observed Trends

One clear trend in our results is that none of these calculations are memory-bound. Interestingly, all of them show extremely low L1 cache miss rates (except for Atlas with RBD.jl where they are merely low, see Fig. 7) despite having an unusually high proportion of load and store instructions (Fig. 4). From this, we conclude that these routines have very small working sets with large amounts of locality (either spatial or temporal). This is consistent with the majority of calculations being linear algebra routines on small arrays. However, the large proportion of loads and stores indicates that few operations are being performed on each fetched element before it is stored. This suggests that there will be an opportunity to improve performance by combining operations or reorganizing data access patterns to avoid loading and storing the same values repeatedly.

Another observation is that the scaling trends of the algorithms (see Section II), are also demonstrated by their corresponding software implementations. Performance and most other measures scale approximately linearly with robot complexity (*i.e.,* number of joints). The Atlas robot has approximately $4.4\times$ the number of joints of iiwa and takes about $6\times$ as long to calculate, on average. HyQ falls proportionally in the middle. This suggests that our results can be extrapolated to estimate performance for other robots using the degrees of freedom. However, there may be a point at higher numbers of joints where the internal matrices' sizes will exceed the L1 cache size and performance will degrade substantially.

One final observation relates to the use of floating-point operations. These routines vary considerably in how much they use packed (vector) floating-point instructions. Since the majority of the math they are doing is linear algebra, we would expect this workload to be highly amenable to vectorization. This suggests two things: 1) that having high-performance floating-point units with vector support will benefit these algorithms, and 2) that these implementations are probably not taking advantage of vector floating-point instructions as much as they could be.

B. Opportunities for Performance Gains

As previously mentioned, all of these implementations have some inefficiencies in their operation. The large proportion of load and store instructions indicates that there is a lot of overhead beyond what the mathematics requires. This view is supported by the lower overheads we see in RobCoGen and Pinocchio; for forward dynamics and mass matrix, they perform significantly fewer loads and stores than the other libraries. RobCoGen further reduces its total number of instructions by unrolling loops and eliminating conditional branching, efficiently reducing overhead. Careful profiling may expose additional opportunities.

A major opportunity for performance gains for the dynamics algorithms would be better use of parallel resources. There would seem to be room for improvement in all three types of parallelism: instruction, data and task. While the instruction-level parallelism (ILP) we measured was reasonable, the processor in our machine is capable of much more. In addition, there is much variability in the use of 128- and 256-bit vector operations indicating that these highly-efficient data-parallel operations may be under-utilized. Finally, none of the implementations make effective use of task-level (a.k.a. thread-level) parallelism. This means that three of the four cores in our testbed machine went unused. As trends in processor architectures are towards greater parallelism rather than greater single-thread performance, it would be worthwhile to exploit these resources.

C. Implications for Dynamics Gradients

As novel control techniques push more of the motion planning workload to the low-level, high-rate part of a robot control architecture, an important requirement will be fast evaluation of gradients of the dynamics. This is because motion planning techniques typically employ gradient-based optimization and local approximations of the dynamics.

Various approaches can be employed to compute gradients of dynamics-related quantities. Perhaps the easiest but crudest technique is numerical differentiation. Automatic differentiation may also be employed, which exploits the chain rule of differentiation to compute exact gradients evaluated at a point, often at a reduced computational cost compared to numerical differentiation. Employing automatic differentiation requires writing the dynamics algorithms in a generic (in C++, templated) way, so that non-standard input types that carry derivative-related information may be used. Pinocchio, RBD.jl, and a fork of RobCoGen [5] are written to support such non-standard inputs. Further performance gains may be achieved using analytical derivatives that exploit the structure of rigid body dynamics, as currently only implemented by Pinocchio [8].

The relation between the performance of algorithms for dynamics quantities and for their gradients is perhaps clearest for numerical differentiation, where the original algorithm is simply run once for each perturbation direction. However, each of these gradient computation approaches has clear links to the basic algorithms analyzed in this paper. As such, we expect insights gained and performance gains made for the basic dynamics algorithms to extend to gradient computations to a large degree.

We also note that if gradients are required, it may be more worthwhile to utilize task-level parallelization in the

computation of gradients, rather than in the basic algorithms themselves, because gradient computations can be trivially parallelized with one partial derivative per thread, while we also expect lower threading overhead due to a more significant workload per thread.

VII. Conclusion

The calculation of robot dynamics is an essential and time-consuming part of controlling a highly-articulated robot. This paper introduced a new benchmark suite with four different implementations of the three most commonly-used dynamics calculations. Our initial analysis of the suite provides information to help steer future optimization and acceleration of this workload.

Our goal in this work was to provide researchers and library implementers with information they can use to understand and optimize these calculations in the future. We found significant similarities and differences between the implementations, which helps to highlight which characteristics are intrinsic to the problem and which are due to the particular implementation. Key insights are that all of these implementations have small working sets and are not highly parallelized. However, it is also clear that they differ in the efficiency with which they perform the essential calculations, leading us to believe that there are still significant opportunities for improvement.

One promising avenue for future work is the use of parallelism. These implementations do not fully exploit the parallelism currently available on even modest desktop computers. It is possible that by using different algorithms or data structures, additional parallelism can be exposed and exploited. With acceleration from harnessing parallelism and other software and hardware techniques, we are optimistic that sufficient improvements can be made to enable the exciting new control algorithms currently under development.

References

[1] N. Correll, K. E. Bekris, D. Berenson, O. Brock, A. Causo, K. Hauser, K. Okada, A. Rodriguez, J. M. Romano, and P. R. Wurman, "Analysis and observations from the first amazon picking challenge," *IEEE Transactions on Automation Science and Engineering*, vol. 15, 2018.

[2] E. Krotkov, D. Hackett, L. Jackel, M. Perschbacher, J. Pippine, J. Strauss, G. Pratt, and C. Orlowski, "The DARPA robotics challenge finals: results and perspectives," *Journal of Field Robotics*, vol. 34, 2017.

[3] M. Diehl, H. J. Ferreau, and N. Haverbeke, "Efficient numerical methods for nonlinear mpc and moving horizon estimation," in *Nonlinear model predictive control*. Springer, 2009, pp. 391–417.

[4] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012.

[5] M. Neunert, C. De Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli, "Fast nonlinear model predictive control for unified trajectory optimization and tracking," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016.

[6] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, vol. 37, 2015.

[7] T. Koolen and contributors, "RigidBodyDynamics.jl," 2018. [Online]. Available: https://github.com/JuliaRobotics/RigidBodyDynamics.jl

[8] J. Carpentier, F. Valenza, N. Mansard *et al.*, "Pinocchio: fast forward and inverse dynamics for poly-articulated systems, 2015–2018." [Online]. Available: https://stack-of-tasks.github.io/pinocchio

[9] M. Naveau, J. Carpentier, S. Barthelemy, O. Stasse, and P. Souères, "Metapod: Template meta-programming applied to dynamics: Copcom trajectories filtering," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*. IEEE, 2014.

[10] M. L. Felis, "RBDL: an efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10514-016-9574-0

[11] F. Marco, B. Jonas, D. G. Caldwell, and S. Claudio, "RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on domain specific languages," *Journal of Software Engineering in Robotics*, vol. 7, 2016.

[12] M. A. Sherman and D. E. Rosenthal, "SD/FAST," 2013. [Online]. Available: www.sdfast.com/

[13] B. Plancher and S. Kuindersma, "A performance analysis of parallel differential dynamic programming on a gpu," in *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2018.

[14] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2008.

[15] J. Y. Luh, M. W. Walker, and R. P. Paul, "On-line computational scheme for mechanical manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 102, 1980.

[16] M. W. Walker and D. E. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *Journal of Dynamic Systems, Measurement, and Control*, vol. 104, 1982.

[17] R. Featherstone, "The calculation of robot dynamics using articulated-body inertias," *The International Journal of Robotics Research*, vol. 2, 1983.

[18] M. Quigley, K. Conley, B. 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*, vol. 3, no. 3.2. Kobe, Japan, 2009.

[19] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[20] G. Guennebaud, B. Jacob, P. Avery, A. Bachrach, S. Barthelemy *et al.*, "Eigen v3," 2010. [Online]. Available: https://eigen.tuxfamily.org/

[21] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous Robots*, vol. 40, 2016.

[22] N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, A. S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater *et al.*, "Valkyrie: NASA's first bipedal humanoid robot," *Journal of Field Robotics*, vol. 32, 2015.

[23] A. Stentz, H. Herman, A. Kelly, E. Meyhofer, G. C. Haynes, D. Stager, B. Zajac, J. A. Bagnell, J. Brindza, C. Dellin *et al.*, "CHIMP, the CMU highly intelligent mobile platform," *Journal of Field Robotics*, vol. 32, 2015.

[24] Intel Inc., "Intel® 64 and ia-32 architectures software developer's manual," *Volume 4: Model-Specific Registers.*, 2018.

[25] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010.

[26] S. Danisch and contributors, "PackageCompiler.jl," 2018. [Online]. Available: https://github.com/JuliaLang/PackageCompiler.jl

[27] KUKA AG, "LBR iiwa," accessed in 2018. [Online]. Available: https://www.kuka.com/products/robotics-systems/industrial-robots/lbr-iiwa

[28] C. Semini, "HyQ - design and development of a hydraulically actuated quadruped robot," *Doctor of Philosophy (Ph. D.), University of Genoa, Italy*, 2010.

[29] Boston Dynamics, "Atlas - the world's most dynamic humanoid," accessed in 2018. [Online]. Available: https://www.bostondynamics.com/atlas

[30] RobCoGen team, "urdf2kindsl," 2018. [Online]. Available: https://bitbucket.org/robcogenteam/urdf2kindsl/

[31] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: new microarchitecture code-named skylake," *IEEE Micro*, vol. 37, 2017.

[32] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007.