RoboX: An End-to-End Solution to Accelerate Autonomous Control in Robotics

Jacob Sacks, Divya Mahajan, Richard C. Lawson, Hadi Esmaeilzadeh

Alternative Computing Technologies (ACT) Lab
Georgia Institute of Technology
{jacks, divya_mahajan, rlawson}@gatech.edu

University of California, San Diego
hadi@eng.ucsd.edu

Abstract—Novel algorithmic advances have paved the way for robotics to transform the dynamics of many social and enterprise applications. To achieve true autonomy, robots need to continuously process and interact with their environment through computationally-intensive motion planning and control algorithms under a low power budget. Specialized architectures offer a potent choice to provide low-power, high-performance accelerators for these algorithms. Instead of taking a traditional route which profiles and maps hot code regions to accelerators, this paper delves into the algorithmic characteristics of the application domain. We observe that many motion planning and control algorithms are formulated as a constrained optimization problems solved online through Model Predictive Control (MPC). While models and objective functions differ between robotic systems and tasks, the structure of the optimization problem and solver remain fixed. Using this theoretical insight, we create RoboX, an end-to-end solution which exposes a high-level domain-specific language to roboticists. This interface allows roboticists to express the physics of the robot and its task in a form close to its concise mathematical expressions. The RoboX backend then automatically maps this high-level specification to a novel programmable architecture, which harbors a programmable memory access engine and compute-enabled interconnects. Hops in the interconnect are augmented with simple functional units that either operate on in-flight data or are bypassed according a micro-program. Evaluations with six different robotic systems and tasks show that RoboX provides a 29.4× (7.3×) speedup and 22.1× (79.4×) performance-per-watt improvement over an ARM Cortex A57 (Intel Xeon E3). Compared to GPUs, RoboX attains 7.8×, 65.5×, and 71.8× higher Performance-per-Watt to Tegra X2, GTX 650 Ti, and Tesla K40 with a power envelope of only 3.4 Watts at 45 nm.

Keywords—Accelerators; domain-specific languages; DSL; in-network computing; model predictive control; MPC; robotics

I. INTRODUCTION

Advances in robotics have had a revolutionary impact on many diverse sectors, ranging from space exploration [1] to medicine [2] to manufacturing [3]. Although robots are set to transform many enterprise sectors, enabling true autonomy and adaptation is predicated on compute-intensive motion planning and control algorithms. Adding improved compute capabilities using general-purpose platforms often requires larger batteries or shortening of the robot’s operational time. Higher capacity batteries increases the weight and/or form factor, which often does not comply with the application requirements [4]. Under these constraints, only a relatively small portion of battery capacity (power) can be allocated to compute resources. In other words, a robot is required to perform copious amount of computation in disproportionately small power envelopes. This discrepancy is more pronounced in smaller systems (e.g., micro/pico Unmanned Aerial Vehicles (UAVs)), which are increasingly gaining traction for use in retail assistance, home care, photography, and surveillance applications.

For instance, a popular consumer UAV, the DJI Phantom 2 [5], requires roughly 131.4 W to power its motors for 25 minutes, its maximum flight time. The exorbitant power consumption of the motors requires all other electronics and peripherals, including the camera, to operate under a restricted power budget of approximately 5.6 W. Such constraints limit the capabilities of the Phantom 2, as complex, autonomous maneuvers require more compute-intensive algorithms operating at greater control frequencies [6]. Providing such abilities with general-purpose hardware would be prohibitively more power-hungry and curtail the already short flight time. Furthermore, as the robot size decreases, the compute power dissipation can become comparable to that of its actuators [4,7]–[10]. These unique challenges call for innovative techniques that offer greater compute capabilities under constrained power budgets. This paper sets out to devise such a solution through hardware acceleration for motion planning and control. Another complementary alternative is to offload most of the computation to off-board local servers or remote clouds. Nonetheless, our hardware accelerator can be utilized to augment the servers and/or reduce their power consumption. Although our solution is agnostic to offloading, it opens the door for motion planning and control on the edge. Computation on the edge is specifically attractive for robotic applications such as disaster rescue missions with intermittent connectivity or military operations where continuous communication with the base station can compromise the stealth capabilities of the robot.

Recent works have focused extensively on acceleration for machine learning [11]–[19]; however, this work instead emphasizes robot motion planning and control. We aim to move beyond the traditional practices of acceleration, which rely heavily on identifying and mapping compute intensive kernels from existing libraries and applications to specialized hardware. Instead, we provide an end-to-end solution, named RoboX1, that enables roboticists to express the physics of the robot and its task in a novel high-level mathematical domain-specific language (DSL). In devising RoboX, we adopt roboticists’ general approach of describing the robot’s physical dynamics as a series of time-varying states, control inputs, and physical constraints. The robotic task is then expressed as a constrained optimization problem. Solving this problem outputs the trajectory of the control inputs and state variables over a discrete number of future time steps. However, pre-solving this problem once before execution does not account for environmental events. Therefore, the optimization problem is continuously solved online. A commonly used framework is Model Predictive Control (MPC) [6,20]–[29], which uses a mathematical model of the robot to predictively plan its behavior over a finite time horizon. As such, MPC is capable of anticipating future events whilst taking into consideration realistic constraints of the robot to plan its current action. A key insight is that while the robot models and task expressions differ across robotic systems and applications, the general structure of the MPC formulation is effectively invariant. RoboX’s DSL exploits this insight to provide high-level language constructs that enable modeling and expression of robotic applications close to roboticists intuition. The RoboX compiler then automatically maps this high-level specification to a novel programmable accelerator and alleviates the burden of programming the accelerator with low-level primitives and APIs.

1 RoboX is phonetically pronounced “ro - box.”

Appears in the Proceedings of the 45th International Symposium on Computer Architecture, 2018
As such the paper makes the following contributions.

1) To move beyond conventional approaches of profiling and partially mapping code regions to specialized hardware modules, we build our acceleration solution atop an algorithmic understanding of the application domain. We observe that many diverse robotic applications are formulated as constrained optimization problems which can be solved online using Model Predictive Control. Using this insight, we develop RoboX, an end-to-end acceleration solution for robotic motion planning and control.

2) RoboX encapsulates a domain-specific language which enables programmers to express robotic applications close to their concise mathematical description. We provide a compiler which transforms this high-level specification into a concrete MPC formulation and solver, which is mapped to the accelerator.

3) The RoboX architecture introduces compute-enabled on-chip interconnections, where hops integrate simple functional units. This functional unit, if required, can perform operations on in-transit data to reduce the burden on compute units. To support this architecture, we devise an ISA that offers three categories of instructions, where each program the interconnect, compute units, and memory interface, respectively. This flexibility allows efficiently executing different phases of robotic applications that alternate between dynamics and solver computation.

Using cycle-accurate simulations, we evaluate RoboX over six different robots: mobile robot, autonomous vehicle, manipulator, micro-satellite, quadrotor, and hexacopter. We compare RoboX performance per-watt improvement compared to the Tegra X2, GTX 650 Ti, and the controller then applies the first control decision to the robot. For example, in Figure 1b, we show a quadrotor at its initial state $x_0$ with an objective of reaching target state $x_f$, the star. The MPC problem converges to an optimal trajectory $\tau(t_0)$, and the controller changes course accordingly. The actual path taken by the quadrotor, $\hat{\tau}$, is now different than one initially planned.

The robot's task is specified as an objective function minimized over a number of future time steps, called the prediction horizon. Longer horizons can improve convergence and lead to more robust control in reaction to external disturbances and obstacles. For the quadrotor, the goals of reaching the target while avoiding the obstacle are incorporated as separate terms in the objective function:

$$\begin{align*}
J &= P_T(x(t_f), Q_T) + \int_{t_0}^{t_f} P_{ob}(x(t), Q_{ob})dt 
\end{align*}$$

where $P_T(\cdot)$ penalizes distance from the target, $P_{ob}(\cdot)$ penalizes proximity to an obstacle, and $Q_T$ and $Q_{ob}$ are weights which indicate the relative importance of each penalty. Penalties incorporated in the objective can be categorized as either running cost or terminal cost. The penalty for obstacle avoidance, $P_{ob}(\cdot)$, is the running cost, as it is enforced at every point along the horizon except at the final point $t_f$. The penalty corresponding to reaching the target is the terminal cost, as it solely considers the final state $x(t_f)$. The final constrained optimization problem then becomes:

$$\begin{align*}
\min_{x(\cdot), u(\cdot)} \ J(x(\cdot), u(\cdot)) \\
\text{s.t.} & \quad x(0) = x_0, \\
& \quad \dot{x} = f(x(t), u(t)), \forall t \in [0,t_f] \\
& \quad u \leq u(t) \leq \bar{u}, \forall t \in [0,t_f]
\end{align*}$$

Figure 1: (a) States and control inputs of a quadrotor. (b) Quadrotor planning trajectory $\tau(t_0)$ towards target (star) and adjusting course to $\hat{\tau}$ due to obstacle (balloon), while staying above altitude $z$ due to a constraint.
where \( x(t) \) and \( u(t) \) are the states and inputs at time \( t \), \( f(\cdot) \) is the quadrotor’s dynamics, and \( x_0 \) is the current state. We have placed constraints \( u \) and \( \dot{u} \) on the control input \( u \), which correspond to the physical limitations of the propellers. This formulation of the constrained optimization problem in Equation 4 is entirely independent of the solver.

As its solver, RoboX uses the primal-dual interior point method [30]. We chose this method as it is commonly used in constrained robotics applications [31]–[36]. This method first discretizes the trajectory over a horizon of length \( N \):

\[
z = [x_i^T \quad \Delta x_i^T \quad u_i^T \quad \Delta u_i^T] \quad (5)
\]

where \( x_i \) and \( u_i \) are state and input vectors at each discrete time point \( i \). Then, all equality constraints corresponding to the quadrotor’s current position and dynamics are concatenated along the horizon into a vector \( g \). Similarly, we construct a vector \( h \) which contains all inequality constraints corresponding to the bounds on the propeller thrusts along the horizon. Instead of directly solving for the states and inputs of the trajectory, interior point methods solve for updates \( \Delta x \) to the current estimates. This solution is found by solving the sparse linear system of the form:

\[
\begin{bmatrix}
H & g_{\Delta x}^T & h_{\Delta x}^T \\
g_{\Delta x} & I & 0 \\
h_{\Delta x} & 0 & S
\end{bmatrix} \begin{bmatrix}
\Delta x \\
\Delta u \\
\Delta \lambda
\end{bmatrix} = - \begin{bmatrix}
J_{\Delta x} \\
n \\
h
\end{bmatrix}
\]

(6)

where \( J_{\Delta x} \) and \( H \) are gradient and Hessian of the objective function, and \( g_{\Delta x} \) and \( h_{\Delta x} \) are gradients of the constraints. The variables \( \Delta x \), \( \Delta u \), and \( \Delta \lambda \) are updates to special variables, known as KKT multipliers and slack variables, used by the interior point method to account for constraints. The matrices \( S, \Lambda, I \) are \( \text{diag}(\sigma), \text{diag}(\lambda) \), and the identity matrix, respectively. RoboX uses a combination of Cholesky decomposition [35] and forward/backward substitution [37] to solve the linear system in Equation 6. The solution provides updates \( \Delta x \), which are applied to the current trajectory. This process is repeated until convergence, after which the control decisions are supplied to the quadrotor.

The algorithms described above provide the theoretical foundation for RoboX to target a wide range of robotic applications. A robotist simply provides a high-level description of the physics and constraints of the robot and a mathematical expression of its objective. The backend of RoboX then translates this high-level specification into the corresponding constrained optimization problem and generates the concrete Interior Point Method solver.

III. ROBOX WORKFLOW

RoboX aims to enable robotists to benefit from acceleration without departing from their concise mathematical formulations. Using these expressions, RoboX automatically generates the optimization problem and solver and compiles it into static schedules for the accelerator. We discuss each component shown in Figure 2.

IV. DSL FOR ROBOTIC CONTROL

RoboX constructs a novel domain-specific language (DSL) to meet the following criterion: (1) provide a modular interface which distills optimal control into its core components and (2) be close to the concise mathematical expressions while remaining independent of implementation. We decompose MPC problems into model and objective formulation. The RoboX language provides System and Task components to represent these elements in a modular fashion. We tackle the second criteria through symbolic computation and group operations. This approach circumvents the need for explicit loops, allowing the code implementation to be independent and simplifying parallelism identification. By keeping
the abstraction at the level of constrained optimization, we avoid tying the program to implementation-specific decisions, such as choice of solver or discretization method.

A. System Components

A system definition is denoted using the System construct and encapsulates all states, control inputs, constraints, and robot dynamics. The state and input keywords declare the robot states and control inputs, respectively. Both are composite datatypes that consist of lower_bound and upper_bound fields, which express physical constraints of the robot. Additionally, the state datatype contains a dt field, which represents the state time derivative. Constant parameters are defined with the param keyword. A summary of all the keywords in the RoboX language are provided in Table I. We illustrate the System definition of a simple mobile robot in the following code snippet.

```
System MobileRobot(...) {
    // system states
    state pos[2], angle;
    // system inputs
    input vel, ang_vel;
    // system dynamics
    pos[0].dt = vel * cos(angle);
    pos[1].dt = vel * sin(angle);
    angle.dt = ang_vel;
    // physical constraints
    vel.lower_bound <= -vel_bound;
    vel.upper_bound <= vel_bound;
    // penalize distance from target
    penalty target_x, target_y;
    target_x.terminal = pos[0] - desired_x;
    target_y.terminal = pos[1] - desired_y;
    target_x.weight <= weight;
    target_y.weight <= weight;
    // constraints on position
    constraint pos_bound;
    pos_bound.running = pos[0] + pos[1];
    pos_bound.upper_bound <= radius;}
```

The above System component defines the robot’s state using the 2D array pos and angle variables, which correspond to the robot’s position and orientation, respectively. The control inputs are the velocity (vel) and the angular velocity (ang_vel). RoboX offers two forms of assignments, symbolic (=) and imperative (<=). Imperative expressions are immediately evaluated in program order and are used for expressing physical constraints and parameters. For example, the lower_bound and upper_bound fields of the control input vel above are imperatively assigned. Symbolic expressions declare the formal relationship between variables, such how the state derivative relates to the other states and the control inputs. In the snippet above, the time derivatives of the position states are symbolically assigned to each element of the vector’s dt field. These symbolic variables can be used in other symbolic expressions to more complex mathematical functions.

B. Task Components

The second component of a RoboX program is a description of the robot’s task. A task is broken down into a series of penalty terms and task-specific constraints. The construct Task denotes a task definition, and the penalty and constraint keywords are used to specify the penalty terms and constraints, respectively. Both are assigned expressions which contain at least one state or input variable. A penalty has a weight field, which sets the relative penalty weight in the objective (default is one). The constraint variable has lower_bound and upper_bound fields to express inequality constraints and an equals field to define an equality constraint. Going back to the mobile robot, we provide an example which instructs the robot to move to a fixed target.

```
System MobileRobot(...)
    reference desired_x, desired_y;
    param weight, radius;
    // penalize distance from target
    penalty target_x, target_y;
    target_x.terminal = pos[0] - desired_x;
    target_y.terminal = pos[1] - desired_y;
    target_x.weight <= weight;
    target_y.weight <= weight;
    // constraints on position
    constraint pos_bound;
    pos_bound.running = pos[0] + pos[1];
    pos_bound.upper_bound <= radius;}
```

The code snippet defines a task moveTo, which specifies terminal penalties, target_x and target_y. The penalty is the difference between some fixed constant location (desired_x, desired_y) and the robot’s current location (pos[0], pos[1]). This encourages the robot towards the target location at the end of the trajectory. A running constraint term pos_bound is defined which instructs the robot to be within a circle of radius radius. The penalties and constraints are indicated as running or terminal by assigning the expression the variable’s running or terminal field, respectively. The example also introduces the reference datatype, which is used to express information from external sources. For example, the target location moveTo may be determined by an external device performing object detection. We illustrate how to utilize references through the following example.

```
reference desired_x;
reference desired_y;
MobileRobot robot(1, 0.01);
robot.moveTo(desired_x, desired_y, 1);
```

A MobileRobot is instantiated as robot and the desired task is called like a method. The references desired_x and desired_y are defined globally and passed to the moveTo task.

C. Mathematical Operations

The RoboX DSL also supports a wide range of mathematical operations. Although our DSL’s mathematical expressions share a resemblance to the language in TABLA [19], the latter offers neither symbolic constructs nor complex primitives that enable expressing the robot physics. Supported operations are categorized into (1) elementary, (2) nonlinear, and (3) group operations, as shown in Table I. Elementary operations consist of addition (+), subtraction (−), multiplication (×), and division (÷). Nonlinear operations comprise nonlinear functions such as cos, sin, tan, acos, asin, atan, exp, and sqrt. Both types may be used to compose symbolic or imperative expressions. The RoboX language performs operations over multi-dimensional variables with group operations. Group operations are declared with the range datatype, which provides a concise means to express accesses over multi-dimensional variables

<table>
<thead>
<tr>
<th>Type</th>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Task</td>
<td>Definition of robot type comprising all necessary attributes</td>
</tr>
<tr>
<td>Data Types</td>
<td>input</td>
<td>Robot control input</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>Robot system state</td>
</tr>
<tr>
<td></td>
<td>param</td>
<td>Constant parameter</td>
</tr>
<tr>
<td></td>
<td>penalty</td>
<td>Term maintained during Task execution</td>
</tr>
<tr>
<td></td>
<td>constraint</td>
<td>Constraint on terms in Task</td>
</tr>
<tr>
<td></td>
<td>reference</td>
<td>Potentially time-varying term used to define penalties</td>
</tr>
<tr>
<td>Fields</td>
<td>lower_bound</td>
<td>Defines an range for array access and group operations</td>
</tr>
<tr>
<td></td>
<td>upper_bound</td>
<td>Fields of lower_bound and upper_bound fields</td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Equality constraints on a variable</td>
</tr>
<tr>
<td></td>
<td>running</td>
<td>Indicates enforced everywhere except last step of horizon</td>
</tr>
<tr>
<td></td>
<td>terminal</td>
<td>Indicates enforced only at last step of horizon</td>
</tr>
<tr>
<td></td>
<td>dt</td>
<td>Time derivative of state variable</td>
</tr>
<tr>
<td></td>
<td>weight</td>
<td>Relative weight of penalty term</td>
</tr>
<tr>
<td>Mathematical Operations</td>
<td>sin, cos, tan</td>
<td>Nonlinear operations</td>
</tr>
<tr>
<td></td>
<td>sum, norm, max, min</td>
<td>Group operations</td>
</tr>
</tbody>
</table>

Table I: Language constructs of RoboX.
without the need for explicit loops. Supported group operations include sum, norm, min, and max. A sum operation adds the elements of an expression over the range indicated by the range variable and yields a scalar. Similarly, a norm computes the Euclidean norm and min and max operations compute the min and max of an expression. For instance, the constraint pos_bound in the moveTo task is actually a norm operation over the position vector:

\[ \text{range } 1[0:2]; \]
\[ \text{pos\_bound[i].running = norm[i](pos[i])}; \]

In this example, the range variable \( i \) is defined over the interval \( (0, 2) \). We can also use a range variable coupled with group operations to perform matrix operators:

\[ \text{state } x[i], R[i][j]; \]
\[ \text{range } i[0:2], j[0:2]; \]
\[ x[i].dt = \text{sum}[j](R[i][j] \times x[j]); \]

In this example, the time derivative of the state vector \( x \) corresponds to \( \frac{dx_i}{dt} = \sum_j R_{ij} x_j \). The range for the group operation is provided in the brackets while the expression is placed within the parenthesis. The range variables \( i \) and \( j \) correspond to the indices in the mathematical formulas that implicitly define a loop. These variables help RoboX identify sources of parallelism in the computations. For instance, it is clear from the code snippet that the computation of each element \( x[i].dt \) is independent.

RoboX’s DSL aims to provide constructs that enable expressing mathematical formulas in a textual representation with an almost one-to-one correspondence. From this DSL, RoboX automatically maps the robot dynamics and solver computation to the accelerator after inserting the parameterized solver code. The next section discusses the unique architecture of RoboX’s programmable accelerator and how it provides the necessary flexibility for the robotics domain.

V. ACCELERATOR ARCHITECTURE

In devising the RoboX architecture, we aim to exploit parallelism available from both the dynamics and solver computations. Data dependencies between operations during dynamics computation for a given time step have limited instruction-level parallelism. In contrast, much of the solver computation involves matrix operations, which exhibit more data parallelism. Therefore, we chose a flexible dataflow architecture to support these two distinct workload phases. As Figure 3 illustrates, the accelerator architecture is organized into a two-level hierarchy of Compute Clusters (CCs), each of which contain a set of Compute Units (CUs). High dependency dataflow graphs necessitate a significant amount of data transfer from the source units to their destinations. This provides an opportunity to perform some computation in the interconnect rather than the destination. Thus, we propose a compute-enabled interconnection fabric which offers simple compute capabilities over in-flight data.

Additionally, a programmable memory access engine actively fetches data rather than passively responding to requests from compute elements. The interconnect and access engine are programmable to simplify the hardware and avoid complex hardware-based arbitration and handshaking between compute and memory elements. These three elements process their own separate statically-scheduled micro-instructions, detailed in Section VI. The following elaborates on each microprogrammable component.

**Compute-enabled interconnect.** Enabling computation in the interconnect needs to add minimal overhead to its normal operation. As such, we propose to augment each hop of the busing system with a limited capability functional unit to operate on the in-transit data. In RoboX, we add a multiply-add unit, which is frequently used in reduction operations for our target domain. Either the fli of data can specify to perform an operation or a preloaded queue in the hop may contain the schedule for operating on the transiting data. A shift register is sufficient for the hops in the RoboX’s architecture, in which the interconnect is preprogrammed with a static schedule and the hops support a single function. A 0 in the shift register indicates that the operation will be bypassed and the normal data delivery is needed. A 1, on the other hand, engages the functional unit in the hop. As Figure 3 illustrates, the CCs are connected through such a compute enhanced tree-bus interconnect. The tree-bus organization of the CCs is beneficial for reduction patterns, which exhibit high data dependencies and are common in matrix multiplications, Euclidean norms, and other solver operations. To further speed these reduction operations, as depicted in Figure 3, the tree bus employs our compute-enabled interconnect concept as defined above. Note that the complex operations are always deferred to the CUs for execution to avoid over complicating the interconnect.

Within the CC, as Figure 4a shows, the CUs are connected via shared bus and also single-hop connections between neighboring CUs. These single-hop connections facilitate low-overhead communication between neighboring CUs, circumventing the shared bus. These hops also have access a multiply-add unit...
to perform operations while passing the data along. While the inter-CU hops can perform aggregation for smaller data arrays, rows of larger arrays can be parallelized across the CCs. Thus, the partial row computation can be performed in a CC and aggregated across the CCs through their compute-enabled interconnect. This feature in the architecture is very useful for parallelizing group operations in the dynamics and penalty computation phases.

**Compute clusters.** Figure 4a shows a single CC. Each CC contains separate queues for the compute and communication microprograms. These microprogram is the result of statically scheduling the compute and communication instructions. As such, CUs do not initiate communication requests but merely consume data available, simplifying the CC design and busing logic. Communication micro-instructions dictate the inter-CU communication through the shared bus by indicating the source and the destination CU(s). As shown, the CUs are connected to the bus through a FIFO. Shared bus communication follows either a one-to-one, one-to-many, or one-to-all broadcast pattern. This is of particular importance in dynamics computation, where multiple CUs may require the same piece of data produced by a single CU in the CC. The compute micro-instructions determine the operations for all CUs in the CC. The CUs can perform distinct operations on unique data to exploit the fine-grained parallelism in the DFG, or the CC can operate in SIMD mode through vector operations. The SIMD mode is useful for performing element-wise multiplication on arrays, while the inter-CU compute-enabled hops aid in reduction of the products. Both SIMD mode and compute-enabled interconnect assists in efficiently performing group operations in the robot dynamics and task penalty computation.

**Compute unit.** Figure 4b illustrates the organization of a single CU, which comprises buffers, registers, ALU, nonlinear operations look up tables, and associated busing logic. We separate the CU memory into separate buffers according to a set of namespaces allocated by the ISA, as discussed in Section VI. Demarcating the buffers allows for parallel access to provide all operands simultaneously. Buffers are implemented as queues, where each element can be optionally popped and discarded or rewritten back to enable reuse. Dedicated registers enable communication between neighboring CUs. Each CU has a three-staged pipeline that access, compute, and write the data. Supported operations are addition, subtraction, multiply, and nonlinear functions as lookup tables. Due to large area of division, it is only supported by one CU per CC. Similarly, to prevent the excessive overhead of large LUTs for every nonlinear operation, each CU only supports two such operations.

**Programmable memory access engine.** RoboX provides a memory access engine, programmed according to the static schedule of the operations. The programmability allows dealing with misaligned data to prevent bandwidth under-utilization. An integrated shifter properly aligns data according to the microprogram loaded into the engine. To hide the latency of external memory accesses, the engine *prefetches* instructions and data according to its schedule and loads them into global instruction and load buffers, respectively. Similarly, all the final results from the compute elements are stored by the access engine based on its microprogram.

Through a compute-enabled interconnect and static microprogramming of the memory access engine, the RoboX architecture allows interconnection and memory to behave as active elements. This design contrasts with traditional architectures, where compute elements actively initiate requests and memory and interconnection subsystems are passive. Next, we discuss RoboX’s unique instruction set abstraction to efficiently express the component interactions.

**VI. Instruction Set Architecture**

To support the RoboX architecture, we propose a novel ISA which (1) splits a program into separate compute, communication, and memory instructions; (2) abstracts away hardware implementation details; and (3) allows static scheduling at compile time. Table II shows the instructions in the ISA for each of the three categories, which are all encoded in 32 bits. These high-level instructions are converted to microprograms that represents the schedule for the for the inter- and intra-CC bus, bypass bit patterns, the compute-enabled hops in the interconnect, and operations for the CUs. Additionally, the ISA remains independent of the implementation of the compute-enabled interconnects. Instructions simply express the group operation performed and at what granularity. Below, we discuss the organization of the ISA and the details of the individual instructions.

**Namespaces.** To simplify the layout of data in memory and facilitate inter- and intra-CC communication, the ISA exposes a set of namespaces which organize data into their respective categories. All instructions share the namespaces INPUT, STATE, GRADIENT, and HESSIAN. Computation and communication instructions also have the namespaces INTRIN, LEFT NEIGHBOR, and RIGHT NEIGHBOR. Memory instructions have namespaces REFERENCE and INSTRUCTION. These namespaces semantically separate data and simplify the communication instructions. As discussed in Section V, RoboX implements most of these namespaces as queues and registers. The memory namespace INSTRUCTION holds all instructions for RoboX to execute. There also needs to be a designated location in memory for external environmental data not captured directly by the state, such as the location of a target or bounds of a racing track. As such, we provide the REFERENCE namespace to hold all such external data relevant for penalty or constraint computation. Memory is partitioned according to these namespaces and determines the layout of all data accordingly.

**Compute instructions.** The compute instructions dictate the computation local to a given Compute Cluster. The supported functions are the same as the elementary and nonlinear operations provided to the programmer by the RoboX language. As Table II shows, these instructions are divided into scalar and SIMD operations and further broken down into queue vs. immediate operations. Scalar instructions indicate the operation to be performed by an individual CU, while SIMD instructions have all the CUs in the CC perform the same operation. Additionally, SIMD instructions use a repeat field, which tells the CC to repeat the SIMD operation a pre-specified number of times but on different data. This strategy reduces the instruction count. Queue instructions require a queue namespace and index to be specified for each source, while immediate instructions allow for one of the sources to be an 8-bit integer. Only the top 8 elements of the queue are addressable. However, each queue-type instruction has a dedicated Pop field which dictates whether the data should remain in the queue after access, discarded after use, or popped and rewritten for later reuse. All of the sources and destinations specified by the instructions are local to the CU which
executes it. As such, the CUs are not concerned with data transfer and simply perform computation on data available in the queues.

**Communication instructions.** Communication instructions shown in Table II orchestrate the intra- and inter-cluster data transfer and in-bus computation. To improve the scalability of the RoboX ISA, CUs within a CC and CCs themselves are organized into quarters. Data transfer instructions comprise Unicast, Multicast, and Broadcast communication styles. A Unicast transfers data from a single CU to another, potentially in differing CCs. The multicast type is a one-to-many communication style, where a CU sends data to either a subset of the CUs within its CC (CU Multicast) or to all CUs within a subset of CCs (CC Multicast). A dedicated field indicates the target CU or CC quarter, and mask bits are used to specify the recipient CU or CC within the quarter. Finally, the Broadcast instruction transfers a data element from a single CU to all the CUs on the accelerator.

**Compute-enabled interconnect instructions.** Instructions for the compute-enabled interconnect include CU Aggregation and CC Aggregation, which perform a pre-specified group operation over the CUs within a CC or across all CCs, respectively. The supported aggregation functions are ADD, MUL, MIN, and MAX. Group operations in the DSL are compiled to a combination of in-bus aggregation and SIMD operations. The sum, min, and max group operations can be implemented directly with their corresponding aggregation functions. However, the norm function cannot be implemented with a single aggregation instruction. Instead, this operation is carried out by sequentially applying a MUL aggregation, an ADD aggregation, and a SQRT operation on the result.

**Memory instructions.** As discussed above, the memory is also organized into different namespaces. Each portion of memory corresponding to a namespace is further subdivided into multiple blocks to enable a fixed-sized instruction to access the full range of memory addresses. The namespace and offset from the current namespace’s block pointer is provided as a field in the Load and Store instructions. A Set Block instruction changes the current block number to the one specified for the indicated namespace. Furthermore, as the ISA is designed to be statically scheduled, RoboX leverages this blocking to organize data in the memory efficiently. To cope with data misalignments, the Load and Store instructions also provide a shift field.

**VII. Compilation Workflow**

The compilation workflow has two phases: (1) Program Translator and (2) Controller Compiler. The Program Translator takes as input a RoboX program and generates a M-DFG of the entire MPC algorithm. The Controller Compiler uses this M-DFG to generate the final static schedules.

**Program Translator.** In RoboX, the solver and discretization method are fixed, allowing us to express it as an invariant yet parameterized code. These parameters are set by the dynamics, penalties, and constraints defined by the RoboX program, as well as the meta-parameters, such as the horizon length and desired controller rate. The Program Translator first assigns an ordering to the states and inputs of the robot. It organizes penalties and constraints into separate running and terminal groupings. The objective function is a summation of the weighted Euclidean norms of each penalty $\sum_i||p_i||^2_W$, where $W_i$ is a diagonal matrix of each weight assigned to the penalty’s weight field. The Program Translator extracts the computation for each of these components and uses automatic differentiation to compute all necessary gradients. Each construct in the RoboX language has a corresponding M-DFG node representation. Elementary and nonlinear operations are simply single SCALAR type nodes with edges expressing its dependencies. Any elementary or nonlinear operation which is defined over an interval specified by a range variable is a VECTOR node. Lastly, group operations are represented by single GROUP aggregation node. Internally, a GROUP node is an ARRAY node which also specifies the aggregation to perform over its results. The Program Translator constructs the final M-DFG by generating the nodes of all the expressions in the RoboX program and merging them according to its solver template.

**Controller Compiler.** The Controller Compiler takes the M-DFG as input and constructs separate operation, data, communication, and aggregation maps. Specifically, the operation map assigns all the M-DFG operations to the CUs except for those executed in the interconnection fabric. The aggregation map keeps track of the CUs and all associated KKT, constraints, and reference variables. Finally, the communication map enumerates which CUs receive each piece of produced data during program execution.

The controller compiler first constructs an initial data map $D$ by pre-assigning the location of DFG operands, or graph edges, which correspond to state or input variables. This data is assigned according to the CU ordering determined by Program Translator, number of prediction time steps, and number of states in the robot model. The Controller Compiler uses an Algorithm 1 to generate mappings and takes as input the DFG graph, initial data map $D$, number of CUs per CC ($\text{ncu}$), and the total number of CUs ($\text{nCUs}$). Its output is a program map $M$, which contains an operation ($M.\text{O}(\text{nCUs})$) and data ($M.\text{D}(\text{nCUs})$) map as an array of lists indexed by the CU. There is also a communication ($M.\text{C}(|E|)$)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar Queue Op</td>
<td>Opcode = 000</td>
<td>Opcode = 001</td>
<td>Opcode = 010</td>
<td>Opcode = 011</td>
<td>Opcode = 010</td>
<td>Opcode = 011</td>
<td>Opcode = 010</td>
<td>Opcode = 011</td>
<td>Opcode = 010</td>
<td>Opcode = 011</td>
<td>Opcode = 010</td>
<td>Opcode = 011</td>
</tr>
</tbody>
</table>

**Table II: The RoboX ISA, which is divided into separate compute, communication, and memory instructions.**
Algorithm 1: Compute-Enabled Interconnect-Aware Mapping.

map which is indexed by an edge of the DFG and stores the CUs to which that data should be sent. Similarly, there is an aggregation graph \((M.A[|V|])\) which is indexed by the vertex of a GROUP operation. The algorithm then proceeds as follows:

1. Initialize the operation, communication, and aggregation maps to null, the data map to \(D\), the graph variable \(G\) to the MD-FG, and zero the CU counter (\(c_{\text{total}}\)).
2. Select a ready vertex \((v)\), meaning all its parents have been assigned. Then, iterate through all operations (op) in vertex \(v\) SCALAR nodes will only have one operation.
3. Check if a source has been mapped \((src)\) for op in vertex \(v\). If not, assign all source nodes to CU counter \((c_{\text{total}})\) and proceed to step (5).
4. Check for second source \((src_j)\) of op in vertex \(v\). If it exists and is not mapped, assign it to \(src_j\)'s CU and set the data map accordingly. Otherwise, inform the communication map that \(src_j\) should be sent to \(src_i\)'s CU.
5. Assign the operation \(op\) to the CU. If \(v\) is of type GROUP, add the CU to the aggregation list for vertex \(v\).
6. Reiterate steps 2 through 6 until all vertices are mapped. The Controller Compiler then uses \(M\) to statically schedule all operations, communication, and memory accesses. From the location of the CUs in each GROUP vertex aggregation map, the aggregation is either performed over the inter-CU hops in a CC or in the compute-enabled tree-bus.

VIII. Evaluation

Benchmark Robots and Tasks. Table III lists the six benchmark robot systems used to evaluate RoboX, which consist of a variety of types of robotic systems. In particular, MobileRobot [21] is a two-wheeled mobile robot employed in a trajectory tracking task. The Manipulator [24] is an arm-like robot comprising a cascade of joints and links. We consider a two-link manipulator executing a reaching task. The AutoVehicle [20] is a four-wheel autonomous vehicle moving on a racing task, where the goal is to maximize velocity. The racing track bounds correspond to position constraints on the car. The MicroSat [22] is a miniature satellite of low mass which must remain in proper orbit under potential disturbances. The Quadrotor [23, 27] and Hexacopter [6] are four- and six-rotor micro UAVs, respectively. Both are engaged in motion planning and orientation control, but have differing dynamics and constraints. This, in turn, changes the computational requirements of the controller. Table III lists each robot, which has differing numbers of states, inputs, and physical constraints and an associated task with a certain number of penalty terms and task-specific constraints. However, the computational requirements of each benchmark do not only depend on these parameters. For each time step in the horizon, the system dynamics, task penalty terms, and constraints need to be evaluated. The complexity of dynamics may significantly vary between robotic systems, even with a similar number of states and/or inputs. For instance, while Quadrotor and Hexacopter have the same number of states, the dynamics of the latter is more computationally intensive. The same is true for the computational requirements for different penalty terms and constraints between tasks.

A. Methodology

CPU Platforms. As shown in Table IV, we compare RoboX to two multicore CPUs running Ubuntu Linux version 16.04: (1) a high performance quad-core Intel Xeon E3 and (2) a low-power quad-core ARM Cortex A57 available on the Nvidia Jetson TX2. The baseline CPU implementation uses the ACADO Toolkit [34] to implement the optimized, self-contained C code. The code is compiled with GCC 5.4 with -O3 -ffree-vectorize to enable aggressive compiler optimization and vector operations for all platforms. The benchmarks use four threads on ARM and eight on Xeon, which supports simultaneous multithreading. ACADO is a high-level framework which supports multiple solvers. We chose the sparsity-exploiting HPMPC interior-point solver, as it demonstrated superior runtime performance over the other options. For a fair comparison, we use the same solver algorithm in RoboX. The HPMPC solver uses BLASFEO [38], which is a BLAS-like library tailored for small to medium matrices (up to a few hundred elements). For larger horizons, we used BLASFEO as a wrapper for the standard BLAS implementations.

GPU Platforms. We also compare RoboX with the three GPUs shown in Table IV: (1) a low-power Tegra X2 available on the Nvidia Jetson TX2, (2) a desktop-class GeForce GTX 650 Ti, and...
Figure 5: Speedup of Xeon E3 and RoboX over ARM A57 baseline.

Figure 6: Speedup of GPUs and RoboX over GTX 650 Ti baseline.

Figure 7: Performance-per-Watt improvement of Xeon E3 and RoboX over ARM A57 baseline.

Figure 8: Performance-per-Watt improvement of GPUs and RoboX over GTX 650 Ti baseline.

(3) a high-performance Tesla K40. Due to limited GPU implementations of interior-point solvers, we compared RoboX with our custom GPU code written in CUDA using cuBLAS. We made our best efforts to hand-tune the code for each GPU platform and optimize the number of blocks and threads-per-block. All benchmarks were compiled separately for each GPU using target-specific flags.

Execution time measurements. To obtain the execution time measurements for the CPUs and GPUs, we calculate the average based off the measured wall clock time of 10000 solver iterations. For the RoboX runtime estimates, we use a custom cycle-accurate simulator with parameters in Table IV. From our empirical study, we found 32-bit fixed-point with 17 fractional bits and 4096-entry LUTs were sufficient to make the effects on convergence negligible.

Power measurements. For the Xeon E3, we use the Intel Running Average Power Limit (RAPL) energy consumption counters available in the Linux Kernel. As the GTX 650 Ti does not support the NVML library but has the same microarchitecture as the Tesla K40, we make a conservative estimation of its power consumption by scaling the Tesla K40 measurements using the ratio of their TDPs. The ARM A57 and Tegra X2 are part of the Jetson TX2 development board, which does not provide a software mechanism to measure energy. Instead, we use the Keysight E3649 Programmable DC Power Supply to measure the power consumption. We subtract the idle average power consumption from the benchmark execution power reading. For the RoboX ASIC, we synthesized the accelerator with the Synopsys Design Compiler (L-2016.03-SP5) using TSMC’s 45-nm high Vt standard cell libraries to generate area and power estimates. As shown in Table IV, the synthesized accelerator has 512 KB of on-chip memory, an area of 8.13 mm², consumes 3.4 W, and operates at 1.0 GHz.

B. Experimental Results

CPU performance comparison. Figure 5 shows the speedup of RoboX and the Xeon E3 over an ARM A57 baseline for a prediction horizon of 32 steps. On average, RoboX has a 29.4× (7.3×) speedup over the ARM A57 (Xeon E3). The performance improvement ranges between 6.2× to 79.1× across the benchmarks. This variation can be attributed to the differences in the robot configurations and computational demands of different dynamics and tasks. For instance, MobileRobot has the lowest speedup, as it has least number of states, penalties, and constraints. In several cases, the observed speedup with RoboX is proportional to the number of states and complexity of the dynamics. The Hexacopter benchmark has the second largest speedup and has the greatest number of penalty terms and control inputs. In some other cases, such as the Manipulator benchmark, the complexity of the dynamics exposes enough opportunity for the accelerator to provide greater benefits despite a lower number of states.

GPU performance comparison. Figure 6 illustrates the speedup of RoboX compared with the Tegra X2 and Tesla K40 with a GTX 650 Ti baseline. RoboX provides an average speedup of 2.0× (3.5×) over the GTX 650 Ti (Tegra X2) for a prediction horizon of 32 steps. These benefits vary between 1.63× (2.89×) and 2.74× (5.17×) over the GTX 650 Ti (Tegra X2). In contrast, RoboX is 1.3× slower than the Tesla K40 on average. This is due to the fact that the Tesla has over twice the number of cores and operates under significantly greater power budget of 235 W.

Performance-per-Watt comparison. The computational resources for autonomous robotics often have to function under a tight power budget. Thus, to evaluate the performance benefits for a fixed energy consumption, we use the performance-per-watt as a metric of comparison. Figure 7 shows the performance-per-watt improvement of RoboX and the Xeon E3 over the ARM A57 baseline. RoboX achieves an average improvement of 22.1× over the ARM A57 baseline, with a range of 4.5× to 65.3×. As expected, the Xeon E3 has a 0.28× lower performance-per-watt over the ARM A57 on average. Over the GPU baselines, RoboX achieves an average improvement of 65.5× over the GTX 650 Ti, with a range of 52.5 to 88.4×. The performance-per-watt improvement over the Tegra X2 is 7.8×, as the Tegra functions under a tighter power budget of 7.5 W. In comparison to the Tesla, RoboX has an average performance-per-watt improvement of 71.8×. Thus, despite the higher performance benefits of the Tesla, RoboX delivers much
higher efficiency under a limited power budget.

To summarize, these results suggest that RoboX delivers a higher performance than Xeon, GTX 650 Ti, and the Tegra X2 with an improved power efficiency, even better than the ARM. These improvements demonstrate the suitability of RoboX for high-performance under a tight power budget, which is attractive for a variety of robotics applications.

**Prediction horizon sweep.** Controller performance often improves with longer prediction horizons, but increases the amount of computation performed at each controller invocation. Figure 9 shows the speedup of RoboX over different prediction horizon lengths. On average, the speedup grows proportionally with the horizon length, from 29.4× to 38.7×. However, different benchmarks are more sensitive to larger horizons than others. For instance, the hexacopter benchmark has the greatest change in speedup for larger horizons, as it has the greatest number of penalty terms. It is also tied for greatest number of states with the Quadrotor, but the hexacopter has more average computation per state. Thus, there are more opportunities for parallelism compared to smaller models.

**Compute-enabled on-chip interconnect.** To illustrate the benefits of the compute-enabled on-chip interconnect, Figure 10 shows the speedup of RoboX with and without the interconnect ALUs for a horizon length of 1024 steps. On average, RoboX without the interconnect ALUs achieves an average speedup of 25.2×, compared with the 38.7× average speedup gained with the compute-enabled interconnect. Overall, the compute-enabled on-chip interconnect provides ~35% increase in average performance.

### C. Design Space Exploration

#### Number of compute units.

While the number of CU s for RoboX is fixed, we performed a design space exploration by varying the number of CUs across each benchmark. Figure 11 shows the speedup sensitivity to the amount of computational resources available with the ARM A57 baseline. As the amount of parallelism in the application is dependent on the prediction horizon, we explore the case where the prediction horizon is 1024 steps. Except for the MobileRobot benchmark, which has the least amount of computation, the speedup initially grows linearly with the number of CUs. Note that we double the number computational resources, starting from 1 CU to 1024 CUs. However, the benefits generally plateau around 256 CUs, as the maximum amount of parallelism in the solver computation is approached. After 256 CUs, there are diminishing returns due to the minimal change in performance and an increase in power consumption due to the presence of more resources.

**Bandwidth sensitivity.** Planning and control tasks are both compute and data intensive, as new state and environmental information, as well as the previous solution, are fetched every controller invocation. Thus, the amount of data retrieved from memory grows with the prediction horizon. While the bandwidth of RoboX is fixed, we perform a sensitivity study to evaluate the effects of bandwidth on the speedup. Figure 12 shows the speedup of RoboX over the ARM A57 baseline for a prediction horizon of 1024 across different bandwidth design points. Intuitively, larger robot models are most sensitive to the increase in bandwidth due to the increase in data. This is particularly true for the hexacopter benchmark, where its speedup varies from 46.1× to 94.3×. While all of the models benefit from increased bandwidth, there are diminishing returns up to a certain point due to this is due to the execution time becoming increasingly dominated by computation.

### IX. Related Work

**Programmable acceleration.** Programmable accelerators have received much attention due to their potential for large gains in efficiency and performance by restricting the workload. Traditional approaches to acceleration rely on identifying and mapping compute intensive kernels to specialized hardware [39]–[48]. Recent work has increasingly focused on developing accelerators for a limited set of applications, particularly machine learning and deep neural networks [11]–[19]. While these previous works have shown significant benefits for a subset of learning applications, they are not directly extensible to robot motion planning and control workloads. In RoboX, we delve into the theory of motion planning and control for robots to leverage commonalities and provide an end-to-end acceleration solution which can target a wide range of robotic applications.

**Hardware implementations for MPC.** There have been several efforts to provide hardware support for MPC algorithms. Prior ASIC designs [49, 50] for MPC do not offer the flexibility to support different robotic models and are also limited to linear...
dynamics. Furthermore, existing FPGA implementations [51]–[57] are also problem-specific, as they restrict the MPC to first-order gradient solvers or even a specific system-task pairs. Recent work uses HLS to accelerate nonlinear MPC with FPGAs [58]–[62]. In contrast, RoboX is not focused on one or a set of robotics applications and does not restrict the dynamics of the robot. RoboX, on the other hand, provides a comprehensive programmable ASIC acceleration including a novel DSL and architecture with features like the compute-enabled interconnect.

**Domain-specific languages for robotics.** Existing DSLs for expressing robot kinematics and dynamics are designed to compose simple, pre-specified primitives together [63]–[66]. However, these languages often limit themselves to specific robot types, such as multi-link manipulators, and do not provide any task-specific information to generate an actual controller. Languages which focus on task specification generate simpler control algorithms and do not support MPC or rely on pre-specific action primitives for particular types of robots [67]–[76]. Other approaches, such as ACADO [31]–[34], expose a high-level API to generate optimized C code. Instead, RoboX provides a mathematical DSL backed by a compiler and hardware architecture. This DSL does not limit the programmer to pre-specified elements and allows the expression of a wide variety of applications.

**Software parallelization for MPC.** Alternative approaches leverage algorithmic approximation techniques to enhance the parallelism of MPC [77, 78]. These purely software-based implementations deliver faster performance at the cost of control accuracy and robustness. RoboX is orthogonal to these approximation techniques and can incorporate them to provide additional benefits.

**In-network computation.** As an emerging area, recent works have explored delegating parts of execution to Network Interface Cards (NICs), routers, and switches [79]–[83]. In contrast, this paper defines the on-chip compute-enabled interconnects.

**X. CONCLUSION**

Robotics and automation have been continuously transforming a wide range of industries. As advances continue in robotics, their computational demand is increasing. As such, this work sets out to accelerate autonomous robotics by providing the cross-stack solution of RoboX. This solution abstracts away the complicated details of control theory, optimization formulation, hardware, and its micro-programming from developers, yet delivers significant performance and efficiency gains. While efficiency is crucial, wide range of applicability is vital for adoption of accelerators. As such, RoboX utilizes model predictive control to move away from traditional practices of offloading code to specialized hardware and provides an end-to-end acceleration solution that builds upon the theory of robotic control.

**ACKNOWLEDGMENTS**

We thank the anonymous reviewers for their insightful comments. We thank Baishen Huang for his contributions to the architectural design and Amir Yazdanbakhsh for his assistance in acquiring energy measurements. We also thank Hardik Sharma and Joon Kyung Kim for their feedback on the text. Jacob Sacks was in part supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, CCF#1553192, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, and gifts from Google, Microsoft, Xilinx, and Qualcomm.

**REFERENCES**
