Synthesis of Resource-Aware Robotic Systems

Andrea Censi

Abstract—This paper describes an approach to make robotic systems “resource-aware”, in the sense that they can adapt optimally to the available computation and communication resources. A resource-aware application is one where the functionality implemented depends on the resources usage, such as timing information (latency, throughput) and power consumption. Allowing this flexibility introduces a recursive constraint between application and resources usage; the resources usage is a function of the application and vice versa. Thus, the design problem consists in finding the least (most efficient) of the fixed points for the composition of those two maps. It is shown that a monotonicity property is sufficient to obtain an algorithm to compute the least fixed point in a systematic way.

I. INTRODUCTION

On my desk rests the cheapest mobile robot that one can build. It is based on a $9 motors-and-chassis kit, a $35 Raspberry PI 2, which has a 900 MHz quad-core A7 and 1 GB of RAM, a Wifi adapter, a cellphone battery, a camera, and a motor controller board, for a total cost of well below $100. Is the Raspberry PI 2 powerful enough to make a robot do interesting things? Yes, definitely. It is much more powerful than the 100 Mhz PowerPC on the computer embedded on the Curiosity rover landed on Mars in 2012. But the answer is also: definitely not. In fact, it is severely under-powered for most “out-of-the-box” implementations of perception algorithms, which are tuned to run in real time on much higher clock speeds and powerful GPUs, like the ones in my laptop.

In a few years, the PI will be as powerful as my current laptop; but it will still be true that my future laptop will be much more powerful than the PI, and much less powerful than the server down the hall. It will also be true, barring revolutions in energetics, that the energy on board of the robot will be scarce. Therefore, an idea that is invariant to technological progress is that mobile platforms should be able to offload computation when possible. For example, the humble robot above should be able to ask for help in processing the data from its camera to my laptop or the server down the hall because it will reduce its battery consumption. This idea belongs to the “Cloud Robotics” and “Internet of Things” memeplexes. Some special cases are already implemented in consumer electronics; Apple’s Siri and Microsoft’s Cortana are two examples of real-time systems that offload computation to a remote server.

This paper considers the general case for robotics: how to create robotic systems that can use distributed computation, under generic constraints on latency, throughput, bandwidth, and energy consumption. The key idea is conceiving of a robotic application at a higher level of abstraction than current practice; in particular, by considering the application separate from (“modulo”) its implementation.

We can rely on some of the abstractions developed in the field of Embedded Systems. In particular, in the “Y-chart” approach [1][2] and closely related formalisms [3][4], the designer models the “application” separately from the “architecture”, thus forming two branches of a “Y” (Fig. 2), while the configured system (“deployment”) is obtained through an optimization problem. In the case when application and architecture are both described with a graph formalism, the optimization problem consists in finding a homomorphism between the two graphs.

To obtain applications that can self-deploy optimally, it is necessary to have abstract models of application, architecture, and deployment, all described independently. Compare, for example, ROS launch files [5], which describe both the application (the modules and their interconnection) together with the deployment (which nodes are connected to which), and do not describe the architecture explicitly.

Resource awareness: This paper considers also the case in which the system is “resource-aware”, in the sense that the behavior itself is automatically tuned with respect to the computation and network resources that are available.

Example 1. Consider a robot whose goal is to map an environment at a given accuracy. The accuracy requirement on the map can be transformed into a minimum spatial...
sampling period; for example, “obtain one sensor reading every $\ell = 0.1$ meters”. Suppose that the on-board computer takes $T$ seconds to process each sensor reading. Then the velocity of the platform platform must be at most $v \leq \ell/T$ (m/s). A “resource-aware” application would be able to change the velocity $v$ as a function of $T$. For example, if the robot has a connection to remote computing resources, and if the latency/bandwidth of the network allow to achieve smaller $T$, then the robot should be able to automatically switch to remote computation, and increase its velocity $v$.

**Example 2.** The amount of computation required varies according to the complexity of the environment, such as the amount of clutter (Fig. 3a). A simple model would be a Poisson forest [6] with variable density. Requiring the performance to stay constant during the mission, for example as measured by the speed achievable by the vehicle (Fig. 3b), implies over-designing the system to account for the worst case (Fig. 3c). If one allows the performance requirements to be relaxed as a function of the environment complexity, for example by requiring the speed of the vehicle to be lowered in more cluttered parts of the environments (Fig. 3d), then the computational requirements can be lowered (Fig. 3e) and a cheaper design might be feasible.

![Fig. 3.](image)

Allowing the application to be dependent on the resources usage introduces a “strange loop” (Fig. 4a) given by the recursive constraints between application and resources. Consider the map “deploy” as the map between the space of “applications” and the space of “resources usage” that gives the least amount of resources (time, energy, . . .) that are necessary to implement the application on a given architecture:

\[
\text{deploy: application } \mapsto \text{ resources usage.}
\]

Suppose that the application is resource-aware, in the sense that there is a map “adapt” that chooses the application based on the resources:

\[
\text{adapt : resources usage } \mapsto \text{ application.}
\]

Therefore, the problem, instead of “find the best implementation of a fixed application”, becomes that of finding the pair of application $a$ and resources $r$ that are a fixed point of (adapt $\circ$ deploy) or (deploy $\circ$ adapt):

\[
a = \text{adapt(\text{deploy}(a))},
\]

\[
r = \text{deploy(\text{adapt}(r)).}
\]

The rest of this paper is devoted to making these notions precise. Section II, IV recall some traditional notions in the field of Embedded Systems:

- Section II describes resource graphs, a formal description of computation and communication architecture.
- Section III describes the formalization of the application.
- Section IV describes the resource allocation problem.

Section V describes the problems arising from having a recursive dependence between application and resource usage, for which the design problem consists in finding the fixed point of (1). A complete solution is presented for a class of application called “monotone”. For this class, there exists a simple iterative algorithm to find the most efficient fixed points or to decide that the problem is unfeasible.

## II. Representing the Architecture

This section and the following two describe notions that are well understood in the field of Embedded Systems. A recent comprehensive introduction is given by Sriram and Bhattacharyya [7]. Some variations are necessary to take into account some of the particularities of robotic applications, especially the different granularity of applications.

The models described here should be understood as plausible abstractions of the real phenomena involved in building distributed systems. In some cases, more advanced models already exist in the literature. For example, the simple “pipe” model of networks does not take into account variable and stochastic delays given by packet drops. Most of the conclusions in Section V would hold for more realistic models.

### A. Resource Graphs

A resource graph (abbreviated as “rgraph”) is a formal description of the available architecture, including computing and network resources.

1) Processor type and capacity: The information in the rgraph will be used to predict the timing of an application. A first-order model assumes that each resource node has a categorical type in a set RTypes, plus a scalar value called “capacity”. Assume that the time it takes to run a procedure depends on the resource type and is inversely proportional to the capacity. For example, the set of resource types could be RTypes = \{686, ARM\} and the capacity is the clock or number of cores. This is the simplest model that allows to express the fact that some procedures might not be supported by some processor type, and some limited generalization capability using the “capacity” values. If this is not an appropriate model, then the formalization can support arbitrary execution times of any procedure on any processor by interpreting the resource type as the ID of the processor.
2) **Network Links:** A first-order model for network links is the “pipe” model (Fig. 5), characterized by the latency $L \geq 0$, measured in seconds, and bandwidth $B \geq 0$, measured in bytes/second. Transmitting a message of length $S$ bytes takes $L + S/B$. Multiple transmissions on the same link get allocated fairly $1/n$ of the available bandwidth.

3) **Resource graphs:** A resource graph is defined as a graph in which the nodes represent the processors and the edges represent network links.

**Definition 1.** A resource graph $rG$ is a directed multigraph $(rN, rE)$ where:

- Each node $rN \in rN$ has two labels:
  - $rN$.type $\in$ RTypes, describing the categorical type.
  - $rN$.capacity $\geq 0$, describing the node’s capacity.
- Each edge $re \in rE$ has two labels:
  - $re$.bandwidth $\geq 0$ is the bandwidth of the link (bits/s);
  - $re$.latency $\geq 0$ is the latency of the link (seconds);
- $re$.capacity $> 0$ is the bandwidth (bits/s).

Furthermore, two special nodes in $rN$ are denoted as “driver-obs” and “driver-cmd”: these represent the sensor and actuator and will constrain the optimization problem.

**B. Process Networks**

The simplest model for POSIX-like processes communicating through sockets (local or remote) is the model given by ideal Khan processes [8, Chapter 2] with unbounded FIFOs, blocking read operations, and non-blocking writes. Two further assumptions are made to predict timing information:

- **Simultaneous reads:** It is possible for a process to read contemporaneously from multiple sockets; this is the functionality implemented by the `select()` system call on Unix systems.
- **Preemptive multitasking:** All processes run with the same priority on a preemptive multitasking operating system with a fair scheduler and negligible time slice with respect to the time scales involved. This means that if two jobs which separately would take an interval $w_1 > 0$ and $w_2 > w_1$ are executed in parallel, the first will finish at time $2w_1$ and the second will finish at time $w_1 + w_2$.

**III. REPRESENTING THE APPLICATION**

Call “application” the set of all algorithms and procedures that on a robot take a stream of observations and transform it to a stream of commands. It is necessary to explicit model the entire application to be implemented as a concrete object that can be manipulated. Call the space of all applications $Apps$ (Fig. 8). There are many possibilities to rigorously describe $Apps$. For example, a set of C++ source files, along with their configuration parameters, and all the libraries on which they rely, is certainly a rigorous description of an application, though it is not one that is easy to manipulate.

The term model of computation (MoC) refers to the semantics of the application specified by the user, abstracted away from the implementation details.

The simplest model of computation is the Homogeneous Synchronous Data Flow (HSDF) model [9, Section 6.3]. The application is specified as a directed multigraph, in which the nodes are called “actors”. Each actor has multiple input and output ports (Fig. 9).

This is an informal description of the semantics of execution for HSDF models:

- There is a finite number of messages stored in a buffer at each input port.
- An actor “fires” when there is a message on all input ports.
- Each firing removes 1 message from each input port, and produces 1 message for each output port that is copied to the input ports of the other connected components. (This is similar to the semantics of Petri Nets [10].)
- The output of an actor depends possibly on the entire sequence of inputs, but not on any other signal.
- There are no side effects to the firing of an actor.

In the HSDF MoC the sequence of messages is “determinate”, which means that the output sequence of an actor is entirely determined by the input sequence. As an alternative, consider the “δ-dataflow” MoC [11], an example of event-based semantics with nondeterminism.

In general, when specifying the MoC there is a trade-off of tractability vs flexibility/expressiveness: a more flexible and expressive MoC will make life (superficially) easier for the user, but it will limit the possibilities of the tool designer.

The HSDF MoC captures the semantics of many robotics applications, with the exception of the side-effect free constraint. In fact, most algorithms in robotics do have a state. Call HSDF~ the HSDF MoC without the requirement of the actors being side-effect free. The set of all application graphs defining an HSDF~ will be our space $Apps$.
IV. DEPLOYING AN APPLICATION ON A DISTRIBUTED ARCHITECTURE

Suppose an application app ∈ Apps and a resource graph \( rG \) ∈ \( RG \) are given, and we wish to find an implementation of the application app as a process network on the architecture described by the given resource graph.

A basic pipeline includes the following steps:

1) **Benchmarking.** The application app is benchmarked to obtain a “computation graph” that indicates what are the computation requirements for each actor and the size of each signal exchanged between actors.

2) **Optimization:** The optimization problem is divided into two subproblems:

   a) **Resource allocation.** This is formulated as the search for a homomorphism between the computation graph and the resource graph.

   b) **Scheduling.** Scheduling refers to the arbitration of the order of executions of the actors on each processor.

3) **Evaluation.** Once a solution has been synthesized, it can be evaluated according to different measures, such as the latency, throughput, and power utilization.

A. Computation Graphs and Benchmarking

A “computation graph” is an abstract representation of the computation and communication requirements of an application\(^1\).

In the case of Homogenous Static Data Flow, the computation graph is isomorphic to the application graph. Each node in a computation graph represents the execution of an actor. Each edge in the computation graph represents a signal between two actors.

**Definition 2.** A computation graph \( cG \) is a directed weakly-connected acyclic labeled multi graph \((cN, cE)\) such that:

- For each node \( cn \in cN \), the label \( cn.ops : RTypes \rightarrow \mathbb{R}_0 \) is a function such that \( cn.ops(rn.type) \) describes the time necessary to carry out the computation on a resource of type \( rn.type \) and capacity \( rn.capacity = 1 \).

- For each edge \( ce \in cE \), the label \( ce.size \) describes the size (in bytes) of the signal flowing between two nodes.

Two special nodes in \( cN \) are marked as “input” and “output”.

![Fig. 10. Each node \( cn \) of a computation graph has a label \( cn.ops \) describing the computation time as a function of a processor, and each edge \( ce \) has a label \( ce.size \) describing the size in bytes of the signal. Computation graphs can, in principle, be obtained from static analysis. In robotics software components are complicated and so it is not plausible to use static analysis.](image1)

The alternative is benchmarking. Benchmarking is a map from the space of applications to the space of computation graphs:

\[
\text{benchmark} : \text{Apps} \rightarrow \text{CG}.
\]

B. Optimization

1) **Resource allocation:** The resource allocation problem consists in choosing which actors are run on each processor, and which network edges are used to communicate which signals.

   This problem can be formalized as the search of a homomorphism between the two graphs \( cG \) and \( rG \).

   This is the usual definition of homomorphism given for directed graphs\(^2\) Section 1.4:

   **Definition 3** (Directed graph homomorphism). Let \( G = (V, E) \) and \( G' = (V', E') \) be two directed graphs. A graph homomorphism \( h \) between \( G \) and \( G' \) is a pair of mappings \( h = (h_V, h_E) \) such that \( h_V : V \rightarrow V' \) and \( h_E : E \rightarrow E' \) such that for each edge \( (u, v) \in E \), the transformed edge \( h_E((u, v)) = (h_V(u), h_V(v)) \) is in \( E' \).

   The map \( h_V \) associates each computation node \( cn \in cN \) to the processor \( rn = h_V(cn) \in rN \) to which it is assigned. Similarly, for each signal \( ce \in cE \), \( re = h_E(ce) \) is the corresponding network link that transports the signal.

   For example, in Fig. 12 the homomorphism indicates that \( cn_1 \) is executed on \( rn_1 \), and that the physical link \( re_1 \) is unused.

   ![Fig. 12.](image2)

Let \( \text{Hom}(cG, rG) \) be the set of possible homomorphisms between the two graphs \( cG \) and \( rG \), taking into account also the additional constraint that the special nodes \( \text{input}, \text{output} \in cN \) match with the corresponding nodes \( \text{driver-cmd}, \text{driver-obs} \in rN \):

\[
h_V(\text{input}) = \text{driver-obs},
\]

\[
h_V(\text{output}) = \text{driver-cmd}.
\]

Optimizing over homomorphisms is, in general, a combinatorial problem. Even recent papers use heuristic techniques such as evolutionary computation\(^\text{[15]}\).

2) **Scheduling:** Once a homomorphism \( h \) has been chosen, what remains is a scheduling problem. This is an entire field in itself. With the assumption of preemptive multitasking, we can assume that each actor is scheduled as an independent
process that starts computing at the moment that all its inputs are ready. Note that it is well known that multitasking is not optimal: if two jobs can be executed in parallel, it is better to finish first the one that is in the critical path \[16\]. There are no further choices to make, and so the schedule is entirely “self-timed” and entirely determined by the homomorphism \( h \).

Still, it is necessary to create a schedule to evaluate latency and throughput. The main steps involved are the construction of a job graph, and the construction of a Gantt chart \[7\;\text{Chapter 5}\].

C. Evaluation

There are many axes along which to evaluate a solution:

- **Latency**, or makespan (measured in seconds): this is the time that it takes to compute a command from the last observation received. Control applications are typically very sensitive to latency.
- **Throughput**, or sampling interval (measured in seconds): This is defined as the minimum period between successive repetitions of the schedule. (For a camera, the inverse of the throughput is the frame rate.) Throughput and latency are completely independent.
- **Processor and link utilization** (measured as %)
- **Power consumption** (Watts): Processor and link utilization can be converted to power consumption, using a model of the hardware or direct measurement.
- **Operating cost ($/s)**: In the case of cloud computing, this is a quite literal $/hour figure (for Amazon, it is on the order of $0.5/hour for a 16 core virtual machine). For mobile robots, a consistent part of the operating cost is the replacement of batteries. This is a monotone function of the onboard power consumption.

Call \( \mathcal{R} \) the product space of the performance measures of interest. The mnemonics for “\( \mathcal{R} \) is “resources”.

Suppose there is a partial order \( \preceq_{\mathcal{R}} \) that expresses an arbitrary preference, with the convention that “less is better”, so that if \( r_1 \preceq r_2 \), one prefers \( r_1 \) to \( r_2 \).

Let \( \text{eval} : \mathcal{C}G \times \mathcal{R}G \times \text{Hom} \rightarrow \mathcal{R} \) be the function that assigns to each computation graph \( \mathcal{C}G \), resource graph \( \mathcal{R}G \), and choice of homomorphism \( h \in \text{Hom}(\mathcal{C}G, \mathcal{R}G) \), its resource consumption \( \text{eval}(\mathcal{C}G, \mathcal{R}G, h) \).

Let \( \mathcal{P}(\mathcal{R}) \) be the set of all subsets of \( \mathcal{R} \).

Let \( \min_{\leq_{\mathcal{R}}} : \mathcal{P}(\mathcal{R}) \rightarrow \mathcal{P}(\mathcal{R}) \) be the function that to each subset the set of minimal (non dominated) elements.

We can abstract the optimization problem as a function \( \text{deploy}_{\mathcal{C}G} \) that assigns to each computation graph the minimal set of resources that are needed to implement the corresponding application on \( \mathcal{R}G \): (Fig. 13)

\[
\text{deploy}_{\mathcal{C}G} : \mathcal{C}G \rightarrow \mathcal{P}(\mathcal{R}),
\]

\[
cG \rightarrow \min_{\leq_{\mathcal{R}}} \{\text{eval}(\mathcal{C}G, \mathcal{R}G, h) \mid h \in \text{Hom}(\mathcal{C}G, \mathcal{R}G)\}.
\]

**Example 3.** Any nontrivial case has multiple solutions that do not dominate each other. One of the trade-offs interesting for robotics is the latency vs throughput trade-off, which arises even in the simplest applications.

Consider the computation graph in Fig. 14 which consists of three actors, each taking 0.25s to execute.

![Fig. 14.](image)

Suppose the target resource graph is the one shown in Fig. 15 with 2 CPUs of the same type with a fast link between them.

![Fig. 15.](image)

Excluding the special input/output nodes, there are 2 processors and 3 actors, and \( 2^3 = 8 \) valid homomorphisms between the two graphs. Fig. 16 shows the trade-off space \( \mathcal{R} \), in this case including only latency and throughput. Of the 8 solutions, 2 are “minimal”, in the sense that they are not dominated by any other. The minimal latency solution is the one that places all actors on the same processor (Fig. 17a). The minimal throughput solution is the one that distributes the load between the two processors, so that the multiple actors are executed in parallel (Fig. 17b).

![Fig. 16.](image)

![Fig. 17.](image)
V. RESOURCE-AWARE APPLICATIONS

The previous section has described how to optimally deploy an application on a given architecture. Given an application \( app \in \text{Apps} \), the corresponding computation graph \( cG \in \text{CG} \) describes the computation and communication requirements to implement the application. Benchmarking is a map from application to a computation graph:

\[
\text{benchmark} : \text{Apps} \rightarrow \text{CG}.
\]

Given a computation graph \( cG \) and a target resource graph \( rG \), the resource allocation problem is finding a homomorphism of \( cG \) to \( rG \) that describes which processors compute which component and which network edge carries which signal. There are usually multiple solutions that can be evaluated in terms of measures such as latency, throughput, energy, and bandwidth consumption. Defining the poset of “resources” \((\mathcal{R}, \leq)\), the optimization process in Section IV can be abstracted as a function from the set of computation graphs to the set of least resource usage:

\[
\text{deploy}_{CG} : \text{CG} \rightarrow \mathcal{P}(\mathcal{R}).
\]

This section introduces the idea of having the application be “resource-aware” by considering a map from the set of resources to the set of applications.

**Definition 4.** A resource-aware application (RAA) is a map \( \Psi : \mathcal{R} \rightarrow \text{Apps} \).

A RAA \( \Psi \) introduces a circular constraint between application and resources (Fig. 18).

![Fig. 18.](image)

**Example 4.** Consider a robot that is visually guided using images from a camera (Fig. 19). Divide the control systems in two blocks: a resample operation that resamples the images to a given resolution \( \rho \in \mathbb{R}_+^\circ \) (pixels/steradians) and a “process” block that lumps together every other function, such as perception, planning, and control.

![Fig. 19.](image)

Suppose that the application is executed (possibly on a distributed infrastructure) with latency \( L \) and throughput \( \Delta \). Consider a performance metric of the form \( J(\rho, L, \Delta) \), such as the precision in following a trajectory. Suppose the goal is to bound \( J \leq J_{\text{max}} \). Define the function \( \rho_{\text{min}}(L, \Delta) \) as the minimum resolution needed to have satisfactory performance:

\[
\rho_{\text{min}}(L, \Delta) = \min_\rho \{ \rho \in \mathbb{R} : J(\rho, L, \Delta) \leq J_{\text{max}} \}.
\]

By convention, set \( \min \emptyset = \infty \) (“the minimum resolution is infinite”). The map \( \Psi \) is the one that associates to each value of the latency \( L \) the application that resamples the image to the minimum resolution:

\[
\Psi : \mathbb{R}_+^\times \mathbb{R}_+ \rightarrow \text{Apps},
\]

\[
L, \Delta \mapsto \begin{array}{c}
\text{resample}_{\rho_{\text{min}}(L, \Delta)} \\
\text{process}
\end{array}.
\]

A. Synthesis of resource-aware applications

Define the set of “feasible operating points” as the subset of \( \text{Apps} \times \mathcal{R} \) that respects the circular constraint.

**Definition 5.** A feasible operating point (FOP) for a RAA \( \Psi \) on a resource graph \( rG \) is a pair \( (app_o, r_o) \in \text{Apps} \times \mathcal{R} \) such that

\[
\text{deploy}_{CG}(\text{benchmark}(app_o)) \leq_{\mathcal{R}} r_o,
\]

\[
\Psi(r_o) = app_o.
\]

At this point we can define the decision and synthesis problem for RAAs.

**Problem 1 (Synthesis of Resource-Aware Applications).**

Given a RAA \( \Psi \) and a resource graph \( rG \):

1) Is there a nonempty set of FOPs?
2) If a nonempty set of FOPs exist, what are the one(s) that correspond to the least usage of resources?

B. Solution for monotone applications

A comprehensive solution to Problem 1 can be obtained for a special class of RAAs, using the framework of monotone co-design described in previous work [17]. More specifically, if the maps \( \Psi, \text{benchmark}, \text{deploy}_{CG} \) that constitute the cycle satisfy some monotonicity properties, then there is an efficient way to find the least FOP or show that no FOP exists.

1) Monotonicity on posets: To state the monotonicity properties we need to define monotonicity for posets.

**Definition 6.** A map \( f : A \rightarrow B \) from poset \((A, \leq_A)\) to \((B, \leq_B)\) is monotone iff \( a_1 \leq_A a_2 \) implies \( f(a_1) \leq_B f(a_2) \).

2) Trade-off space: Given a poset \((P, \leq)\) consider the set of subsets of \( P \) whose elements do not dominate each other. In the literature these subsets are called “antichains”, but a better name is the “trade-off space”.

**Definition 7.** For a poset \((P, \leq)\), the trade-off space \( \mathcal{T}(P) \) is the set of equivalence classes \( \mathcal{T}(P) \equiv P/_{\text{min}_\leq} \).

**Lemma 1.** The trade-off space \( \mathcal{T}(P) \) is a complete partial order for the relation \( \leq_T \) defined as \((A \leq_T B) \equiv (A \leq_B \land (B \leq_A) \land (A \leq B)) \equiv (\forall a \in B : \exists a \in A : a \leq b) \).

3) Poset of computation graphs: The space of computation graphs \( \text{CG} \) can be made a poset with the relation \( \leq_{\text{CG}} \). Intuitively, \( cG_1 \leq_{\text{CG}} cG_2 \) if \( cG_1 \) requires less computation or less communication.

**Definition 8.** The set \( \text{CG} \) is a lattice for the relation \( \leq_{\text{CG}} \) as follows: \( cG_1 \leq_{\text{CG}} cG_2 \) iff:

1) There exists a homomorphism \( \varphi : cG_1 \rightarrow cG_2 \).
2) For all \( cN_1 \in cN_1 : \text{cn}_1, \text{ops} \leq \varphi_N(\text{cn}_1), \text{ops} \).
3) For all \( ce_1 \in cE_1 : \text{ce}_1, \text{size} \leq \varphi_E(\text{ce}_1), \text{size} \).

4) Monotone Resource-Aware Applications: The monotonicity property that a RAA \( \Psi \) must satisfy is given by Def. 8 below. Intuitively, a RAA is monotone if an increase in
resources usage corresponds to more computation or communication needed.

**Definition 9** (Monotone RAAs). A RAA $\Psi$ is monotone if the composition $(\text{benchmark} \circ \Psi) : R \rightarrow CG$ is a monotone map between the two posets $(R, \leq_R)$ and $(CG, \leq_{CG})$

Note that the definition does not use any partial order on the space Apps directly; so most of this reasoning could be generalized to different model of computations.

**Example 4** (continued). In this example, the resource space $R$ describes latency $L$ and throughput $\rho$. For a large class of tasks, such as trajectory tracking, $J(\rho, L, \Delta)$ can be assumed to be nonincreasing in $\rho$ (a higher resolution gives lower error) and nondecreasing in $L$ and $\Delta$ (higher latency and lower sampling give larger error). Thus the function $\rho_{min}(L, \Delta)$ is an increasing function of $L, \Delta$. The RAA $\Psi$ is monotone in the sense of Def.9 if it holds that if the image resolution is increased, then the computation and communication needed do not decrease.

If the property of monotonicity holds, then there exists a simple way to obtain the most efficient FOP.

**Proposition 1.** For a monotone (Def.9) RAA $\Psi$, the minimal elements of the set of FOPs (Def.5) can be found by constructing an increasing sequence in $\mathcal{T}(R)$ starting from $T_0 = \{ \bot \}$ and iterating $T_{k+1} = F(T_k)$, with the map $F$ given by

$$F : \mathcal{T}(R) \rightarrow \mathcal{T}(R),$$

$$T \mapsto \min_{\preceq_R} \bigcup_{r \in T} \text{deploy}_{\mathcal{CG}}(\text{benchmark}(\Psi(r))).$$

If the sequence $\{T_0, T_1, \ldots \}$ converges to a finite set of values $T_\infty$, then its elements are the minimal FOPs. If the sequence converges to $\top$, then there are no FOPs.

The interpretation of $F$ in equation 5 is the following. For each of the points $r \in T$, compute the corresponding application $a = \Psi(r)$. Benchmark the application to obtain the computation graph $CG = \text{benchmark}(\Psi(r))$. Then find the possible solutions $\text{deploy}_{\mathcal{CG}}(CG) \subset R$. Take the minimum over all possible alternatives, and then iterate. If the sequence converges to a subset of $R$, that is the subset of the minimal FOPs. If the sequence does not converge, then it is a certificate for the non-existence of FOPs.

**Proof.** First, we show that the map $\text{deploy} : CG \rightarrow \mathcal{P}(R)$ is monotone. Consider the map $\text{eval} : CG \times \mathcal{RG} \times \text{Hom} \rightarrow R$ that associates to each tuple (computation graph, resource graph, homomorphism) to the resources used. Consider the map $\text{eval}'$ obtained by fixing $\mathcal{RG}$ and $\varphi$ in $\text{eval}:

$$\text{eval}' : CG \rightarrow R,$$

$$CG \mapsto \text{eval}(\mathcal{RG}, CG, \varphi).$$

Then we can prove that $\text{eval}'$ is monotone. Take $CG_a \leq CG_b$, with the relation $\leq_{CG}$ defined in Def.8. This means that $CG_a$ and $CG_b$ are isomorphic, and that each operation in $CG_b$ takes less computation time, and each signal takes less size. We can take into account computation time and transmission time separately by constructing a chain

$$CG_a = CG_1 \leq CG_2 \leq \cdots \leq CG_n = CG_b,$$

in which each graph differs from the previous one only by the computation time of one node or the size of one signal. If we show that $\text{eval}'(CG_i) \leq \text{eval}'(CG_{i+1})$, then it follows that $\text{eval}'(CG_a) \leq \text{eval}'(CG_b)$. Consider now the case where $CG_a$ and $CG_{i+1}$ differ by only computation time of one node. The latency, throughput and cpu usage will not decrease from $CG_i$ to $CG_{i+1}$, so that $\text{eval}'(CG_i) \leq \text{eval}'(CG_{i+1})$. (It is not true that there will necessarily be an increase in latency and throughput, because the node might not be on the critical path.) Equivalently, an increase in the size of the signals will not decrease latency, throughput or energy consumption. Thus $\text{eval}'$ is monotone.

The map $\text{deploy}_{\mathcal{CG}}$, defined in 2, is the minima elements of $\text{eval}$; so if $\text{eval}'$ is monotone, then also $\text{deploy}_{\mathcal{CG}}$ is monotone.

Consequently we have concluded that there are:

- Two complete partial orders $(CG, \leq_{CG})$ and $(\mathcal{T}(R), \leq_T)$
- A map $\text{deploy} : CG \rightarrow \mathcal{T}(R)$ that is monotone.
- A map $(\text{benchmark} \circ \Psi) : R \rightarrow CG$ that is monotone.

This is a monotone co-design problem in the sense of Definition 11 in [17]. The conclusions then follow from Proposition 1 in [17]. Reproducing the main argument here, consider the map $h : \mathcal{T}(R) \rightarrow \mathcal{T}(R)$ defined as the composition $h = \text{deploy} \circ \text{benchmark} \circ \Psi$. This map is a monotone endofunction on a poset, which implies by Kleene’s theorem [18] that a least fixed point always exists, and it can be found by constructing the ascending Kleene chain starting from the bottom element of poset. $\square$

**Example 4** (continued). If we consider only the latency $L$ (and ignore the throughput), then the problem becomes scalar and the solution can be discussed graphically. The control objectives induce the function $\rho_{min}(L)$, which is an increasing function of $L$ (Fig.20a). The resource graph induces a function $L_{min}(\rho)$, which is increasing (Fig.20b). Together, they define the set of FOPs; which might be nonempty, as in Fig.21a or could be empty, if the two functions do not intersect.

In this case the iteration 5 is simply an iteration over $L$, defined as:

$$F : L \mapsto L_{min}(\rho_{min}(L)).$$

Whether the sequence $\{0, F(0), F(F(0)), \ldots\}$ converges to a fixed point or not depends on whether the set of FOPs is empty. If there exists a FOP then the iteration will converge to the least one (Fig.21a). If there are no FOPs, then the iteration will diverge (Fig.21b).
VI. DISCUSSION

This paper described an approach to creating “resource-aware” robotic systems that are able to optimally use distributed computation resources and adapt their behavior according to the resources available.

To obtain these abilities, it is necessary to think of the “application” abstracted away from the “architecture”. This idea was very fruitful in the Embedded Systems community. Both application and architecture can be described in terms of graphs: a graph that describes the computation and communication needed, here called “computation graph” (Def. [2]), and a graph that describes processors and network links available, here called “resource graph” (Def. [1]). The resource allocation problem consists in finding a homomorphism between the two graphs, and it can be completely automated. These ideas should be adopted in robotics, as they make systems both more efficient and easier to develop.

In addition, in robotics there is an additional level of irreducible complexity, in creating what were called here “resource-aware applications” (Def. [1]). The irreducible complexity consists in the fact that, if the application depends on the resource usage, and the resource usage depends on the application, then the design problem is not only finding the least resources to implement a given application, but rather finding a “feasible operating point” (Def. [3]). A tuple (application, resources) that is a fixed point of the circular constraints.

This paper studied a class of resource-aware applications that have certain monotonicity properties (Def. [9]). For these, there exists a simple algorithm to obtain the least feasible operating point, or a certificate that none exists (Proposition [1]).

In general, these resource-awareness functionalities are common to many robotic scenarios, but also orthogonal to each specific application; thus these functionalities are part of what we could call a “robot operating system”. Just like an “operating system”, in the traditional sense of the word, arbitrates the access of “processes” to “resources” (memory, CPU), a “robot operating system” should be able to arbitrate the access of a “robotic application” to distributed “resources”, which now include also quantities such as bandwidth, energy, and budget for cloud computing. In addition, a “robot operating system” would be the right level to formalize automated mechanisms for degrading/upgrading the behavior of the system based on the resources available.

REFERENCES


