Robotics: Science and Systems 2016
Ann Arbor, MI, USA, June 18-22, 2016

Task Variant Allocation in Distributed Robotics

José Cano*, David R. White†, Alejandro Bordallo*,
Ciaran McCreesh†, Patrick Prosser†, Jeremy Singer† and Vijay Nagarajan*
*School of Informatics, University of Edinburgh, UK
†School of Computing Science, University of Glasgow, UK

***Abstract*—We consider the problem of assigning software processes (or tasks) to hardware processors in distributed robotics environments. We introduce the notion of a *task variant*, which supports the adaptation of software to specific hardware configurations. Task variants facilitate the trade-off of functional quality versus the requisite capacity and type of target execution processors. We formalise the problem of assigning task variants to processors as a mathematical model that incorporates typical constraints found in robotics applications; the model is a constrained form of a multi-objective, multi-dimensional, multiple-choice knapsack problem. We propose and evaluate three different solution methods to the problem: constraint programming, a constructive greedy heuristic and a local search metaheuristic. Furthermore, we demonstrate the use of task variants in a real instance of a distributed interactive multi-agent navigation system, showing that our best solution method (constraint programming) improves the system's quality of service, as compared to the local search metaheuristic, the greedy heuristic and a randomised solution, by an average of 16%, 41% and 56% respectively.**

I. Introduction

Modern robotics systems are increasingly distributed, heterogeneous and collaborative, incorporating multiple independent agents that communicate via message passing and distributed protocols. A distributed approach can offer desirable qualities such as improved performance. Heterogeneity refers to the type and amount of hardware resources (e.g. sensors, CPU capacity) available on each agent in the system. In such systems, the efficient allocation of software processes (referred to as *tasks*) to hardware processors is of paramount importance in ensuring optimality. Previous works [13, 14] generally take an approach that considers only a fixed set of tasks, equivalent to a "one size fits all" architecture, limiting the ability of a system to adapt to different hardware configurations, and reducing the opportunities for optimisation.

Instead, we advocate the development of systems based on the selection and allocation of what we term "task variants". Task variants are interchangeable software components that offer configurable levels of quality of service (QoS) with a corresponding difference in the amount and/or type of computing resources they demand; such variants naturally arise in many scenarios, and often deployed systems consist of a particular subset of variants that have been implicitly chosen by a system architect. For example, consider alternative feature detection algorithms to solve a common task in a robotics vision pipeline: different algorithms provide increasingly sophisticated recognition methods but at the cost of increasing CPU load. Similarly, a variant may offer accelerated processing by targeting specialised hardware (e.g. GPUs).

Currently, the crucial step of selecting and allocating such task variants is typically performed using ad-hoc methods, which provide no guarantee of optimality and may thus lead to inefficient allocation. In this paper, we take a more systematic approach. We formalise the task variant allocation problem and propose three different solution methods that are able to efficiently exploit available resources with the objective of maximising QoS while ensuring system correctness.

We focus on distributed heterogeneous robotics systems where variants are naturally available for several tasks. In particular, our work has been driven by a case study, in the form of a distributed system of agents running on ROS [24]. The application implements a framework for inferring and planning with respect to the movement of goal-oriented agents in an interactive multi-agent setup — full details can be found in [4]. There are two types of agents navigating in the same physical space: autonomous robots represented by KUKA youBots [3] and humans. Each agent is pursuing a goal (a specific spatial position in the scenario) while avoiding collisions with other agents, based on online sensor processing and beliefs concerning the latent goals of other agents.

Specific tasks are used to accomplish this objective in a distributed fashion. For example, robots infer navigation goals of other agents from network camera feeds, provided by at least one *Tracker* task — meanwhile humans act independently and are assumed to navigate as a rational goal-oriented agent through the space. Some tasks can be configured via parameter values (e.g. the camera frame rate for the *Tracker* task) that translate into variants for that task. Each of these variants produces a different level of QoS, which we assume is quantified by an expert system user. Thus, the objective is to select task variants and allocate them to processors so as to maximise the overall QoS while agents reach their goals.

The contributions of the paper are as follows: i) we introduce a mathematical model that represents the task variant selection and allocation problem; ii) we propose three different solution methods (constraint programming, local search metaheuristic, greedy heuristic) to the problem; iii) we evaluate and compare the solution methods through simulation; iv) we validate the solution methods in a real-world interactive multi-agent navigation system, showing how our best solution method (constraint programming) clearly outperforms the average QoS of the local search metaheuristic by 16%, the greedy heuristic by 41%, and a random allocation by 56%. To the best of our knowledge, we are the first to address task allocation in the presence of variants in distributed robotics.

II. Problem Formulation

We now model the problem of task variant allocation in distributed robotics, in a general formulation that also applies to the specifics of our case study. We consider allocation as a constrained type of multi-objective, multi-dimensional, multiple-choice knapsack problem. Whilst instances of these three problems are individually common in the literature [11, 18], the combination is not. In addition, we allow for a number of unusual constraints describing task variants that distinguish this formulation from previous work (e.g. the specific type of hardware required to run a variant).

Our formulation of the problem divides cleanly into three parts: the *software* architecture of the system, including information about task variants; the *hardware* configuration that is being targeted as a deployment platform; and the constraints and goals of task *selection and allocation*, which may be augmented by a system architect.

A. Software Model

A software architecture is defined by a directed graph of tasks, $(T,\ M)$ where the set of tasks $T = \{\tau_1 \dots \tau_n\}$ and each task τ_i is a unit of abstract functionality that must be performed by the system. Tasks communicate through message-passing: edges $m_{i,j} = (\tau_i,\ \tau_j) \in M \subseteq T \times T$ are weighted by the 'size' of the corresponding message type, defined by a function $S : m_{i,j} \to \mathbb{N}$; this is an abstract measure of the bandwidth required between two tasks to communicate effectively.

Tasks are fulfilled by one or more *task variants*. Each task must have at least one variant. Different variants of the same task reflect different trade-offs between resource requirements and the QoS provided. Thus a task τ_i is denoted as set of indexed variants: $\tau_i = \{v_1^i \dots v_n^i\}$. For convenience, we define $V = \cup_i \tau_i$, such that V is the set of all variants across all tasks. For simplicity, we make the conservative assumption that the maximum message size for a task τ_i is the same across all variants v_j^i of that task, and we use this maximum value when calculating bandwidth usage for any task variant.

A given task variant v_j^i is characterised by its processor utilisation and the QoS it provides, represented by the functions $U, Q : v_j^i \to \mathbb{N}$. The utilisation of all task variants is expressed normalised to a 'standard' processor; the capacity of all processors is similarly expressed. QoS values can be manually (Section V-A) or automatically generated (future work), although this is orthogonal to the problem addressed.

B. Hardware Model

The deployment hardware for a specific system is modelled as an undirected graph of processors, $(P,\ L)$ where the set of processors $P = \{p_1 \dots\ p_n\}$ and each processor p_k has a given processing capacity defined by a function $D : p_k \to \mathbb{N}$. A bidirectional network link between two processors p_k and p_m is defined as $l_{k,m} = (p_k, p_m) \in L \subseteq P \times P$, so that each link between processors will support one or more message-passing edges between tasks. The capacity of a link is given by its maximum bandwidth and is defined by a function $B : l_{k,m} \to \mathbb{N}$. If in a particular system instance multiple processors share a single network link, we rely on the system architect responsible for specifying the problem to partition network resources between processors, such as simply dividing it equally between processor pairs.

C. Selection and Allocation Problem

The problem hence is to find a partial function $A : V \to P$, that is, an assignment of task variants to processors that satisfies the system constraints (i.e. a *feasible* solution), whilst maximising the QoS across all tasks, and also maximising efficiency (i.e. minimising the average processor utilisation) across all processors. As A is a partial function, we must check for domain membership of each task variant, represented as $dom(A)$, to determine which variants are allocated.

We assume that if a processor is not overloaded then each task running on the processor is able to complete its function in a timely manner, hence we defer the detailed scheduling policy to the designer of a particular system.

An optimal allocation of task variants, A^*, must maximise the arithmetic mean of QoS across all tasks (the global QoS):

$$max \quad 1/n_{tasks} \sum_{v_j^i \in dom(A)} Q(v_j^i) \tag{1}$$

Whilst minimising the average utilisation across all processors *as a secondary goal*:

$$min \quad 1/n_{proc} \sum_{p_k \in P} \sum_{v_j^i \in dom(A):\ A(v_j^i)=p_k} U(v_j^i) \tag{2}$$

Exactly one variant of each task must be allocated:

$$\forall \tau_i \in T,\ \forall v_j^i, v_k^i \in \tau_i : \\ (v_j^i \in dom(A) \wedge v_k^i \in dom(A)) \implies j = k \tag{3}$$

The capacity of any processor must not be exceeded:

$$\forall p_k \in P : \Big(\sum_{v_j^i \in dom(A):\ A(v_j^i)=p_k} U(v_j^i) \Big) \leq D(p_k) \tag{4}$$

The bandwidth of any network link must not be exceeded:

$$\forall l_{q,r} \in L : \Big(\sum_{i:A(v_j^i)=p_q} \sum_{k:A(v_l^k)=p_r} S(m_{i,k}) + S(m_{k,i}) \Big) \leq B(l_{q,r}) \tag{5}$$

In addition, *residence constraints* restrict the particular processors to which a given task variant v_j^i may be allocated, to a subset $R_j^i \subseteq P$. This is desirable, for example, when requisite sensors are located on a given robot, or because specialised hardware such as a GPU is used by the variant:

$$v_j^i \in dom(A) \implies A(v_j^i) = p_k \in R_j^i \tag{6}$$

Coresidence constraints limit any assignment such that the selected variants for two given tasks must always reside on the same processor. In practice, this may be because the latency of a network connection is not tolerable. The set of coresidence constraints is a set of pairs (τ_i, τ_k) for which:

$$\forall v_j^i \in \tau_i, \forall v_l^k \in \tau_k :\ (v_j^i \in dom(A)\ \wedge\ v_l^k \in dom(A)) \\ \implies A(v_j^i) = A(v_l^k) \tag{7}$$

III. Solution Methods

We now propose and describe our three different centralised approaches to solving the problem of task variant allocation: constraint programming (CP), a greedy heuristic (GH), and local search metaheuristic (LS). These are three broadly representative search techniques from diverse families of solution methods, as outlined by Gulwani [9].

A. Constraint Programming

We expressed the problem in MiniZinc 2.0.11 [21], a declarative optimisation modelling language for constraint programming. A MiniZinc model is described in terms of variables, constraints, and an objective. Our model has a variable for each variant, stating the processor it is to be assigned to; since we are constructing a partial mapping, we add a special processor to signify an unassigned variant. Matrices are used to represent the bandwidth of the network and the sizes of messages exchanged between tasks. The model along with the source code can be found online [26].

Most constraints are a direct translation of those in Section II-C although the constraint given by Equation 3 is expressed by saying that the sum of the variants allocated to any given task is one — this natural mapping is why we selected MiniZinc, rather than (for example) encoding to mixed integer programming. The development of a model that allows MiniZinc to search efficiently is key to its success, and we spent some time refining our approach to reduce solution time.

There are two objectives to be optimised, and we achieve this by implementing a two-pass method: first the QoS objective is maximised, we parse the results, and then MiniZinc is re-executed after encoding the found optimal value as a hard constraint whilst attempting to minimise processor utilisation.

The full model is too large to list here, but to give a flavour, we show our variables, a constraint, and the first objective:

```
array[1..nVariants] of
  var 0..nProcessors: assignments;

constraint forall (p in 1..nProcessors) (
    sum([if assignments[v] == p
         then utilisations[v]
         else 0
         endif
        | v in 1..nVariants])
      <= capacities[p]);

solve maximize sum(
  [if assignments[v] != 0
   then qos[v]
   else 0
   endif
   | v in 1..nVariants]);
```

MiniZinc allows instance data to be separated from the model. Part of a data file looks like this:

```
nProcessors = 3;
capacities  = [ 100, 100, 223 ];
links       = [|    -1, 17745, 17676
               | 17745,    -1, 17929
               | 17676, 17929,    -1 |];
```

To solve instances, we used the *Gecode* [7] constraint programming toolkit, which combines backtracking search with specialised inference algorithms. We used the default search rules, and only employ standard toolkit constraints.

It addition to being used as an exact solver, *Gecode* can also run in *anytime* fashion, such that it reports the best solution found so far. Our system reports both the increasingly better solutions produced during the run and any globally optimal result, where found. In our evaluation we consider both the standard mode, which returns the global optimum after an unrestricted runtime (Section V-C), and also this anytime mode that returns the best result found so far (Section V-E).

B. Greedy Heuristic

Our second solution method is a non-exact greedy algorithm that uses a heuristic developed from an algorithm originally designed for solving a much simpler allocation problem [5]. The procedure is described in Algorithm 1, and attempts to obey constraints, then allocate the most CPU intensive tasks possible to those processors with the greatest capacity.

Algorithm 1 Greedy Heuristic

1: P_{max} = sort $processors$ by max capacity
2: T_{max} = sort $tasks$ by max variant size
 # Allocate variants with residency constraints
3: **for** $task$ in T_{max} **do**
4: V_{min} = sort $variants$ of $task$ by min variant size
5: **for** $variant$ in V_{min} **do**
6: **if** $variant$ has residency constraints AND $task$ has no $variant$ assigned **then**
7: Allocate $variant$ to $processor$ from $R^{task}_{variant}$
 # Allocate variants with coresidency constraints
8: **for** $task$ in T_{max} **do**
9: **if** $task$ has coresidency constraints AND $task$ has no $variant$ assigned **then**
10: Allocate smallest $variant$ to $processor$ from P_{max}
 # Allocate remaining variants
11: **for** $task$ in T_{max} **do**
12: **if** $task$ has no $variant$ assigned **then**
13: Allocate smallest $variant$ to $processor$ from P_{max}
 # Upgrade variants where possible
14: **for** $task$ in T_{max} **do**
15: **if** sufficient $capacity$ in assigned $processor$ **then**
16: Allocate larger $variant$ of $task$
17: **if** all $tasks$ assigned **then**
18: **return** $allocation$

First, the smallest task variants with residency constraints are allocated to processors, beginning with the largest processor if the subset R^i_j for a given task variant v^i_j contains more than one element. Next, the smallest variants of any tasks with coresidency constraints are assigned selecting processors from P_{max}. Then, the smallest variants of any remaining, unallocated, tasks are allocated, again preferring processors with more capacity. Finally, the algorithm attempts to substitute smaller variants with larger ones on the same processor. Note

that the way in which the next processor (from R^i_j, P_{max}) or variant is selected must also ensure that allocations will not result in a violation of any previously satisfied constraints.

Also note that the greedy heuristic is not guaranteed to find a solution, but if it finds one it is always feasible, i.e. satisfies the system constraints. The ability to provide solutions is greatly determined by any residency and coresidency constraints.

C. Local Search Metaheuristic

The third algorithm we propose is a simple local search metaheuristic employing random restarts. The process is described by Algorithm 2. Initially, a random assignment is generated by allocating a random variant for each task to a random processor, and all choices are made uniformly random. There is no guarantee a randomly generated allocation will satisfy the constraints of the model, and indeed the search algorithm is not guaranteed to find a feasible solution in general. As there is no way to determine if the global optimum has been found, the algorithm continues to search the space of assignments until a given `timeout` is reached. The search may find a local optimum, in which case a random restart is used to explore other parts of the search space (lines 6-7).

Algorithm 2 Local Search Metaheuristic

```
1: current ← random assignment
2: while time < timeout do
3:     for n in neighbours(current) do
4:         if n is superior to current then
5:             current ← n
6:     if no improvement then
7:         current ← random assignment
```

The neighbourhood of a solution in the space of allocations is defined as all those solutions that can be generated by substituting another variant of the same task for one already allocated, or by moving a single variant to a different processor. In order to determine if one solution is preferable to another, a priority ordering amongst the constraints and objectives is established, in order of importance:

1) No processors should be overloaded.
2) The network should not be overloaded.
3) Residency constraints must be satisfied.
4) Coresidency constraints must be satisfied.
5) Average QoS per task should be maximised.
6) Average free capacity per CPU should be maximised.

A solution is feasible if the first four constraints are satisfied, after which the search will try to optimise QoS and then reduce processor utilisation to free up capacity. This priority ordering method is preferred over the alternative of a *weighted sum objective*, an approach found elsewhere in the literature [17]. Weighted sum approaches require the user to define numerical relationships between objectives and constraints, which is a somewhat inelegant approach to this problem. For the same reason, we prefer local search over *simulated annealing* [25], an algorithm we also experimented with, which relies on a numerical gradient in the constrained objective space.

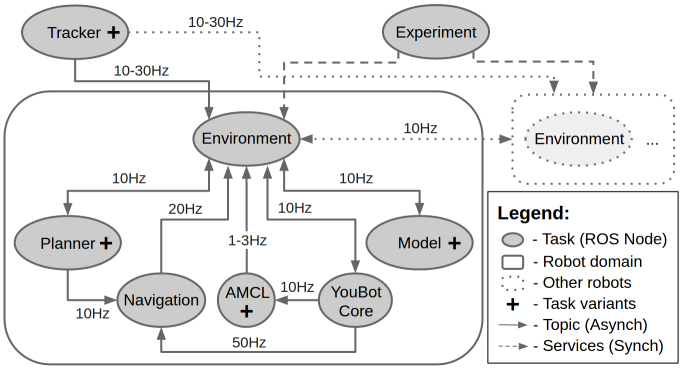


Fig. 1: Case study software architecture, composed of one *Tracker* instance per camera, one instance of each task in the *Robot domain* per robot, and one *Experiment* instance for the complete system.

IV. Example Case Study

Our case study serves as a specific instantiation of the general model presented, with which we can test our algorithmic solutions in a real system. We first present a "baseline" instance of the system, consisting of a single robot, person, server and camera. This simplified configuration illustrates the system components and the constraints imposed on them. Each agent (robot or human) is pursuing a spatial goal. The application's overarching QoS metric is a combination of essential requirements (e.g. avoid collisions between agents, minimise travel time to reach target goals), as well as more sophisticated preferences (e.g. minimise close-encounters and hindrance between navigating agents, minimise the time taken to infer the true agent goal). Therefore, task variants must be selected and allocated across available processors with the objective of optimising global QoS based on the selected variants' individual QoS values.

A. Software Architecture

Figure 1 shows a high-level diagram representing the software architecture of the case study. It is composed of multiple tasks and their message connections. In the figure, connections are labelled with message frequencies, which can be obtained from the maximum bandwidth requirement described in Section II-A. The QoS values for the variants of a given task represent the proportional benefit of running that task variant; a variant that has a higher QoS, however, would typically incur a higher CPU usage. We rely on an expert system user to estimate QoS values for task variants.

We now describe for each task in our case study, the corresponding variants (see Table I for details):

- *Tracker*: A component of a distributed person tracking algorithm that fuses multiple-camera beliefs using a particle filter. The variants for this task are based on the input image resolution and the output frame rate given a fixed number of cameras. The higher the output frame rate the more accurate the tracking.
- *Experiment*: A small synchronous task that coordinates all robots taking part in the experiment.

- *Environment*: A local processing task required by each robot. This task combines information generated by the local robot, other robots, and elsewhere in the system (i.e. *Tracker*, *Experiment*).
- *Model*: An intention-aware model for predicting the future motion of interactively navigating agents, both robots and humans. The variants for this task are based on the number of hypothetical goals considered given a fixed number of agents. A higher number of modelled agent goals will lead to more accurate goal estimates.
- *Planner*: Generates an interactive costmap, which predicts the future motion of all agents with relation to other agents' motion given their inferred target goals. This costmap is used by the *Navigation* task for calculating the trajectory to be executed.
- *AMCL*: A task performing localisation relying on laser data and a known map of the environment [23]. The variants of this task vary with the number of particles the monte-carlo localisation may use during navigation, since a larger number increases localisation robustness and accuracy in environments populated with other moving obstacles. We assume the robot moves on average at the preferred speed of 0.3m/s (min 0.1m/s, max 0.6m/s).
- *Navigation*: This task avoids detected obstacles and attempts to plan a path given the interactive costmap of the agents in the environment, ultimately producing the output velocity the robot platform must take. The variants of *Navigation* depend on the controller frequency, that is, the number of times per second the task produces a command velocity. The higher the frequency, the more reactive and smooth the robot navigation becomes.
- *YouBot_Core*: A core set of ROS packages and nodes that enable the robot to function, for example etherCAT motor connectivity, internal kinematic transformations, and a laser scanner sensor. This task must always run in the corresponding robot (a residence constraint).

Finally, it is critical that a robot can execute all of its own tasks, even if only using the least computationally demanding variants. Those tasks are represented within the robot namespace in Figure 1. This is essential to ensure a continued service in periods of network outage, albeit at lower levels of QoS.

B. Hardware Architecture

The hardware integrating the baseline system is composed of a single network camera and two processors, that is, a robot with onboard processor and a remote server. Robot and server communicate through a wireless network, and camera and server through a wired network. In practice the network bandwidth is currently not a limiting factor, as both networks are dedicated and private in our lab.

V. Evaluation

In this section, we first describe the results of an empirical characterisation of the baseline system, which is mandatory to evaluate both the solution methods and the case study itself. We then extend this characterisation to define a set of system instances of increasing size and complexity. Having established these benchmark problems, we employ them to evaluate the utility of our solution methods, in two stages.

In the first stage, we compare the quality of solutions returned by the three proposed methods to answer the following research questions:

- RQ1A. Is it possible to find globally optimal variant selections and allocations using constraint programming?
- RQ1B. How well can a straightforward greedy heuristic and the local search metaheuristic perform on this problem, relative to the constraint programming method?
- RQ1C. How well do the results produced by the three solution methods translate to deployment on the physical system outlined in Section IV?
- RQ1D. How effective are the allocations proposed by our solution methods compared to random allocations?

In the second stage, we compare an *anytime* version of the MiniZinc model solver against the local search metaheuristic, to explore their performance over time. Our research question is as follows:

- RQ2. How do local search metaheuristic and "anytime" constraint programming compare in terms of their solutions quality after a given period of run-time?

A. System Characterisation

We performed an offline characterisation of the baseline system using common monitoring utilities from ROS (e.g. *rqt*) and Linux (e.g. *htop*). The objective was to measure for each task in the system the following values: i) the average percentage of CPU utilisation required for each variant on each processor, and ii) the average frequency at which messages published by variants are sent to other tasks, along with the bandwidth required for each type of message.

Table I summarises the values obtained. Column two represents the number of variants for each task, and column three the value of the parameters that create the task variants (see Section IV-A). The next three columns include the average values of CPU utilisation, frequency and bandwidth for each task variant — note that the maximum values for frequency are shown in Figure 1. The CPU values for the *Tracker* task assume only one person in the environment. Columns seven and eight show the residence and coresidence constraints for each variant and task respectively. Finally, the last column represents the normalised QoS associated with each task variant, where 100 is the maximum value. Note that we have assigned QoS value "1" to single variant tasks because they have much less impact in the system behaviour, which is reflected in low CPU utilisation values in Table I.

The focus of this work is task variant allocation, for which we require QoS values as inputs. Although QoS values were manually generated based on real system measurements, they may be automatically generated, but we leave this for future work. It is worth noting that the user is required to provide QoS values only "once" for each task variant. Therefore, when the system is scaled up by replicating tasks on more robots or cameras, the user is not required to assign new QoS values.

TABLE I: Task variants characterisation.

Task	Variants	Parameters	CPU	Freq (Hz)	BW (KB/s)	Res	CoRes	QoS
Experiment	1	-	1	10	1	server	-	1
Tracker	4	Output freq. (25 20 15 10)	200 160 120 80	25 20 15 10	2.5 2.0 1.5 1.0	server	-	100 90 70 40
Environment	1	-	1	10	0.5	-	-	1
Model	3	Num. goals (10000 3500 4)	59 39 17	10 10 10	5 5 5	-	-	100 60 20
Planner	1	-	1	10	0.5	-	Navigation	1
AMCL	3	Particles (3000 500 200)	66 41 19	2.5 2.5 2.5	1 1 1	-	-	100 75 50
Navigation	3	Controller freq. (20 10 2)	50 39 25	20 10 2	1 0.5 0.1	-	Planner	100 67 33
Youbot_Core	1	-	16	10	0.5	robot	-	1

Finally, we specify the characteristics of the hardware used to obtain the measurements. The robot's on-board processor is an Intel Atom, 2 cores @ 1.6GHz, 2GB RAM. The server's processor is an Intel i5-3340, Quad Core @ 3.30GHz (Turbo), 16GB RAM. Note that all CPU measurements are normalised to the robot CPU capacity (= 100). From this, we can understand why the *Tracker* instances (which have a high CPU requirement) can only run in the server, translating into a residence constraint. The networks employed are a wireless 802.11ac network at 300Mbps, and a 1Gbps Ethernet network.

B. System Instances

In order to obtain more complex instances of the system, we only need to add processors (robots, servers) and/or cameras, allowing the system to cope with a more complex environment and complete more difficult challenges. As these parameters are varied, the total number of tasks and variants change accordingly, but the number of variants for each task is fixed. Table II summarises the set of instances comprising our benchmarks, and the number of tasks and variants generated for each case — note that only one server is used for all cases.

TABLE II: System instances considered

Instance	Processors	Robots	Cameras	Tasks	Variants
1	2	1	1	8	17
2	2	1	2	9	21
3	2	1	3	10	25
4	3	2	1	14	29
5	3	2	2	15	33
6	3	2	3	16	37
7	4	3	1	20	41
8	4	3	2	21	45
9	4	3	3	22	49
10	4	3	4	23	53

C. Simulation Results: QoS Analysis

We now analyse and compare the QoS values of solutions provided by the three proposed methods (since these are simulation results, we call them expected values). Remember that the allocation of more powerful variants translates into higher global QoS values, and strongly correlates with improved overall system behaviour. For example, switching from the least to most powerful variant of the $Tracker$ task (QoS values 40 and 100, Table I) actually provides more accurate and faster tracking of people in the environment. This in turn provides the $Planner$ and $Model$ tasks with better data, improving the robots ability to navigate (e.g. avoiding collisions).

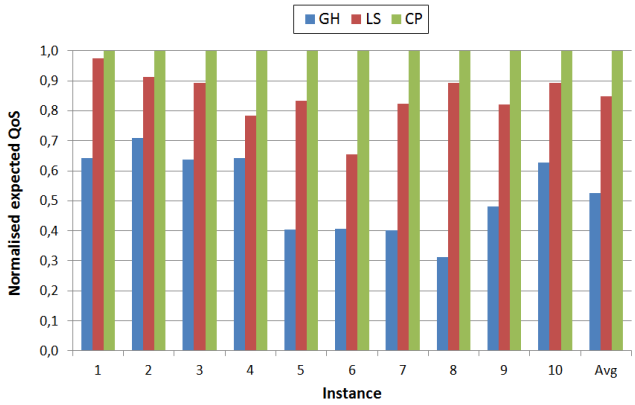


Fig. 2: Expected QoS for greedy heuristic (GH), local search (LS), and constraint programming (CP). Server capacity = 400.

We execute Python programs implementing the three proposed methods for the instances described in Table II. Answering RQ1A, we found that constraint programming finds the globally optimal solution for all instances analysed. In other words, for each instance this method provides the allocation of task variants to processors with the best possible average QoS and minimum CPU usage. Since constraint programming provides the best possible QoS, we normalise the QoS provided by the greedy and local search methods to the optimum. Figure 2 shows results comparing the three methods — note that values for LS are actually the average of three independent runs considering the amount of time used by CP. Therefore, answering RQ1B, we observe that LS and GH achieve an average of 15% and 47% less QoS than CP respectively.

Since we maintain the server capacity (= 400) across all instances analysed, the problem becomes more constrained as the total number of variants increases. As an example, CP solves $Instance1$ allocating the most powerful variants for all tasks. However for $Instance10$, all tasks need to use less powerful variants in order to satisfy the CPU capacity constraint (e.g. the four $Tracker$ tasks use the least powerful).

D. Analysis of Case Study Behaviour

Having obtained the simulation results, our next step is to validate that the expected QoS values obtained via simulation match the behaviour of the real system. To do this, we performed experiments for instances 1-6 from Table II in our case study environment. For each instance, we configured the allocation of task variants to processors computed by the

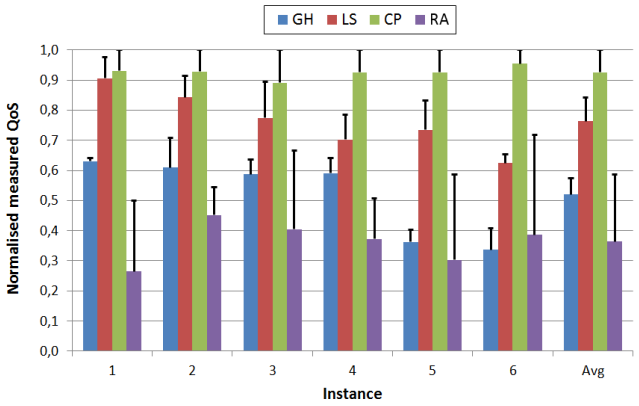


Fig. 3: Measured QoS for greedy heuristic (GH), local search (LS), constraint programming (CP), and random allocations (RA). Error bars represent the deviation from the expected QoS values.

solution methods — note that only a single human agent is present in the environment for all experiments. Then, the measured QoS value for each instance and method is obtained by applying the following formula:

$$QoS_{measured} = \sum_{\tau_i \in T} QoS_{v_j^i} \times \frac{F^o_{v_j^i}}{F^e_{v_j^i}} \tag{8}$$

where $QoS_{v_j^i}$ is the expected QoS value for task variant v_j^i as predicted by our solution methods, $F^o_{v_j^i}$ is the observed frequency of messages produced by v_j^i on the real system and $F^e_{v_j^i}$ is the expected frequency associated with v_j^i (Table I). These two frequencies can differ due to overloaded processors (for infeasible solutions) and/or approximation errors in the system characterisation. Therefore, this frequency ratio determines the effectiveness of a task variant in the real system.

Figure 3 shows the results. The black error bar for each column denotes the difference between measured QoS (top of column) obtained with Equation 8, and expected QoS (error bar upper end) obtained by simulation. Answering RQ1C, the measured QoS values for local search, constraint programming and the greedy heuristic only deviate by 8%, 7% and 5% on average respectively from the expected values. This result validates the accuracy of our methodology.

Finally, we also examined the system behaviour considering random allocations of task variants to processors. Figure 3 also includes these results (RA), where each bar actually corresponds to the average QoS of three randomly generated allocations. Answering RQ1D, we see how the measured QoS values for random allocations deviate much more from the expected ones, by an average of 22%, than those for the proposed solution methods. The reason is that our solution methods produced feasible allocations for the six instances analysed (i.e. satisfying system constraints), thus differences are only due to approximation errors in the system characterisation. However, some of the random allocations produced infeasible solutions, which translated into overloaded processors and therefore larger differences with the expected values.

In summary, constraint programming improves by 16%, 41%, and 56% on average over local search metaheuristic, greedy heuristic and random allocations respectively.

E. Anytime Approaches

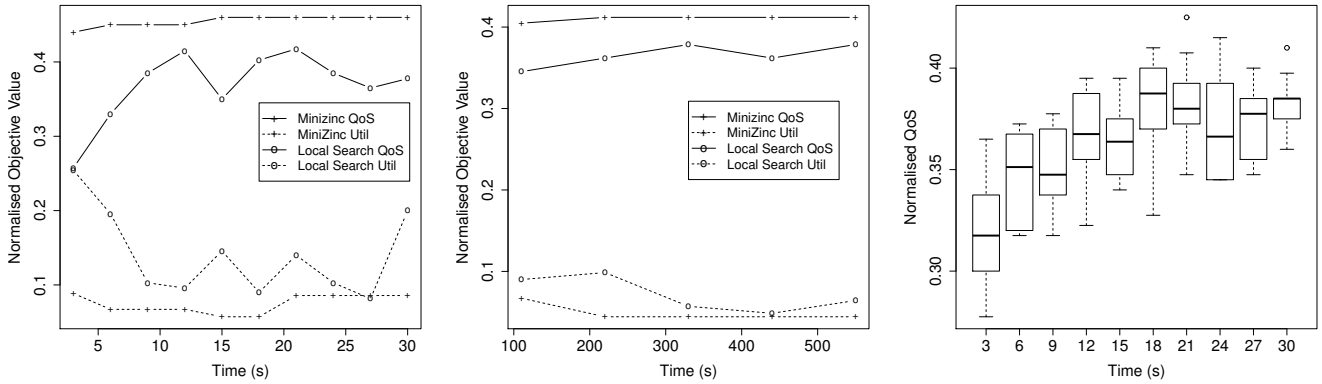
We now consider task allocation using the MiniZinc model solver *Gecode* and the local search metaheuristic as *anytime algorithms*, where the best allocation currently known can be returned at any point during their execution. This is particularly important if we are to allocate variants in larger systems or at run-time. The two algorithms approach the problem differently, because using *Gecode* requires a two-pass procedure where each objective is optimised in turn, whereas the local search metaheuristic attempts to optimise both objectives simultaneously. Therefore, the relative performance of the two algorithms is of interest.

The experiments were performed on a 2.7GHz Intel Core i5 iMac with 16GB RAM, which gives similar solution times to those presented above. We first ran *Gecode* to completion against each benchmark instance. We selected the smallest two instances that resulted in significant runtimes, which were Instance 7 (approximately 25 seconds) and 8 (550 seconds).

We then executed *Gecode* and local search with increasing timeout values, to evaluate how the solutions they found improved over time. Figures 4a and 4b show typical results. The graphs show two objectives: firstly, the Quality of Service objective as defined by Equation 1, and secondly the Utilisation objective as defined by Equation 2. Each intermediary result is from an independent run of the algorithms, avoiding the problem of autocorrelation. All results in Figure 4 are normalised to ideal ("1"), which represents: i) for QoS values, the QoS of using the most powerful variants; ii) for CPU utilisation, unutilised processors (free capacity of 100%).

These graphs illustrate a clear trend that answers RQ2: MiniZinc produces superior results in the same amount of time, and is our preferred anytime solution method. Promisingly, it also produces high quality results within a short timeframe, which may enable dynamic optimisation in the future and also increases our confidence in its ability to scale to larger systems. Local search produces feasible solutions with better utilisation values (more free capacity) in a short amount of time, however our case study architects are primarily concerned with QoS. As our local search algorithm is implemented in Python, it may be argued that MiniZinc has an unfair advantage in that its solvers are written in C; however, the highly optimised nature of constraint solvers is actually a strong argument in favour of adopting them, particularly as they improve through continuous development over time.

The fifth value for local search QoS in Figure 4a is lower than the preceding and following values, which suggests that there is a certain amount of variance in the results produced by local search, based on the seed provided. To measure the variance, we repeated the experiment ten times using local search, and present the results in Figure 4c. This underlines the fact that the performance of local search is quite variable, although it generally makes steady progress over time.



(a) (b) (c)

Fig. 4: Anytime results: (a) Instance 7; (b) Instance 8; (c) Distribution on Instance 7 with 10 Repetitions.

VI. Related Work

Much work has been performed in the area of task allocation in distributed robotics, where different types of optimisation problems have been addressed. A comprehensive taxonomy can be found in [12], which in turn is based on an earlier taxonomy [8]. According to these taxonomies, the task variant allocation problem presented in this paper falls in the category of Cross-schedule Dependencies (XD), that is, the effective utility of each individual task-agent allocation depends on both the other tasks an agent is performing, and the tasks other agents are performing. Several types of system configurations are supported within this category — e.g. MT-SR-IA considers multi-task robots (MT), single-robot tasks (SR), and instantaneous tasks assignment (IA). Furthermore, problems in this category can be formulated with different types of mathematical models. In our case, we use a special form of knapsack formulation (Section II). Below we outline key related work in distributed robotics falling in the same category, highlighting how our work differs from past research.

The first difference arises from the number of tasks and agents considered. Prior work based on the linear assignment problem [22] assumes a single task per agent [20, 16, 15, 14]. In our case, the number of tasks is equal to or greater than the number of agents (and the number of variants is greater still). A second point is related to the number of agents simultaneously completing tasks. In [6, 27, 2] several agents are required, which is a subset of our problem. Another consideration is that our system is fully heterogeneous, i.e. all tasks and processors may be different. Some past work does assume heterogeneous tasks and multiple instances of every task [19], but does not consider different variants of the same task, which is the principal addition to the problem here.

Aleti et al. [1] provide a high-level general survey of software architecture optimisation techniques. In their taxonomy, our work is in the problem domain of design-time optimisation of embedded systems. We explore optimisation strategies that are both approximate and exact. We evaluate our work via both benchmark problems and a case study. In terms of the taxonomy in [1] our work is particularly wideranging.

Finally, Huang et al. [10] consider the selection and placement of task variants for reconfigurable computing applications. They represent applications as directed acyclic graphs of tasks, where each task node can be synthesised using one of four task variants. The variants trade off hardware logic resource utilisation with execution time. Huang et al. use an approximate optimisation strategy based on genetic algorithms to synthesise the task graph on a single FPGA device.

To summarise, no existing work in the robotics field addresses all of the considerations that our proposal does, i.e. a constrained, distributed, heterogeneous system with more tasks than nodes and different variants for the tasks.

VII. Conclusion

We have addressed a unique generalisation of the task allocation problem in distributed systems, with a specific application to robotics. We advocate the use of task variants, which provide trade-offs between QoS and resource usage by employing different algorithms and/or taking advantage of heterogeneous hardware. We have presented a mathematical formulation of variant selection and assignment, and evaluated three solution methods on instances from a problem generator based on a robotics case study. We conclude that our solution methods are very effective in selecting and allocating variants such that QoS is optimised and resource usage minimised. We find high-quality solutions that translate well to real systems, providing a useful tool for the system architect.

Acknowledgments

This work was supported by the AnyScale Applications project under the EPSRC grant EP/L000725/1, and partially by the EPSRC grants EP/F500385/1 and EP/K503058/1, and the BBSRC grant BB/F529254/1. We thank Ornela Dardha for her valuable help in the problem formulation.

References

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *Software Engineering, IEEE Transactions on*, 39(5):658–683, 2013.

[2] S. Balakirsky, S. Carpin, A. Kleiner, M. Lewis, A. Visser, J. Wang, and V. A. Ziparo. Towards heterogeneous robot teams for disaster mitigation: Results and performance metrics from robocup rescue: Field reports. *J. Field Robot.*, 24(11-12):943–967, November 2007.

[3] R. Bischoff, U. Huggenberger, and E. Prassler. Kuka youbot - a mobile manipulator for research and education. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4, May 2011.

[4] A. Bordallo, F. Previtali, N. Nardelli, and S. Ramamoorthy. Counterfactual reasoning about intent for interactive navigation in dynamic environments. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 2943–2950. IEEE, 2015.

[5] J. Cano, E. Molinos, V. Nagarajan, and S. Vijayakumar. Dynamic process migration in heterogeneous ROS-based environments. In *Advanced Robotics (ICAR), 2015 International Conference on*, pages 518–523, July 2015.

[6] Jian Chen, Xiao Yan, Haoyao Chen, and Dong Sun. Resource constrained multirobot task allocation with a leader-follower coalition method. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 5093–5098, Oct 2010.

[7] Gecode Team. Gecode: Generic constraint development environment. http://www.gecode.org, 2006.

[8] Brian P. Gerkey and Maja J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9): 939–954, 2004.

[9] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 13–24, 2010.

[10] Miaoqing Huang, V.K. Narayana, M. Bakhouya, J. Gaber, and T. El-Ghazawi. Efficient mapping of task graphs onto reconfigurable hardware using architectural variants. *Computers, IEEE Transactions on*, 61(9):1354–1360, 2012.

[11] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

[12] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, October 2013. ISSN 0278-3649.

[13] Dong-Hyun Lee, S.A. Zaheer, and Jong-Hwan Kim. Ad hoc network-based task allocation with resource-aware cost generation for multirobot systems. *Industrial Electronics, IEEE Transactions on*, 61(12):6871–6881, Dec 2014.

[14] L. Liu and D. Shell. A distributable and computation-flexible assignment algorithm: From local task swapping to global optimality. In *Proc. of Robotics: Science and Systems*, Sydney, Australia, July 2012.

[15] L. Liu and D. Shell. Optimal market-based multi-robot task allocation via strategic pricing. In *Proc. of Robotics: Science and Systems*, Berlin, Germany, June 2013.

[16] Lingzhi Luo, N. Chakraborty, and K. Sycara. Provably-good distributed algorithm for constrained multi-robot task assignment for grouped tasks. *Robotics, IEEE Transactions on*, 31(1):19–30, Feb 2015.

[17] R. Timothy Marler and Jasbir S. Arora. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, 41 (6):853–862, 2009.

[18] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[19] Natsuki Miyata, Jun Ota, Tamio Arai, and Hajime Asama. Cooperative transport by multiple mobile robots in unknown static environments associated with real-time task assignment. *IEEE T. Robotics and Automation*, 18 (5):769–780, 2002.

[20] Changjoo Nam and D.A. Shell. Assignment algorithms for modeling resource contention and interference in multi-robot task-allocation. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 2158–2163, May 2014.

[21] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Proc. of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, 2007.

[22] David W. Pentico. Assignment problems: A golden anniversary survey. *European Journal of Operational Research*, 176(2):774–793, 2007.

[23] Patrick Pfaff, Wolfram Burgard, and Dieter Fox. Robust monte-carlo localization using adaptive likelihood models. In *EUROS*, pages 181–194, 2006.

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

[25] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: An NP-Hard problem made easy. *Real-Time Systems*, 4(2):145–165.

[26] David White and Jose Cano. Task variant allocation repository. https://github.com/ipab-rad/task_alloc.

[27] Yu Zhang and Lynne E. Parker. Considering inter-task resource constraints in task allocation. *Autonomous Agents and Multi-Agent Systems*, 26(3):389–419, 2013.