Enabling Optional Software Components In Multi-Constrained Deployment Optimization Problems

Hamilton Turner and Alan Baines
Dept. of Electrical and Computer Engineering
Virginia Tech
Email:{hturner, abaines}@vt.edu

Abstract—This paper presents methods for adding optional software components to a Distributed Real-Time Embedded (DRE) deployment optimization problem, and methods for attempting to enable deployment of the maximal number of optional software components. By building upon prior work that allows rapid deployment optimization, one can predict probability of success with high confidence, and utilize that information to generate a lower bound on number of times deployment of an optional component should be attempted. This paper also introduces three algorithms for selecting which optional components should be attempted in a deployment, and experimental data which shows that the order in which optional components are considered has a high impact on the maximal number of optional components that can be included in a deployment.

Keywords—deployment, optional software components, distributed real-time embedded systems

I. INTRODUCTION

Emerging Trends and Challenges.

Modern aerospace craft can contain hundreds of mission-critical software components, tens of specialized computing hardware, and multiple redundant networking topologies. This diverse architecture and strict software requirements, such as real-time software execution, together result in most modern aerospace craft being an example of a distributed real-time embedded (DRE) hardware system. Deployment optimization, in which necessary software components are mapped onto available hardware nodes, is a critical challenge in modern DRE systems.

These deployment challenges are frequently faced in large, complex systems such as the Lunar Reconnaissance Orbiter or the International Space Station. Common challenge set sizes include thousands of software components, hundreds of hardware nodes, and tens of types of constraints. Constraints can include resource allocation challenges (e.g. ensuring that all software components have enough memory or ensuring that real-time scheduling challenges can be met), co-location challenges (e.g. critical software system X and it's backup system Y cannot be executing on the same hardware node), bandwidth and connectivity challenges (e.g. software component A frequently communicates with B, so they should be co-located to conserve bandwidth), and others. Searching for a valid deployment, i.e. one that does not violate any constraints, often took upwards of a month for large systems.

The complexity of these deployment challenges, and the large lengths of time required to solve them have resulted in projects limiting software components to one of two categories—software that will be deployed, and software that will not, due to hardware resource limits. This decision is often made before before either software or hardware development is complete. By delaying the final deployment decision to later in the production process, improvements and new components can continue to be added, and recent techniques for deployment optimization have enabled the possibility of performing a complete deployment in under one minute [1]. This new rapid deployment approach enables a number of interesting new technology ideas.

Open Problem ⇒ Rapid Runtime Deployment Including Optional Software Components.

One primary lacking feature in DRE deployment optimization was the ability to specify various software components as optional. By providing optional components, a rapid deployment system could continually search after finding a valid deployment for the required components and attempt to include as many optional software components as possible. Moreover, in many state changes, such as an asteroid that happens to be flying close to a deployed satellite, it would be possible to rapidly evaluate the potential gain of performing a redeployment of software to a system if there was a continually-executing deployment system that monitored environmental context.

Solution Approach ⇒ Using Statistical Methods and Providing Optional Component Selection Algorithms

To address the need for optional component inclusion in software components, we have enabled distinction of the ‘no solution found’ and ‘no valid solution’ states by utilizing standard statistical methods. We have further generated and tested algorithms for ensuring that the maximal number of optional software components are located. Our solution included three algorithms for adding optional components to DRE system. Due to the speed of the underlying code, our solution is able to try multiple configurations to determine if they are deployable while keeping solutions that work and discarding solutions that do not.

In Section V we present empirical data that we have gathered from experiments showing that all methods were able to add optional components. The First-Fit Algorithm was
able to include the most optional components, followed by
the Greedy. These results improve upon existing solutions by
allowing and finding optional components. This paper provides
the following contributions to the study of including optional
software components in deployment optimization:

- A method of confidently differentiating between ‘No
  Valid Deployment Found’ and ‘No Valid Deployment’
  states given a rapid deployment solver
- A comparison of three algorithms for attempting to
deploy optional components using a deployment solver
- An extension upon prior work which allows including
optional components in a complete DRE system specifi-
cation
- An example implementation of software that computes
valid deployments while trying to maximize the utility of
optional components

The remainder of this paper is organized as follows: Section
II describes a fractioned spacecraft deployment, which we
use as a motivating example throughout the paper; Section III
discusses the challenges that we faced when attempting to add
optional components to a deployment; Section IV covers the
our solution approach to enabling deployment of optional com-
ponents; Section V presents empirical results; and Section VII
presents concluding remarks and lessons learned.

II. MOTIVATING EXAMPLE: FRACTIONED SPACECRAFT
DEPLOYMENT

The fractioned spacecraft architecture case study, is an
approximation of a real-world system used in the NASA Earth
Science Enterprises Magnetospheric Multi-Scale (MMS) [2]
mission. The system uses five satellites, with six sensors on
each satellite, as a solar-terrestrial probe. Figure 1 shows an
overview of the MMS mission case study system.

The Fractioned spacecraft weather satellite is controlled by
a ground station which can dynamically change the spatial
deployment of software components to satellites in order to
change system functionality, adapt to hardware failures, or
improve system performance. While the example shown in
figure 1 shows only three software components, production
deployment challenges involve hundreds or thousands of soft-
ware components and many more component interactions.

The shown satellite runs a set of software components, such
as the ImageAcquisition component for capturing weather
images, the FeatureIdentification component for identifying
features and characteristics of captured images, and the
FeatureReporting component for relaying detected feature
information to the ground station. A generated deployment
must ensure that each software component meets its real-time
deadlines, memory consumption limits, and does not need
more than the available communication bandwidth.

III. CHALLENGES OF INCLUDING OPTIONAL SOFTWARE
COMPONENTS IN DEPLOYMENT OPTIMIZATION

Although including optional software components in a DRE
deployment optimization would clearly have a number of ben-
efits, such as enabling higher hardware utilization, increasing
system redundancy, and reducing cost, there are a number of
challenges. Not only is the solution state space large, but the
density of valid solutions is unknown. This section describes
in detail two of the harder technical challenges resulting from
attempting to include optional software components into a
deployment optimization approach.

A. Challenge 1: Optional Components Cause Explosion In
Solution State Space Size

For deployment optimization challenges, each solution state
is typically an n-length vector, where n is the number of
software components. Each vector item is a mapping from a
software component to a hardware node. Given this rep-
resentation, the unpruned size of the state space is every
possible permutation of software onto hardware nodes. Given
α hardware nodes, and β software components, the state space
size is αβ. Adding just one optional component increases the
state space from O(αβ) to O(αβ+1). While various heuristics
and pruning methods can be used to reduce this the state space,
it is likely that for non-trivial numbers of optional software
components this increase in state space size will result in an
increase in deployment optimization execution time.

In our motivating example, this challenge could manifest
as system redundancy or system functionality. For example,
if there are modules such as RareFeatureIdentification, that
provide functionality beyond what the basic FeatureIdenti-
fication module allows, then including these optional components
in the deployment may make sense. However, each included
optional component causes a large jump in solution state space
size. Moreover, perhaps the system is interested in providing
redundancy on three modules – this would be equivalent
to having three optional modules with the same constraint
fingerprints as their required counterparts and would cause a
large increase in state space size.

B. Challenge 2: Inability to Differentiate No Solution from No
Solution Found

Given the large number of complex constraints involved in
finding a valid solution to a deployment optimization problem,
it can be difficult to express concisely why a solution was not
found. In general, it is unlikely that any moderately complex
deployment optimization software can differentiate between
an impossible deployment situation and one where a solution
has yet to be found. This presents unique challenges for
including optional software components in a deployment: if
a deployment is attempted with optional software components
and fails, then we are left unable to determine if that failure is
due to poor luck in initial random particle placement, or if the
failure is actually due to an inability to satisfy all deployment
constraints given the optional software components.

For example, if the optional software component RareFea-
tureIdentification does not located a valid deployment, it is un-
clear if it has a set of constraints that, in conjunction with other
required and other optional components, renders it impossible
to deploy, or if the deployment solver needs more time. This
decision has to be counterbalanced with the utility of the
RareFeatureIdentification component e.g. is the component
likely to be beneficial enough that it’s worth trying harder.
Lastly, in many DRE systems the component deployment may stay fixed for many years after the initial deployment is settled, which might substantially affect the effort that should be put into deploying the optional components by increasing their value over a long period of time.

IV. UTILIZE PROBABILITY OF DEPLOYMENT SUCCESS TO ENABLE ATTEMPTING DEPLOYMENTS WITH OPTIONAL COMPONENTS

Figure 2 shows the high-level approach for testing if optional software components can be included in a deployment optimization. The primary work of this component is the ability to differentiate between a solution being impossible with the given constraints, or a solution failing to be located due to a sparse solution space, and methods of selecting optional components to rapidly locate a valid deployment with a maximal number of optimal components included. This solution builds on prior work which provided a deployment optimization solver based upon integrating a metaheuristic, e.g. particle swarm, and multiple heuristic, e.g. First-Fit Bin Packing, algorithms. This prior work enabled rapid deployment optimization, thereby enabling the building of algorithms on top of a deployment optimization solver.

The primary contributions of this research include:

- An algorithm for selecting optional components to be considered in a deployment
- A method of reasonably differentiating between ‘No Valid Deployment Found’ and ‘No Validation’ states
- An extension upon prior work which allows including optional components in a complete DRE system specification

A. Differentiating Between ‘No Solution’ and ‘No Solution Found’ states

While prior deployment optimization approaches rendered large-scale repeated deployments infeasible, the small deployment times established by prior work make large-scale executions of a deployment possible [1]. Given this drastic change, we believe it is possible to run experiments to determine the probability of a valid solution being found. By taking a diverse set of hardware/software combinations known to have a solution and leaving algorithm tuning parameters, such as the number of initial particles in the particle swarm, constant, we can determine the percentage of times a valid solution will be found if one exists, using the algorithms tuned parameters described. Standard statistics can then be used to determine a 95% confidence interval for this percentage. This interval can be used to determine the number of times a deployment should be attempted before reasonably assuming that there is no valid solution, as shown below:

Define $p$ as the percentage of times a valid deployment will be found.

Define $n$ to be the sample size, where $n$ is much smaller than the population.

Standard error of $p$ = $SE_p = \sqrt{\frac{p(1-p)}{n}}$

95% Confidence Interval = $[p - 1.96 \times SE_p, p + 1.96 \times SE_p]$ 
(Note that $p = \mu$)

Re-Sets Required = $\max\left(\frac{1}{p - 1.96 \times SE_p}, \frac{1}{p + 1.96 \times SE_p}\right)$

Note that once a system definition is changed e.g. an optional component is added, this probability of success will change. In all cases, the new probability of success will be equal to or less than the current probability - adding more resource-consuming software components will never make a deployment challenge easier. However, this gives a base number of attempts required for determining if the solution is valid or invalid, e.g. if the required components need at 4 attempts to succeed then testing the required components plus one optional will require at least 4 attempts as well. As optional components are added, small experiments can be performed to update the new value of $p$ and therefore update the base number of cases to be tested.
B. Selecting Optional Components for Deployment Testing

In order to solve adding optional components to a deployment, three algorithms were constructed. Each had increasingly complex methods in attempt to get better or faster results.

1) Naive Algorithm: As Listing 1 shows, the Naive algorithm simply iterates over the list of optional components and attempts to deploy each one. It makes no attempt to choose more (or fewer) constrained optional components, and components are not reconsidered if a deployment including them fails. This approach to choosing optional software components has a $O(N)$ running time.

```
optional = list of optional components
included = empty list
required = list of non–optional components

while optional.size != 0:
current = pop(optional)
try deploy(included+current+required)
p-success times:
    if solution_found:
        included.add(current)
```

Listing 1. Naive Algorithm

2) First Fit Algorithm: The First Fit algorithm (Listing 2) is derived from the first fit decreasing heuristic approach to bin packing, in which elements are sorted in decreasing order of volume and each element is inserted into the first bin in which it will fit. For bin packing, the goal is to first place elements that will cause the most restrictions, and incrementally place easier elements into the first bin in which they fit. This approach is intended to reduce the branching factor of a search. For optional software components, the exact opposite e.g. choosing components that will increase the problem constraints the least (e.g. maintain the branching factor the most) approach makes sense. If all optional software components have utility one, then choosing components that will preserve the highest branching factor should result in a solution that is similar to First Fit bin packing, and will hopefully be similarly heuristically pseudo-optimal.

Unfortunately, determining the added difficulty due to one software component is hard due to the interconnected nature of the constraints. Understanding added difficulty is analogous to understanding the density of solutions in the search space, which is difficult to know without actually attempting the deployment. Due to these limitations, this algorithm would primarily be useful where one component was less constrained and more valuable (e.g. more utility value) than another component on every single constraint, which would safely allow the less constrained component to be considered first.

```
optional = sort(optional_components)
included = empty list
required = list of non–optional components

while optional.size != 0:
current = pop(optional)
try deploy(included+current+required)
p-success times:
    if solution_found:
        included.add(current)
```

Listing 2. First Fit Algorithm

3) Greedy Algorithm: This approach reevaluates the cost of adding components between each addition, on the basis that with the addition of a new component, the solution space has changed. This is due to the massively interconnected nature of deployment constraints. Methods that use this approach would have a heuristic sort method that attempts to intelligently guess which components will result in a higher branching factor. These components are then attempted for deployment first. Due to the need to sort the component list repeatedly for each iteration, this approach can require $O(N^2)$ in the worst case.

```
optional = list of optional components
included = empty list
required = list of non–optional components

while optional.size != 0:
sort(optional)
current = pop(optional)
try deploy(included+current+required)
p-success times:
    if solution_found:
        included.add(current)
```

Listing 3. Greedy Algorithm
V. Empirical Evaluation of Learning

A. Experimental Platform

The experimental platform used was a Dell blade server with two Intel E5620 (2.4GHz, 6 cores) processors, with 24 Gb of RAM split evenly between the two processors. In all experiments described below, the exact same experiments were run on multiple blade servers in parallel to increase the quantity of experimental data resulting. Due to the use of timing information, it was also ensured that each blade server had a clean install of CentOS release 5.8 kernel 2.6.18-308.1.1.el5 and each experiment process as isolated as possible (e.g. no other non-critical processes).

B. Experiment 1: Determining Baseline Probability of Deployment Success

In order to use the methods outlined in Section IV-B, we first had to determine a baseline success probability factor for the running of the deployment optimization. This can be difficult due to the randomness expressed by heuristics being utilized and the dynamically changing solution space created by adding and removing optional components. Deployment optimization was not the main algorithm of interest, so we chose to limit the input configuration parameters of the deployment optimization to a single configuration that was known to occasionally find a valid solution. The underlying deployment optimization is utilizing a particle swarm, and we set the dominant parameters of iterations and particle count to 20 and 200 respectively. Using this limited input space, we were able to locate the success probability by taking a simple random sample with a 95% confidence interval. We found that the probability of a valid deployment being achieved in 20 iterations was 34% with a 3.1% margin of error after taking 220 samples. This means that \( \frac{1}{0.34 - 0.31} \) or 4, is the number of times we would have to attempt a deployment with only required components to differentiate between no solution and no solution found. Adding more software components can only reduce this probability (and therefore increase the number of attempts necessary), but we know that trying each optional component deployment fewer than four times is highly unlikely to generate a result.

We were able to use multiple problem definitions and effectively add optional components, but there was no clear discernible pattern as to the number of optional components being deployed e.g. the utility added. This challenge led us to Experiment V-C.

C. Experiment 2: Evaluation of Optional Component Selection Algorithms

While Experiment V-B showed that it was in fact possible to add a number of optional software components by utilizing the \( \frac{1}{2} \) method, it was unclear how many components are being added. Section IV-A outlined three different approaches to selecting optional components which we propose as different methods to attempt adding of optional software components. In this experiment we evaluated the average number of optional software components that each algorithm was able to add to a deployment with the intention of comparing each algorithm.

D. Analysis of Results

Both Experiments V-B and V-C showed that we were able to successfully deploy multiple optional components in addition to the required component deployment. We were able to show that the impact of required components on the number of optional deployable software components was effectively minimized in Experiment V-C, allowing us to effectively evaluate

Due to the high variability of optional component deployment found in Experiment V-B, we chose to simplify the required component deployment challenge in order to focus on the performance of optional component selection. This was done by essentially treating all components as optional—due to challenges of the underlying deployment software, we accomplished this by making the required components have a predictable deployment plan, and verified that deploying the required components succeeded with 100% probability across one thousand test cases.

Our problem space included 16 computational nodes, 16 required components, and 14 optional components. It is unknown the true maximum number of components that the system can deploy, and therefore we chose to evaluate based upon the average performance over a large number of trial runs. Each algorithm was left to run repeatedly over long duration to attempt to overcome the randomness seen in the previous experiments.

Hypothesis: Higher-complexity algorithms would deploy more optional components. We expected that the First Fit Algorithm would be able to include more optional components than the Naive Algorithm, and that the Greedy Algorithm would be able to include more optional components than the First Fit Algorithm due to it’s substantially longer running time.

Experiment 2 Results. All of the methods were able to add some number of optional components to the list of required components. From Table I the results of three algorithms can be compared. The first-fit algorithm was able to achieve the best results, maximally adding 7 of the total 14 optional components. The Naive Method interestingly always returned the same results, which verified that the randomness associated with the underlying required components was successfully negated. The Greedy algorithm was actually worse on average than the Naive algorithm and took substantially longer, and was therefore not a very effective improvement. Unfortunately the improved algorithms had only minor changes on the effectiveness of inserting optional components. However, it is theoretically possible that seven optional components found by the First Fit algorithm are very close to the global best case.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Naive</th>
<th>FF</th>
<th>Greedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Max</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Average</td>
<td>5.31</td>
<td>4.88</td>
<td>4.88</td>
</tr>
<tr>
<td>Std Dev</td>
<td>0.77</td>
<td>0.70</td>
<td>0.70</td>
</tr>
<tr>
<td>Average Execution Time (seconds)</td>
<td>149</td>
<td>172</td>
<td>731</td>
</tr>
</tbody>
</table>

D. Analysis of Results

Both Experiments V-B and V-C showed that we were able to successfully deploy multiple optional components in addition to the required component deployment. We were able to show that the impact of required components on the number of optional deployable software components was effectively minimized in Experiment V-C, allowing us to effectively evaluate
the optional component picker. The First-Fit algorithm, which was derived from the first-fit bin packing heuristic, was the most successful at maximizing optional software components.

As shown in Figure 3, when the First Fit algorithm was able to fit seven optional components into the deployment, the average execution time to complete the deployment was less than the average time required to complete other (less successful) deployments. Two possible explanations for this include the increased execution time of a failure or the impact of a successful initial sort. In the general case, failed deployments take longer to execute in our system because the must attempt the deployment the maximal number of times, whereas valid deployments immediately return success. Additionally, a better initial sort of the components would result in components that are less constraining being packed first, which would not only make valid deployments be easier to find due to the lower complexity overall problem, but also would allow the algorithm to move on to attempting the next optional component.

VI. RELATED WORK

Genetic Algorithms Many of the related works [3] [4] [5] [6] [7] [8] used genetic algorithms to explore the solution space. The fitness function for these papers included financial cost [3], Quality of Service (QoS) [4] [8], performance and reliability [5], multiobjective [6], best service [7]. These domain-specific fitness functions allow the genetic algorithms to find near-optimal solutions for these problems. The goal in our DRE system is to maximize the number of optional components, while still being able to perform all of the required components.

Web Based Service Optimization These papers [3] [4] [8] focused on the goal of maximizing QoS or reducing cost of Web based service solutions. By managing multiple web services, these papers are able to focus on optimizing QoS [4] [8] or cost [3]. Web Based Services typically consider every services as optional and therefore can choose a solution space that is least constrained by composing components that work best together. DRE systems are required to have some set of components, and optional components must fit in the pre-existing constraint space.

Dynamic and On-Demand Allocation of Resources Work in [9] allowed for dynamic optimization of QoS-Aware services, while [7] focused on methods for On-Demand optimization of services on mobile phones. Both focused on managing services given a set of resources. [9] focuses on re-optimization when services crash or peers die. By modeling behaviors of services [7] can change based on current context. In contrast DRE systems perform off-line optimization allowing the resources that would normally be consumed by the optimizer to be used by additional required and optional components, and giving more detailed exploration of the solution space [9] [7].

VII. CONCLUDING REMARKS & LESSONS LEARNED

This paper discussed the challenge of including optional components in a DRE software deployment solver. Much of the work in this paper was directly enabled by [1], which allowed deployments to be attempted on a rapid timescale, such as under one minute. This paper utilized statistical methods to differentiate between “no solution possible” and “no solution found” states with a high level of confidence, and then used these methods to test multiple methods of automatically choosing which optional components to attempt to include in a deployment plan at one time. From our research on optional component deployment, we learned the following important lessons:

1) **It is Possible to Include Optional Software Components In a DRE Deployment Solver.** Despite the high complexity and highly random (due to underlying heuristics) nature of solving DRE deployment challenges, we were able to show that optional deployments can be included.

2) **Determining Constrainedness of a Problem is Difficult.** It can be extremely hard to create and utilize challenge problems for determining if a DRE deployment solver is performing at some expected level. As each optional component changes the solution space, it can be difficult to understand if the algorithm selecting optional components is performing poorly or has unlucky chosen a particular component that is highly constrained. Recognizing that the solution space never becomes denser lead to some algorithmic insights, and in future work we plan to extend this train of thought.

3) **First Fit Works Well as a Strategy for Sorting and Attempting Optional Component Deployment.** In Section IV-B2 we describe an algorithm derived from first fit bin packing. While in FFBP the goal is to choose the elements that most highly constrain the bins, we reversed the ordering and tried to deploy optional components in the order of least constrained to most highly-constrained. The claim was that lesser constrained components would be amenable to many of the initial constraint spaces provided by the required components and would therefore be successfully deployed rapidly. Experiment V-C shows that this method...
does indeed do better than the other approaches our solution tried.

The deployment solver described in this paper is available in open source form from http://code.google.com/p/ascentdesignstudio, and the experiments and associated data can be retrieved by contacting the authors.

REFERENCES


