Stressing Search with Scenarios for Flexible Solutions to Real-Time Task Allocation Problems

Paul Emberson, Iain Bate, *Member, IEEE*

Abstract—One of the most important properties of a good software engineering process and of the design of the software it produces is robustness to changing requirements. Scenario-based analysis is a popular method for improving the flexibility of software architectures. This paper demonstrates a search-based technique for automating scenario-based analysis in the software architecture deployment view. Specifically, a novel parallel simulated annealing search algorithm is applied to the real-time task allocation problem to find baseline solutions which require a minimal number of changes in order to meet the requirements of potential upgrade scenarios. Another simulated annealing based search is used for finding a solution which is similar to an existing baseline when new requirements arise. Solutions generated using a variety of scenarios are judged by how well they respond to different system requirements changes. The evaluation is performed on a set of problems with a controlled set of different characteristics.

Index Terms—maintainability, extensibility, heuristics, search, scheduling, scenarios

1 Introduction

Architecture design choices in software-based systems development can have a significant effect on the quality attributes of the system such as performance and maintainability [1]. Good software architecture design involves modelling it from more than one view. The focus of this paper is the mapping of software tasks to a hardware platform for real-time systems which is associated with the transition from the *process* view to the *physical / deployment* view [2] of an architecture. These views are further explained by figure 1. The process view defines the tasks and the dependencies between them. It is only when a mapping of this software to the physical hardware platform is chosen that the non-functional properties of the system become fully apparent [2]. In particular, software modules which are independent of each other in the logical view can still have timing interactions between corresponding run-time tasks. With a system using a system wide shared message bus, as in figure 1, there is the potential for small changes to a tasks' schedule to ripple throughout the system.

It is desirable for each iteration of a design to be closely related to the previous one on the assumption that the original design was not fundamentally flawed. Reasons for this include:

- maintaining documentation and the understanding engineers have of the system [3]
- containing changes may lower the cost of testing and re-certification of safety critical systems [4], which "dwarfs hardware costs" [5]
- changes to an interface between third-party components at the time of integration is expensive [6]

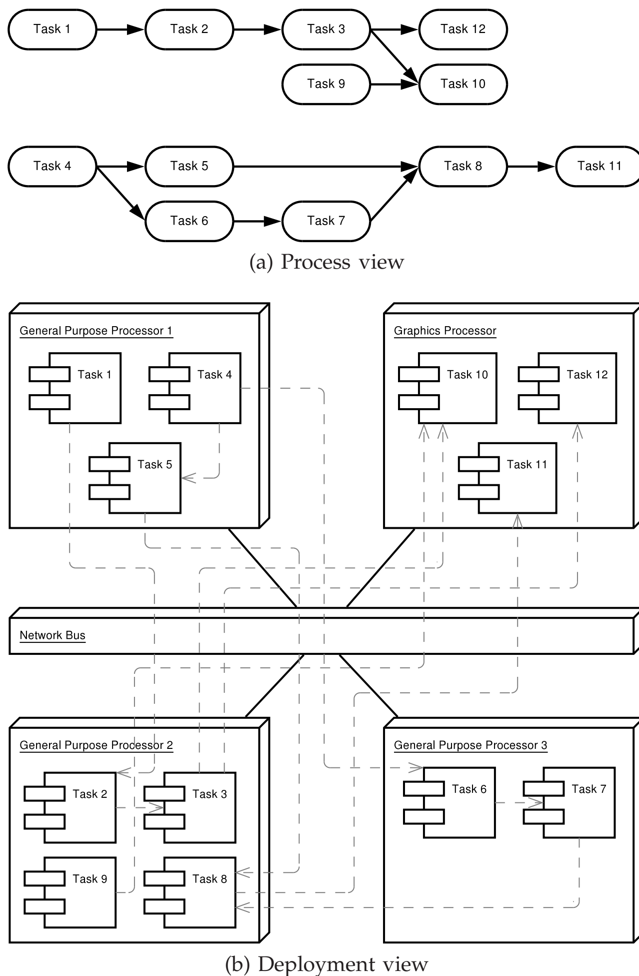
Along with understandability and testability, the modifiability of an architecture is one of the key qualities for determining architecture maintainability [7]. Much of the work on creating flexible software architectures has focused on scenario-based analysis methods [8], [9]. These methods rely on applying potential requirements changes onto an existing architecture model and viewing the responses of metrics for qualities such as performance or dependability. Modifiability is the most prevalent architectural quality targeted for evaluation and improvement by software architecture analysis methods. Even when modifiability cannot be measured directly, the effect of a change scenario on other measurable qualities gives an indication of how difficult it will be to make the change. Therefore, a combination of change scenarios and one or more quality metrics, which aren't a direct measure of flexibility, can give insights into the modifiability of the architecture.

Within real-time system design, the assignment of tasks to processing units and the construction of a suitable schedule is called a *task allocation* problem. It has been identified as one of the most important architectural problems in automotive software engineering [10]. The problem also involves assigning messages to communication buses with tasks and messages collectively referred to as *schedulable objects*. There are well defined quantitative real-time system models which can calculate a response when a scenario is applied to a possible solution. This makes task allocation a good target for automated scenario-based analysis. Depending on the requirements of the system, the response metrics used could include missed deadlines, jitter, volume of network traffic, and load balancing between processors.

The direction of recent research in task allocation has been towards finding solutions which not only meet hard timing requirements but are also of a high quality with respect to other attributes such as reliability [11], [12] and flexibility [13]. This work addresses flexibility

- *Paul Emberson and Iain Bate are with the Department of Computer Science, University of York, York, YO10 5DD, U.K. E-mail: {paul.emberson,iain.bate}@cs.york.ac.uk*

Manuscript received September 01, 2008; revised ...



(a) Process view

(b) Deployment view

Fig. 1. Process and deployment architecture views

by applying scenarios to solutions within a search-based task allocation framework. The emphasis on improving a diverse range of non-functional requirements with automated optimisation and, in particular, search techniques coincides with other work in the field of Search Based Software Engineering (SBSE) [14]. Harman has stated that the robustness of solutions is an important area of future research in SBSE [15]. The work in this paper addresses this concern for the task allocation problem. The algorithms also have the potential to be applied to other scheduling problems in the field of SBSE such as software project management [16].

There are three main contributions made in this paper. The first is a parallel simulated annealing algorithm which searches for solutions for both the current problem to be solved and one or more change scenarios. In particular, it has been designed to remove platform dependent variance and repeatability issues which can cause problems when doing experimentation whilst still taking advantage of modern multicore platforms. The second contribution is an investigation into how different styles of scenarios enhance the flexibility of problems with differing characteristics. This demonstrates the additional insight that search-based software engineering combined with systematic experimental method can bring to a problem. It is important not to give the impression that an algorithm can be taken off the shelf and provide total automation without any additional effort. The algorithm will require configuration of its parameters and must be considered within the wider software engineering process. The third contribution of this work is the process used to configure the algorithm, its use in the initial development of a flexible baseline solution and then how it can evolve this solution during maintenance.

The structure of the remainder of this paper is as follows. Section 2 presents related work in the areas of scenario-based architecture evaluation and real-time task allocation. Section 3 describes the step-by-step process used to configure the algorithm and generate solutions at different stages of the software engineering process. The sections following this carry out this process. Section 4 describes how the set of test cases was generated and how the search algorithms were configured to meet the varying demands of different problems. Section 5 proposes and evaluates an algorithm for finding solutions in the vicinity of an existing solution. Section 6 utilises results from sections 4 and 5 in constructing the algorithm for generating flexible solutions with scenarios. Conclusions are given in section 7.

2 Related Work

Scenarios form a core part of software architecture analysis. An early method is the Scenario Architecture Analysis Method (SAAM) [8] which measures modifiability by counting how many and by how much components are affected by a scenario. An equivalent metric is used in section 6 for assessing how flexible a generated solution is. A survey of architecture analysis methods [17] shows that a large proportion of the work concentrates on the social and organisational issues involved in generating scenarios and undertaking architecture analysis as part of a wider development process. These issues are also relevant to the generation of scenarios for automated methods but are outside the scope of this paper.

Task allocation is an NP-hard problem and much research has looked at how to solve increasingly large and more complex problems using a variety of optimisation techniques. Some examples include simulated annealing [11], [18], [19], solving via conversion to a SAT problem [20], branch and bound [21], particle swarm optimisation [12] and constraint programming [22]. Direct performance comparisons between these works is difficult since the underlying system model and schedulability analysis varies between studies.

Whilst conversion to another class of problem such as SAT opens up the use of many high-quality off the shelf optimisation tools, it also requires the extra step of converting the objective function and constraints to a suitable form. For the purposes of this work, metaheuristic search has the advantage of being able to build implementations of timing analysis and other quality

metrics into the objective function with little additional work. The chosen algorithm for TOAST (Task Ordering and Allocation Search Tool), described in sections 4 to 6, is based upon simulated annealing [23] which has a proven track record in task allocation.

In all previous work on task allocation, there has been little attention paid to the affects of different problem characteristics, such as the ratio of tasks to processors, on the performance of optimisation methods and the flexibility of the solutions generated. Most experiments are limited to variations in problem size based on the number of tasks and processors [11]. Yin et al. [12] acknowledge that the level of inter-communication between tasks, referred to as task interaction density, is an important factor. Extrapolation of results outside of the class of problems experimented upon without a good understanding of their characteristics is an issue not just within the field of task allocation or even SBSE but throughout many areas of experimental research with heuristics [24], [25]. This work pays close attention to these criticisms.

An alternative to scenario-based analysis for improving flexibility at the architecture level is sensitivity analysis. For real-time systems, the limits of the attributes of the system such as task execution time are found using binary search. Racu et al. [6] also take an SBSE approach by combining sensitivity analysis with a multi-objective search optimisation framework. They currently only adjust priority levels of tasks and assume the allocation is already defined. Each sensitivity calculation is effectively evaluating a scenario for particular values of object attributes several times over and is computationally expensive. Scenario-based analysis selects a smaller number of possible changes. The overall computational cost of sensitivity analysis will depend on whether it is applied to each individual object or for larger groups of objects. The TOAST tool includes a coarse grained sensitivity analysis across all objects but primarily as a heuristic to guide the search towards schedulable solutions rather than as an aid to flexibility.

This work is an evolution of previous work by the same authors. Initial work [13] with the same aims of improving flexibility in real-time system design used an algorithm that was inferior both in terms of performance and solution quality. A new parallel simulated annealing algorithm was presented in a paper on finding task allocations for systems with multiple modes [26] which has a similar problem formulation to finding allocations for multiple scenarios. Further improvements to the algorithm are introduced in this paper. Specifically, a second step which required running further searches based on the solutions from each thread has been eliminated by exchanging more information between threads whilst they are running. Furthermore, a new synchronisation mechanism ensures repeatability of experiments and ensures variations in results are not affected by the operating system environment. Neither the process of configuring the algorithm for different problem characteristics nor the relationship between problem characteristics and flexibility has been investigated in any of the authors' previous work.

3 Experimental Process

The technical aspects of this paper are described in three parts:

1) the creation of problems and configuration of the search algorithm to solve them,
2) an evaluation of an algorithm for minimising differences between previous and new solutions,
3) the creation of an algorithm for generating more flexible solutions through the use of scenarios.

These are described over the course of sections 4, 5 and 6 respectively. In order to facilitate understanding, this section gives a broad overview of the process and explains some key concepts and terminology used throughout the rest of this paper.

3.1 Terminology

The meanings of some key phrases as used in the context of this paper are given below.

A *schedulable object* is a task or a message. A *scheduler* is a processor or communications bus.

A *system specification* is a set of schedulable objects and schedulers along with their attributes such as task worst case execution time, message size and communication bus speeds.

A *(system) configuration* is a table of assignments of schedulable objects to schedulers as well as an assignment of priorities to schedulable objects which defines their schedule.

A *task allocation problem* is the problem of finding a configuration for a given system specification in order to meet some criteria. A *solution* to a task allocation problem is any such configuration.

A *scenario* is a system specification for a hypothetical situation which would change the system requirements.

A *system upgrade* or *actual specification change* is a change to the original system specification which is actually needed at this point in time — unlike a scenario, which is a prediction of changes in the future.

A *baseline configuration* is a configuration created at some time in the past. This is usually mentioned in the context of a system upgrade to emphasise the difference between the new configuration which is needed to meet the system upgrade specification and the configuration already in existence.

A *problem characteristic* is any quantifiable measurement that gives information about a problem specification. Examples include the number of tasks and the ratio of messages to tasks.

A *problem class* is a group of problems related by the values of one or more problem characteristics.

A *cost function* is a term for an objective function used in a problem where the aim is to minimise the value of the objective function, i.e. to minimise the cost value.

In the case of the TOAST tool used for the evaluations in this paper, the cost is calculated as a weighted sum of the values of other functions. For example, the proportion of missed deadlines and object sensitivity. Each of these functions is a *cost function component* or *cost component*. The weight associated with each cost function component is a *cost component weighting*.

An *algorithm parameter* is any parameter which affects the behaviour of the search algorithm other than cost component weightings. For example, the cooling rate used in simulated annealing.

Parameter tuning is the activity of finding algorithm parameter values and cost component weightings which are good at solving at problems in a particular class.

3.2 Process

There is an important distinction between the experimental process used to decide whether, for example, one scenario causes the search to generate more flexible solutions than another scenario and the practical steps an engineer would use to generate a flexible baseline configuration. With the former, experimental rigour needs to be emphasised though sensible compromises sometimes need to be made due to limitations in available compute resources and time. For a practitioner, this rigour is less important than achieving acceptable results. However, tuning of the algorithm at each stage of the process as set out below is important in generating high quality solutions. A process which includes these additional steps should be more robust to differences between problems when compared to a process which only involves the running of the algorithm.

The key to any good experimental method is controlling sources of unwanted variance [27] so that they do not mask or distort the effect caused by the factor of interest. For experiments with heuristic search algorithms which have a deterministic objective function, variance in the response can be caused by any of the following:

1) differences in algorithm parameters and cost component weightings or use of a different algorithm
2) differences in the environment in which the algorithm is run which can affect decisions within the algorithm or the response it gives
3) seed values used to initialise pseudo-random number generators
4) solving different problems

Two types of experiment are performed in this paper. The first are parameter tuning experiments where values for the first type of factor are chosen and the factors need to be controlled. The other investigates the effects of different problems and scenarios on the ability to generate a flexible baseline. In this situation, the particular problems are chosen and the first three sources of variance need to be controlled.

The algorithm and its parameters are predetermined by source code and input data and so are easy to control. Environmental differences which affect response include:

- the interaction with other processes and / or processor speeds if the response is time dependent
- scheduling decisions made by the operating system which can affect asynchronous parallel algorithms [28] especially when running on a heterogeneous platform

In this work both of these issues are avoided; responses are based on the number of cost function evaluations and the parallel algorithm in section 6 uses a synchronous communications mechanism to ensure repeatability.

The effect of using different random number seeds is a source of nuisance variance [27] which cannot be eliminated but can be managed with the use of repetitions. Previous work has suggested that finding good algorithm parameters which reduce the mean search running time can also reduce its variance [29]. This is corroborated by results in section 4.

Managing variance due to different problems is an exercise in deciding which characteristics should be used to classify problems in the context of the selected algorithm and its parameters. If the characteristics used to classify problems are not sufficiently specific then there will be too much variance between results for problems in the same class for the classification to be useful. This has to be balanced with the reduced practicality of using problem classes which are too small [25]. In this work, four problem characteristics are considered and, in section 4.4, all possible pairs are tested to see which classification has the biggest impact on performance.

A step by step overview to clarify the experimental process is now given. The section of this paper which covers each step is given in parentheses.

1) Select classes of problem to be studied and generate example problems. (Section 4.2)
2) Tune algorithm parameters and cost component weightings for different classes of problem. Once the search is configured, generate three solutions for each problem. These will act as a control for evaluating increases in flexibility when scenarios are used. (Section 4.4)
3) Generate changes to problem specifications which will act as system upgrades to test flexibility. (Section 5.2.1)
4) Evaluation of algorithms and parameters for minimising change. Using baselines generated in step 2, try to find solutions to system upgrades generated in step 3 which have as few differences as possible from the baselines. (Section 5.2)
5) Generate scenarios for evaluation. (Section 6.2)
6) Using method developed in step 4, evaluate the baselines generated with scenarios with each other and also with the baselines generated in step 2. If the requirements of an upgrade specification can be met with fewer changes, the baseline is said to be more flexible. (Section 6.3)

4 Problem Selection

4.1 System Model

The real-time system model is described here in sufficient detail to comprehend the issues surrounding the generation of example task allocation problems. For a more in depth description readers are referred to the work on distributed scheduling analysis by Palencia and Harbour [30] which is used to calculate response times.

Tasks send messages to one another which creates dependencies between tasks. The structure of these dependencies take the form of directed acyclic graphs with tasks represented as nodes and messages represented as edges. Each independent task graph is a *transaction*. Within the system specification, each task is assigned a worst case execution time (WCET) and a period. The utilisation of a task is its WCET divided by its period. The total system utilisation is the sum of all task utilisations. Each task also has a deadline. Timing requirements are specified in terms of calculated response times being less than or equal to deadlines. In this work, deadlines are set equal to periods. Messages have a maximum size attribute. When a message is assigned to a communications bus, a worst case communication time is calculated based on the speed and latency characteristics of the bus. All messages and tasks within the same transaction have the same period.

The hardware model consists of a number of processors connected together with communications buses. A bus is attached to each processor for messages sent between tasks on the same processor. The attributes of these buses can be set to simulate what in reality may be an alternative mechanism such as shared memory communication. As discussed in section 3 the aim is to understand the effects of a small number of characteristics and control others. For this reason rather than technical ones, the example hardware platform is fairly conservative. All processors are connected together with a single broadcast network and are homogeneous in the sense that a task's WCET does not depend on the processor it is allocated to.

4.2 Test Case Generation

A test case generator tool is used to generate system specifications whose characteristics are dependent upon parameters given to the tool. The number of tasks is used as a unit of size for the problem. Where relevant, other parameters are specified as proportions of the number of tasks. The number of processors is calculated from the tasks per processor parameter. The number of messages sent between tasks is set by the messages per task parameter. Tasks are randomly grouped into transactions according to the tasks per transaction parameter.

Periods for each transaction are chosen from a given range such that different orders of magnitude are sampled from with equal probability as suggested by Davis et al. [31]. They are rounded to a whole value with a granularity of $1/10$ of the lower limit of the period range.

TABLE 1
Varied problem characteristics

Problem Specification	Utilisation per proc.	Tasks per proc.	Messages per task	Max period / min period
01	40	5	1	10 (5.90)
02	65	5	1	10 (5.00)
03	40	8	1	10 (2.3)
04	65	8	1	10 (6.4)
05	40	5	2	10 (5.9)
06	65	5	2	10 (8.6)
07	40	8	2	10 (9.6)
08	65	8	2	10 (6.0)
09	40	5	1	1000 (30.8)
10	65	5	1	1000 (62.0)
11	40	8	1	1000 (39.3)
12	65	8	1	1000 (81.3)
13	40	5	2	1000 (110.6)
14	65	5	2	1000 (367.8)
15	40	8	2	1000 (14.9)
16	65	8	2	1000 (127.2)

TABLE 2
Fixed problem characteristics

Characteristic	Fixed Value
number of tasks	40
processor connectivity	1 (single broadcast network)
network bandwidth	2048
network latency	0
message size	30000
tasks per transaction	$25\% * 40 = 10$ tasks (implies 4 transactions)
transaction length	$40\% * 10 = 4$

WCETs are set so that the overall utilisation per processor meets a value set by a parameter to the problem generation tool and utilisation is distributed evenly between transactions.

The final stage of system generation is to connect tasks together with messages to form the task graphs for each transaction. A parameter, which is specified as a percentage as the number of tasks in a transaction, decides the average transaction length defined as the longest path through the task graph. Another parameter sets the ratio of messages to tasks.

For the experiments conducted in this paper, 16 problems were generated using all combinations of four problem characteristics set at two different levels. These characteristics and their values are shown in table 1. The characteristic shown in the final column is the upper limit of the range of periods divided by the lower limit. The ranges used were $[1000, 10000]$ and $[1000, 1000000]$. The values given in parentheses is the actual maximum period value divided by the minimum period within that particular test case. The levels of utilisation leave scope for system upgrades which increase utilisation in later experiments.

Table 2 gives a list of problem characteristics which are set at fixed values for all problems. The size of problem

TABLE 3
Number of unscheduled objects with random search

Problem	Tasks	Mess.	Total	Problem	Tasks	Mess.	Total
01	5	0	5	09	2	0	2
02	17	5	22	10	9	0	9
03	1	0	1	11	4	0	4
04	9	0	9	12	8	1	9
05	5	2	7	13	10	10	20
06	18	13	31	14	16	19	35
07	6	6	12	15	10	2	12
08	13	9	22	16	15	8	23

in terms of the number of tasks, messages and processors is similar to that used by Zheng et al. in an evaluation of a subsystem of an experimental vehicle [32].

4.3 Search Algorithm

4.3.1 Random Search

For a quick assessment of problem complexities, a random search was performed. Each run of the search tried 400000 random solutions and stored the solution with the lowest number objects not scheduled. For each problem, the best result from three repetitions is shown in table 3. No solution met all constraints but the number of unscheduled objects gives a crude estimate of problem difficulty. The results suggest that problems with higher utilisation and more messages per task are more challenging.

4.3.2 Simulated Annealing

To find solutions which meet all requirements, a guided search is needed. The algorithm used is simulated annealing [23]. Pseudo-code for this algorithm is listed in figure 2. At each iteration step, ssize configurations are sampled from the neighbourhood. The configuration in the sample with the lowest cost is then put forward to the simulated annealing acceptance criteria. The algorithm has three configurable algorithm parameters: the initial temperature, sample size and the number of inner loop iterations. The cooling factor is fixed at 0.99 but the rate of cooling can be changed via the inner loop iterations parameter. The stopping condition function depends on the needs of each experiment. The neighbourhood of a solution is the union of the set of all single task and message reallocations and the set of priority reorderings which change the priority of a single task or message to a new slot position.

The other parameters which need to be tuned are the cost component weightings. The list of cost component functions are given in table 4. The full formulae and details of these functions are given in previous work [33]. The purpose of each is now briefly described.

- g_1 is a function which gives the proportion of deadlines missed. When this reaches 0, it shows that the solution meets all timing requirements.
- g_2 penalises tasks which need to communicate but are assigned to separate clusters of processors. For all the test problems in this work, all processors are interconnected so it will always evaluate to 0 but is included for consistency with previous work.
- g_3 penalises any task or message which is allocated in such a way so as not to be able to receive its input or send its output. This could happen if a message is assigned to the intra-processor bus for the processor the source task is allocated to but the destination task is on a different processor.
- g_4 checks for any schedulable objects involved in the same transaction but whose priority ordering does not match their precedence ordering.
- g_5 performs sensitivity analysis on the execution/communication times of tasks and messages.
- g_6 measures the variance in utilisation between processors. Reducing this improves the balance of load between processors.
- g_7 is a metric for whether schedulable objects involved in the same transaction are allocated to the same schedulers. This reduces communication overheads.
- g_8 has a similar purpose as g_7 but concentrates on pairs of tasks which are adjacent to each other in a transaction.
- g_9 adds a penalty for any schedulers which are more than 100% utilised since no feasible solution can include such a configuration.

```
input: init ;                                /* the initial solution */
input: ssize ;                               /* sample size parameter */
input: inittemp ;                            /* initial temperature */
input: maxinner ;                            /* inner loop iterations */
input: stop ;                                /* stopping condition function */
begin
  curconf = init;   bestconf = init;
  curcost = costfn(curconf) ;
  bestcost = curcost;
  repeat
    i = 0 ;
    repeat
      (newconf, newcost) = samplenei(curconf, ssize) ;
      if newcost < bestcost then
        bestconf = newconf;   bestcost = newcost;
      end
      if randuniform() < exp(-(newcost - curcost)/t) then
        curconf = newconf;   curcost = newcost;
      end
      i = i + 1 ;
    until i == maxinner or stop() ;
    t = t * 0.99 ;
  until stop() ;
end
```

Fig. 2. Simulated annealing algorithm

All cost function components produce a value in the range $[0,1]$. The weighted sum of these component values is normalised using the sum of the weights so that all cost function values are also in the range $[0,1]$. This is important in reducing interactions between weightings and algorithm parameters which makes parameter setting easier. In particular, the temperature based acceptance criteria which uses the absolute change in cost value would not be affected if all cost component weightings

TABLE 4
Cost function components

Function (as in [33])	Name
g_1	Missed Deadlines
g_2	Unreachable Tasks
g_3	Unconnected input / output
g_4	Incompatible Priority/Precedence Order
g_5	All object sensitivity
g_6	Processor load balance
g_7	Transaction object grouping
g_8	Separated communicating tasks
g_9	Over utilised schedulers

were multiplied through by a constant.

These component functions are designed to act as good heuristics for particular classes of problem. For example, grouping objects in the same transaction together will generally create a good solution. However, for situations where this is not possible, weighting this component too highly will severely reduce performance.

4.4 Parameter Tuning

The cost component weightings and search algorithm parameters must be tuned to work with the set of test case problems described in section 4.2. The reasons for tuning parameters is twofold. Firstly, it will improve performance so experiments can be run in less time. More importantly, optimising the search algorithm according to different problem characteristics should reduce the variance between runs and reduce the likelihood of the algorithm failing to find a solution when one exists.

In this work, the cost component weightings and search algorithm parameters are tuned separately. This makes the total number of experiments, which has an exponential relationship with the number of parameters, more manageable. It is a potentially sub-optimal approach since there is no guarantee that the two parameter sets are independent. The aim, however, is to find a set of parameters which performs well across the problem test set and optimality is not a requirement.

The method used for tuning parameters is the same as described by Poulding et al. [29] who also gives further references on experimental methodology. The method can be condensed into three stages:

1) run experiments with different input levels,
2) use regression techniques to create a model which maps the input levels to the response variable,
3) use optimisation to find the inputs which minimise the response according to the generated model.

The model of the algorithm behaviour is an approximation based on interpolating between parameter levels used in experiments. It can also be inaccurate by overfitting to effects caused by sources of nuisance variance. Nevertheless, it will systematically and efficiently find a good design point in the parameter space [29] to which small adjustments can be made if necessary.

4.4.1 Tuning Component Weightings

Since the cost function is normalised, they can sum to any constant value which is chosen arbitrarily at 30000 for these experiments. A suitable experimental design where relative proportions rather than absolute values need to be found is a mixture model simplex lattice design [29]. Using this design for each of the 9 factors at 3 levels requires 165 runs per problem. To cover all 16 problems with 3 repetitions for each combination of weightings requires 7920 runs. Within each repetition of the experiment each factor level will be assessed 55 times, counteracting sources of nuisance variance such as different random seed values. With each experiment having the potential to run for several hours if no acceptable solution is found, the White Rose Grid at York [34] was utilised connected to other clusters of computers using BOINC [35].

The stopping condition for these experiments was when all timing requirements were met ($g_1 = 0$) or the maximum number of evaluations, set to 300000, was reached. This value, known as a *censoring level*, is chosen to balance having a high number of successful runs, which improves the accuracy of the fitted model, and the time required to complete. The search algorithm parameters were set to values found by trial-and-error in preliminary work. Initial temperature was set to 0.003, sample size to 1 and max inner loop iterations to 5000. The response variable was the number of evaluations recorded when the search stopped. A model was created using survival regression based techniques which take account of runs which terminate without finding a valid solution. Inputs which minimised the value of this model were found using LINDO Optimization [36] software.

Since the aim is to take account of problem characteristics, as well as generating a model to best fit data across all 16 problems, the problems were classified in different ways using all possible combinations of two of the four problem characteristics. For each of these six classification schemes, a model was fitted to each of the four groupings within it and minimised. The mean of the minimal response estimated by the model was used as a guide to the quality of the classification scheme. The classification scheme which gave the best response was to form problem groups by their utilisation per processor and tasks per processor characteristics. Both the weightings found by fitting a model over all 16 problems and those found by classifying the problems were tested by running the search algorithm with the appropriate weightings with 3 repetitions on each problem. For these 48 runs, the maximum number of evaluations was increased to 400000 to reduce failed runs and the search algorithm parameters were left as before.

The results of these tests are shown in table 5. For the single set of weightings 10 runs failed to find a solution within the allowed number of evaluations so a precise value for the mean cannot be calculated. The second line of the table shows that using separate weightings

TABLE 5
Tuned parameter tests

Parameters	Mean Evaluations	Failures
Single weightings set	$>$ 100945	10
Classified weightings	51137	0
As above with specific weights for prob. 06, prob. 14	36416	0

TABLE 6
Problem dependent component weightings

Problem(s)	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9
01,05,09,13	16434	3	3	4173	3	3	9375	3	3
02,10	14578	3	3	2840	5199	3335	3	4036	3
03,07,11,15	16264	3	3	5030	3	3	8688	3	3
04,08,12,16	16413	3	3	4630	3	3	8939	3	3
06	12000	3	7364	3	4076	2605	3	2446	1500
14	17773	3	3000	3	2513	3	3	3804	2898

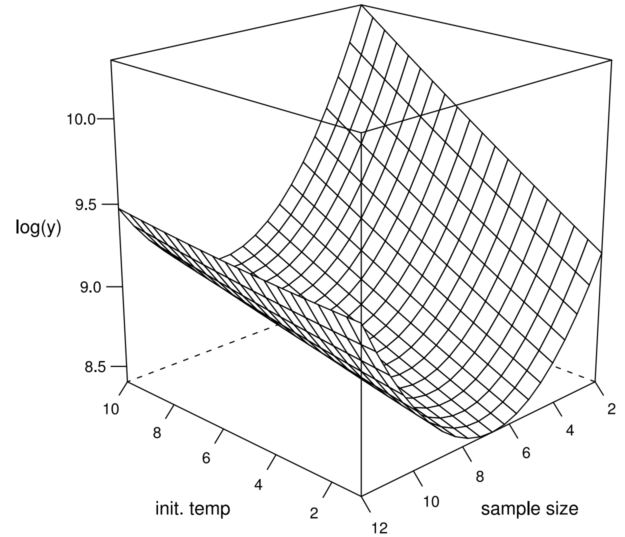


Fig. 3. Algorithm parameter model

for different problem classes substantially improved the performance of the algorithm.

These tests showed that two problems, 06 and 14, were needing an order of magnitude more evaluations to find solutions for. These problems both have a combination of high processor utilisation, low number of tasks per processor and a high message to task ratio. The fact that problems with only two of these characteristics appear to be substantially easier is an example of the insight that can be gained from systematic experimentation with search algorithms. After reviewing the data used to fit the models for these two problems it was found that from the 990 runs covering these two problems, only 1 run found a solution within the censoring level. To get a more varied response for different weightings, a constrained mixture model was used so that only inputs where the weighting for missed deadlines was greater than 12000 were considered. This experiment had 271 successful runs from the 990. The final row of table 5 shows the results from this experiment and that the mean number of evaluations was further reduced.

The final table of weightings to be used for future experiments is given in table 6. A look at this table shows that the results are sensible. Component weightings for component g_1, upon which a stopping criterion is based, has consistently high values and weightings for g_2, which was noted in section 4.3.2 not to be useful for these problems, has the lowest permitted value.

4.4.2 Tuning Algorithm Parameters

Once component weights had been decided, the other search algorithm parameters could be tuned. A full factorial experimental design was performed for the three parameters at 3 levels for all problems, again with 3 repetitions. This lead to a further $3^3 * 16 * 3 = 1296$ experimental runs. The results of fitting and optimising the model gave an initial temperature of 0.001, with 500 inner loop iterations and a sample size of 7. A slice of the model, with inner loop iterations fixed at 500 is shown in figure 3 shows the minimum for the sample size close to 7. Raising the value higher causes the search to be more aggressive in finding lower cost solutions but at the expense of it getting stuck in local optima. Lower values help the search avoid becoming trapped in local optima but takes longer to find a solution. The interaction between sample size and initial temperature is also revealed by the plot. Lowering the initial temperature and increasing the sample size both cause the search to be more exploitative as opposed to explorative. At high sample sizes, the model shows that better results will be obtained using a higher initial temperature to avoid getting stuck in local optima. At low sample sizes, the graph indicates better performance at lower initial temperatures which reduce search exploration.

In reality, when the proposed algorithm parameters were tested, they were found to be overly aggressive with many runs becoming stuck in local optima and failing to find a solution. Modelling according to problem classes produced similar results. This demonstrates that the fitted model is not totally representative of the algorithm behaviour. This is most likely due to limitations in the shape of the model and sources of nuisance variance. It did however, act as a good guide towards using more aggressive search parameters than were originally being used for component weighting tuning experiments. Within just a couple of trial-and-error runs, it was found that setting the initial temperature to 0.002, the sample size to 5 and the inner loop iterations to 5000, much improved results could be obtained.

The result of testing these parameters versus those used in the test for the last result in table 5 is shown in figure 4. Despite the improvements made by selecting specific weightings for problems 06 and 14, these problems were still requiring a much larger number of evaluations and more crucially, as can be seen in figure 4,

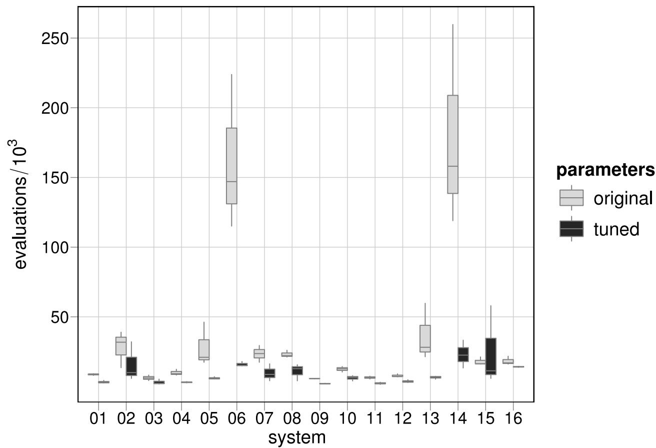


Fig. 4. Original vs tuned algorithm parameters

a much larger degree of variance between repetitions. By tuning the search algorithm parameters, both the number of evaluations and variance between repetitions was dramatically reduced. The only compromise was an increase in variance between repetitions for problem 15 though the mean number of evaluations for this problem still decreased. The mean number of evaluations required over all 48 test runs was reduced by over 73% from 36416 to 9567.

5 Minimising Changes To Solutions

This section presents the algorithm used for finding a solution that is similar to a baseline yet still meets the needs of an upgrade specification. The flexibility of a particular baseline will be measured in terms of the number of modifications required to transform the baseline into the solution found by this algorithm for different upgrade specifications.

5.1 Configuration Difference Cost Components

Two additional cost function components are introduced to penalise solutions which are further away from a baseline. They use metrics for measuring allocation changes and priority changes, for which full formulae can be found in previous work [26]. Each one takes two configurations as inputs and outputs a value indicating the difference between them.

- g_{tac} is the component used to measure task allocation changes. It is a measure of the proportion of tasks which are not allocated to same processor in both configurations.
- g_{tpc} is the component used to measure task priority changes. It only has non-zero values for tasks which are allocated to the same processor in both configurations since comparing priorities on different processors has no meaning. Only changes to order rather than absolute priority value are penalised since a relabelling of priorities which maintain the same order will have no effect on the schedule.

These two components can be combined into an overall measure of change using a normalised weighted sum such as that given below.

$$g_{mod} = \frac{10 \cdot g_{tac} + g_{tpc}}{11} \tag{1}$$

The cost a priority change incurs versus the cost of an allocation change will depend on the motivation for reducing change and how these costs relate to real world financial costs. Since a priority change value for a particular task is non-zero only if its allocation is the same, allocation changes should have a higher weighting else the search may favour different allocations to reduce the priority change metric. The 10 to 1 ratio used above is suitable to demonstrate the use of this function.

5.1.1 Combining With Timing Constraint Components

The cost function that was used throughout section 4 is relabelled as g_{sched} and then this value is combined with the result of g_{mod}, defined in equation (1), to produce an overall cost value. Grouping components together to form these higher level components simplifies balancing the effects of components which guide the search towards solutions which meet timing constraints and those components which penalise changes.

Influenced by the importance of component g_1 in finding solutions which meet all schedulability constraints, the following equation was proposed to balance the two sets of components.

$$f = \frac{w_{sched} g_{sched} + (1 - g_1)^b g_{mod}}{w_{sched} + (1 - g_1)^b} \tag{2}$$

where w_{sched} changes the balance between meeting schedulability requirements and the need to reduce the number of modifications. As the proportion of objects with missed deadlines, g_1, decreases, more emphasis is put on minimising change. The value of b alters the bias towards the constraints.

5.2 Algorithm Evaluation

Experiments were conducted to evaluate the performance of a search algorithm using (2) as a cost function. The steps taken were:

1) Generate upgrade specifications to use as a test set
2) Ensure existing parameters are suitable for finding solutions to upgrade specifications and a solution can be found for each one
3) Evaluate values for the parameters w_{sched} and b in equation (2) for their ability to find solutions with minimal changes from a baseline.

5.2.1 Upgrade Specification Generation

The upgrades which were generated were limited to utilisation increases where the increase was spread evenly between transactions. The levels of utilisation increase are given in table 7. Initially the utilisation increases used were the same for all problems. However, when

TABLE 7
Utilisation increase levels

Utilisation Increase Level	Actual Utilisation Increase	
	Low Starting Utilisation	High Starting Utilisation
1	4%	2%
2	8%	4%
3	12%	6%
4	16%	8%
5	20%	10%

large utilisation increases were applied to some problems which already had a higher starting utilisation, it was not possible to find solutions for the upgrade specifications. Therefore, smaller utilisation increases were made for the upgrade specifications associated with problems having a higher initial utilisation as listed in table 1. For each utilisation increase level and each problem, 3 separate upgrades were generated to be used for repetitions in later experiments. There were $5 * 16 * 3 = 240$ upgrade specifications generated in total.

5.2.2 Upgrade Validation

Upgrade specifications were checked to be feasible by searching for a solution to each of them without regard for any existing baseline. The cost component weightings used were taken from the results for higher utilisation problems in table 6. Given the additional challenge posed by the utilisation increases, the sample size was reduced from 5 to 3 to allow the search to be more explorative. These weightings and parameters were successful in solving all of the upgrade specifications.

As a reference for future experiments, the solutions found while checking the feasibility of upgrade specifications in section 5.2.1 had a mean proportion of allocation changes from their baselines of 0.842. Given that there are be several valid solutions for each problem, this high value is not surprising.

5.2.3 Parameter Evaluation

An evaluation was performed for a range of values of w_{sched} with $b = 0$ and with $b = 40$. Experiments with $b = 0$ are labelled as "fixed" since the balance between schedulability constraints and minimising change between solutions does not change throughout the course of the search. Experiments with $b = 40$ are labelled as "adaptive". The problem to be solved is to find a value for w_{sched} such that the solution which the search produces has as few modifications as possible but with all schedulability requirements met.

For all experiments where modifications from a baseline needed to be reduced, the search algorithm uses the baseline as an initial solution. For the experiments in this section, the solutions generated from the final set of tests described in section 4.4 were used as baselines. Since the upgraded specifications have much in common with the original problem specifications, the previous baseline is often a good partial solution with many schedulability constraints already met. Rather than allow the search to be over explorative, parameters were modified so that it would perform a thorough search near the baseline. The sample size was maintained at 3 but the initial temperature was dropped to 0.001. The stopping criteria were set so that 300000 evaluations were performed or the search stopped immediately if the initial solution met all schedulability requirements with 0 modifications.

The weightings used were the same as those in the initial experiment which checked the solvability of the upgrade specifications but with one important change: the weighting for component g_1 which penalises missed deadlines was increased for the following reasons. Components $g_2, \ldots, g_9$ are aimed at guiding the search towards a solution which meets all timing constraints ($g_1 = 0$). However, even after all timing constraints are met, they continue to be in tension with cost components for minimising change making this objective harder to achieve. Since the search problem is made easier by using a baseline configurations as starting point, the utility of these guiding components is reduced. Increasing the weighting of g_1 by a factor of 6 reduced their influence by a suitable amount.

The number of times that a schedulable solution was not found for the fixed and adaptive weighting methods are shown in table 8. There was a common subset of 200 problems solved in every experiment. A comparison of the number of allocation changes required for each method applied to this subset is shown in figure 5. This

TABLE 8
Failures for different weighting levels

w_{sched}	**Fixed** $(b = 0)$	**Adaptive** $(b = 40)$
2.5	36	25
5	14	11
20	6	3
40	5	4
80	2	2
200	2	2

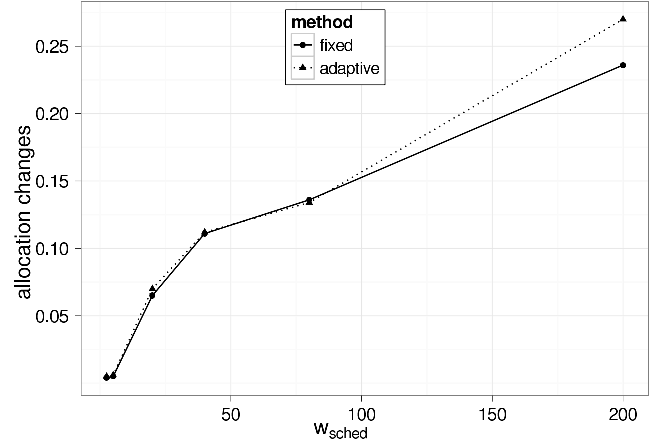


Fig. 5. Comparison of methods for solved problems only

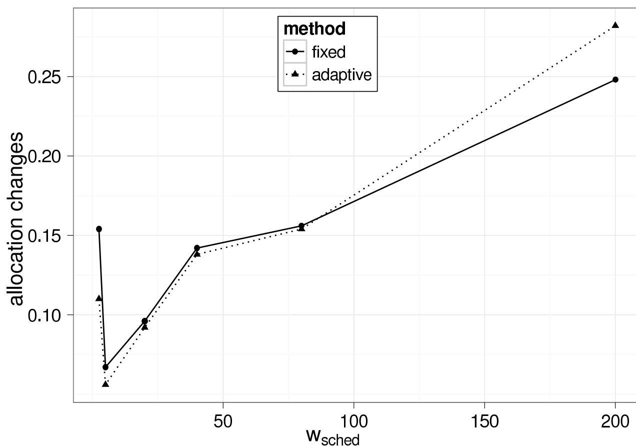


Fig. 6. Comparison of methods including all problems

graph shows that at low to medium values of w_{sched} the performance is nearly identical. However, when weightings are set to favour schedulability over change more strongly, the fixed weighting method outperforms the adaptive method. As the value of w_{sched} increases, the number of changes increase. This is because, even when solutions with fewer changes exist, if there is little pull towards the original baseline, the search will tend to move away until it finds a solution with all constraints met and become trapped in this good local optimum.

To be able to compare the flexibility of baselines in future experiments the results from table 8 and figure 5 need to be combined. This raises the question of how to treat a failed result. It is known from the validation experiment described in section 5.2.2 that a solution exists for all upgrade specifications. However, each failure indicates that the search was not able to find a solution near the given baseline. One possibility is to substitute the result for the number of changes between the baseline and the solution found in the experiment in section 5.2.2. However, this is a somewhat arbitrary data point and using these solutions compared to an alternative set of solutions could have a significant impact on results. Instead, a value of 1 which is the worst case proportion of allocation changes was used. The results of treating failures in this way are shown in figure 6. At low values of w_{sched}, the adaptive method performs better since it appears to be more robust at finding a schedulable solution with small amounts of change when weightings are set strongly in favour of reducing modifications. However as the value of w_{sched} increases, it becomes easier for the search to find schedulable solutions and the fixed weighting method achieves better results.

Figure 6 shows that the best compromise of having few failed results and minimising change occurs when w_{sched} is 5. A more in depth analysis of the two methods was performed for this value. From the 240 runs, a feasible solution was found by both methods 224 times. The number of allocation changes differed on only 17 of these 224 runs and the mean proportion of changes was 0.0087 for both methods so there is little to separate them on this basis. When failures are included, the mean proportion of allocation changes for the fixed method is 0.0671 and 0.0563 for the adaptive method. A paired Wilcox exact test of these two sets of data returned a p-value was 0.370. This means that, when w_{sched} is 5, the null hypothesis that these two methods are equivalent cannot be rejected. As a comparison, for a w_{sched} value of 2.5, at which point both methods performed worse, the p-value from the equivalent test was 0.02244. In this situation, the null hypothesis can be rejected at the 95% confidence level, so the evidence that the adaptive method provides benefit is stronger.

TABLE 9
Selected w_{sched} values

Problem	w_{sched}	Problem	w_{sched}
01	2.5	09	2.5
02	20	10	5
03	2.5	11	2.5
04	2.5	12	2.5
05	2.5	13	2.5
06	2.5	14	20
07	5	15	2.5
08	2.5	16	2.5

Finally, this raises the question of which values of w_{sched} to use and with which algorithm. Although experiments such as those run here can suggest values to be used as a starting point, it is recommended that values should be determined on a case by case basis for real world use since much will depend on the baseline. Since invalid solutions will be rejected in this case, there is no reason not to use the simpler fixed weighting method. For the purposes of experimentation, however, which tests many baselines over several problems, values were derived from these results which provided the best guide available. Some failed runs were expected in the results and so the adaptive method is preferred. Assuming that baselines generated using scenarios will be at least as flexible as one generated arbitrarily, low values of w_{sched} were favoured apart from for problems which had a high proportion of failures. The selected values are given in table 9. The results corresponding to these values using the adaptive method produce a mean allocation change value of 0.036. This is a further improvement on selecting a w_{sched} value of 5 for all experiments which produced a mean of 0.0563 as previously stated.

6 Searching With Scenarios

The search algorithm for generating baseline solutions, presented in section 6.1, is based on the assumption that if a solution for a problem specification meets all or nearly all of the requirements of a scenario, the solution produced will be more flexible with respect to changes of a similar nature to the scenario. This assumption is tested in sections 6.2 and 6.3.

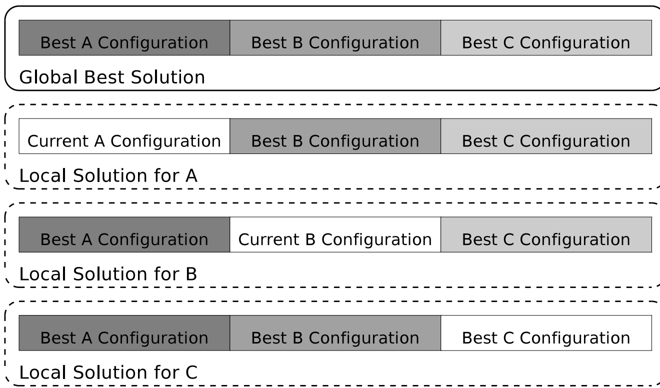


Fig. 7. Solution representation for multi-problem search

6.1 Flexible Baseline Algorithm

6.1.1 Requirements

The functional and non-functional requirements driving the design of the flexible baseline algorithm are motivated by the needs of the broader process outlined in section 3. These are:

1) to allow there to be differences between solutions produced for each scenario and the problem specification and possibly for a valid solution to a scenario to never be found
2) to minimise the differences between the final solutions for each of the input specifications
3) to be scalable in terms of the additional computation required for evaluating multiple scenarios
4) to produce repeatable results for a fixed set of inputs (including a random number generator seed)

The first requirement was accomplished by assigning the main problem specification and each of the scenario specifications to separate instances of the search procedure. This means that multiple solutions can be produced and if one search fails to find a solution, others will not automatically fail. Minimisation of the differences between solutions is performed by exchanging details of the best solution found so far between each instance of the search and trying to pull solutions towards each other. The third requirement is attained by running each of these search procedures in parallel. This allows the algorithm to scale as long as the hardware platform provides sufficient processing cores for the number of scenarios used. To be able to reproduce results, a synchronisation mechanism overlays the parallel search threads so that the sequence of interactions between the searches does not depend on the time at which those interactions occur. This mechanism is described later in this section.

6.1.2 Representation And Cost Evaluation

Figure 7 shows the representation of the solution which would be held by three search threads, labelled A, B and C, solving three specifications, i.e. the main problem specification and two scenarios. Each search thread operates on a section of the solution specific to its specification. The other parts of the solution are filled with the best solutions found by other search threads for their section of the solution. The global best solution is a configuration made up of the best solutions found for each specification. This is the solution returned by the search when all search threads have terminated.

The cost function used by each search thread operates on the whole of its locally stored solution, not just the part pertaining to the specification it has been assigned to. Within each thread, the schedulability of each system and differences between each section of the configuration are assessed using equation (2) from section 5.2 and then the mean is taken across all systems to obtain a final cost value. By penalising solutions with more differences between configurations each search is encouraged to find solutions similar to the others.

As with work in the previous section the correct balance of achieving schedulable solutions and minimising change must be found. In this case the difference between solutions for each specification rather than the difference to a fixed baseline is being reduced. The scenarios act as a stress on the solution to the main problem and the emphasis on minimising change is less critical than before. With reference to equation (2), a value of 10 was chosen for w_{sched} parameter and b is set to 40 once again. On a small number of occasions, this emphasis on obtaining similar solutions was too great to meet all timing requirements of the main problem specification and the baseline was regenerated with w_{sched} set to 40.

6.1.3 Synchronisation Mechanism

The strategy for synchronising threads is to exchange information whenever a search thread finds a solution better than the currently held global best solution. For a given set of inputs, a search will find its first improvement on the initial solution after the same number of evaluations on every run. Therefore, this can be used as a basis for creating a repeatable sequence of interactions.

When each search thread is first spawned from the main application, each thread locally stores a copy of all its state variables which can affect the decisions it takes. For simulated annealing, this includes the current temperature and inner loop count. In order to achieve repeatability, the state of the pseudo-random number generator must also be controlled. The implementation in TOAST uses a separate instance of the Mersenne Twister [37] in each thread. These are seeded by the main application when each thread is initialised.

The steps taken to synchronise the searches are best explained by way of the example shown in figure 8. At the top of the diagram, the searches have just passed a synchronisation point, which could be the start of the search. The next synchronisation point is based on which search finds a solution better than the global best solution in the fewest evaluations since the last synchronisation point. If more than one search finds a better solution after the same number of evaluations,

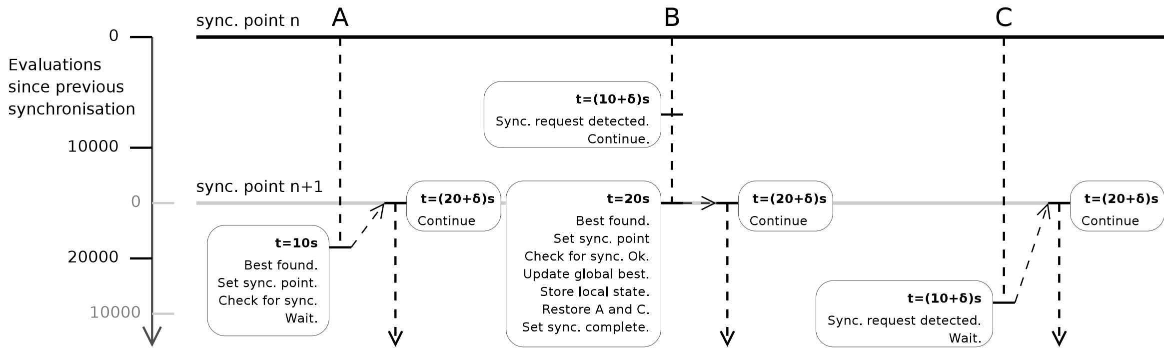


Fig. 8. Synchronisation mechanism

then the one with the lower cost is preferred. After this, ties are broken based on a deterministic ordering of the search threads. Figure 8 shows the following steps.

1) After 10 seconds, search A finds a new best solution and sets a global data flag requesting synchronisation. It now waits for other searches to reach the same number of evaluations.
2) Just after 10 seconds searches B and C read this synchronisation request. Search B hasn't performed enough evaluations and continues. Search C is past the synchronisation point so stops and waits.
3) After 20 seconds, search B still hasn't reached the synchronisation point but has found a new best solution. It sets the synchronisation point to a lower number of evaluations. All searches are now at or past the synchronisation point. Search B is the only search exactly at the synchronisation point so it updates the global best with its solution. It then makes a copy of its local state. The other searches restore their local state from the last copy made. B sets a flag indicating synchronisation is complete and continues.
4) Searches A and C continue using the new best solution found by B as their starting point.

An additional optimisation which has been found to greatly improve performance is to try the solution for one specification on another. Since scenarios are usually a modified version of the original problem, high quality solutions are often common to both problems. After each synchronisation, instead of sampling from the usual neighbourhood of solutions, the algorithm uses the sections of the configuration for other specifications as its possible next steps. For example, thread A would evaluate solutions for B and C on its own portion of the configuration. These can be accepted or rejected using the usual simulated annealing acceptance criteria.

6.2 Scenario Generation

Scenarios were generated in a random but controlled way so that the tasks which were modified were different to the tasks changed in the upgrade specifications described in 5.2.1. The flexibility of baselines generated with scenarios was assessed using these upgrade specifications and the techniques developed throughout section 5. The characteristics of the scenarios are shown in table 10. Each row of the table corresponds to the scenarios used for generating a set of 3 baselines.

TABLE 10
Scenarios

Scenario	Num scenarios	%age tasks changed	Utilisation increase
noscen	no scenarios used	–	–
scen1	1 scenario	40%	4%
scen2	1 scenario	40%	11%
scen3	1 scenario	40%	18%
scen4	1 scenario	40%	25%
scen5	1 scenario	20%	25%
scen6	1 scenario	60%	25%
scen7	3 scenarios	40%	11%

The scenario labels have the following meanings. *noscen* corresponds to the results from the previous section where baselines were generated without the use of scenarios. Scenarios *scen1*, . . . , *scen4* were generated by randomly selecting an equal number of tasks from each transaction so that, in total, 40% of the systems' tasks were changed to achieve a range of different utilisation increase levels. Certain problems were selected for further evaluation with scenarios *scen5*, *scen6* and *scen7*. Scenarios *scen5* and *scen6* change different proportions of tasks and *scen7* uses three separate scenarios, each generated in the same manner as *scen2*.

6.3 Scenario Evaluation Results

The same weightings which were used to generate previous baselines, i.e. those tuned for the characteristics of each problem and listed in table 6, are also used to generate baselines with scenarios.

Results showing the combined flexibility of all baselines generated with the first four scenarios and without any scenarios are shown in figure 9. The first point of note is that the baselines generated without any scenario performs better than may be expected. This can be explained by the fact that the parameters used to minimise change between a baseline solution and one for the upgrade specifications were specifically tuned for these

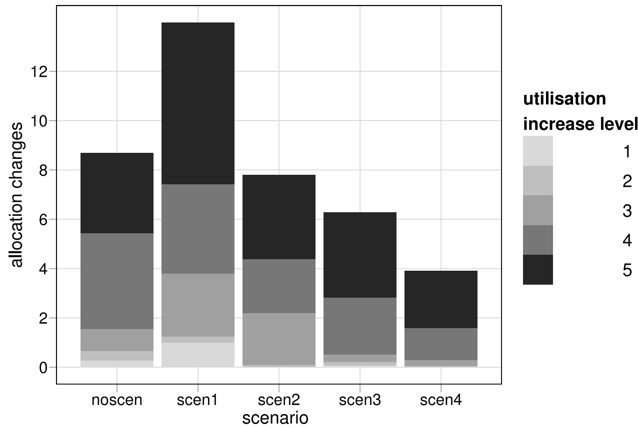


Fig. 9. Changes required to meet upgrade requirements

baselines in section 5.2.3. This puts the new baselines generated with scenarios at a slight disadvantage. For the other solutions, increasing the size of utilisation increase in the scenario gradually improves the flexibility of the system. The shading in each bar in figure 9 shows the proportion of the number of changes attributed to each upgrade. For *scen4*, nearly all allocation changes for upgrades with lower level utilisation increases have been eliminated and allocation changes required for higher level utilisation increases have been reduced.

Figure 10 breaks down the results of figure 9 by problem characteristic. This shows a very clear pattern. The problems which have the least flexible solutions by far are those with a high utilisation and a low number of tasks per processor. In a problem with a lower number of tasks per processor, each task is using a larger chunk of utilisation on average and so it is more difficult to fully utilise the available resources of each processor. One problem in particular requires more changes but there is insufficient data within a single cell to draw conclusions with respect to its characteristics.

As was described in section 5.1, priority changes can only be compared once allocation changes have been removed. Figure 11 shows the priority changes required to meet the upgrade specifications but only for the 12 systems which were shown in figure 10 to have negligible allocation changes. This graph once again shows that using scenarios which stress the solution with larger utilisation increases reduce the number of changes required to perform an upgrade.

For the four systems which had a high per processor utilisation and low number of tasks per processor, further baselines were created using scenarios described in the final three rows of table 10. The flexibility of the baselines generated for just these four systems is compared in figure 12. Scenario *scen5* increased utilisation per processor by the same amount as *scen4* but concentrated the change amongst fewer tasks. In general, if it were known which tasks would change, then a more targeted scenario makes sense but for the style of upgrade tested here, these scenarios do not perform well. Scenario *scen6* does the opposite, spreading the utilisation increase over a larger number of tasks. This style of scenario actually generates the most flexible baselines for these upgrades showing that diluting the utilisation increase over more tasks does not have a negative effect. Finally, *scen7* represents a combination of 3 scenarios each with the same characteristics as *scen2*. It has been shown that increasing the amount of stress a scenario applies via a larger utilisation increase improves flexibility. An alternative way of increasing the stress is to use multiple scenarios with smaller utilisation increases. A comparison between baselines generated with *scen2* and *scen7* validates this statement.

7 Conclusion

The task allocation problem is an important part of the architecture selection process for distributed real-time systems and affects the system's flexibility with regard to requirements changes and upgrades. A three stage process was given for generating flexible solutions to task allocation problems and then taking advantage of the flexibility. The first two stages, used during system design, were tuning the algorithm for the problems of interest and generating baseline solutions using scenarios to enhance flexibility. The third stage, upgrading the baseline with as few changes as possible, is used during the maintenance part of an engineering process.

A simulated annealing based search algorithm was tuned for a set of problems. It was found that classifying problems according to certain characteristics can greatly improve algorithm performance. The best characteristics for classifying problems were found to be the utilisation per processor and tasks per processor.

The method of generating baselines used a parallel search algorithm where threads collaborated to produce similar solutions for a current problem specification and potential upgrade scenarios. By using a synchronisation mechanism based on the thread which found a new best solution with the fewest evaluations since the last synchronisation point, experiments could be conducted in a repeatable manner on a variety of platforms.

The use of scenarios was found to allow upgrades to be performed with fewer changes to baselines even though the predicted changes were different from the upgrades tested. This was true of all problems evaluated. Using single scenarios which contained larger changes or multiple scenarios with smaller changes enhanced flexibility more than a single scenario with made only a small change to the original problem.

Four problem characteristics were studied throughout this paper. It was found that problems with a combination of high processor utilisation, a low number of tasks per processor and a high message to task ratio required the most effort to solve efficiently. In terms of producing flexible baseline solutions, the interaction between processor utilisation and number of tasks per processor

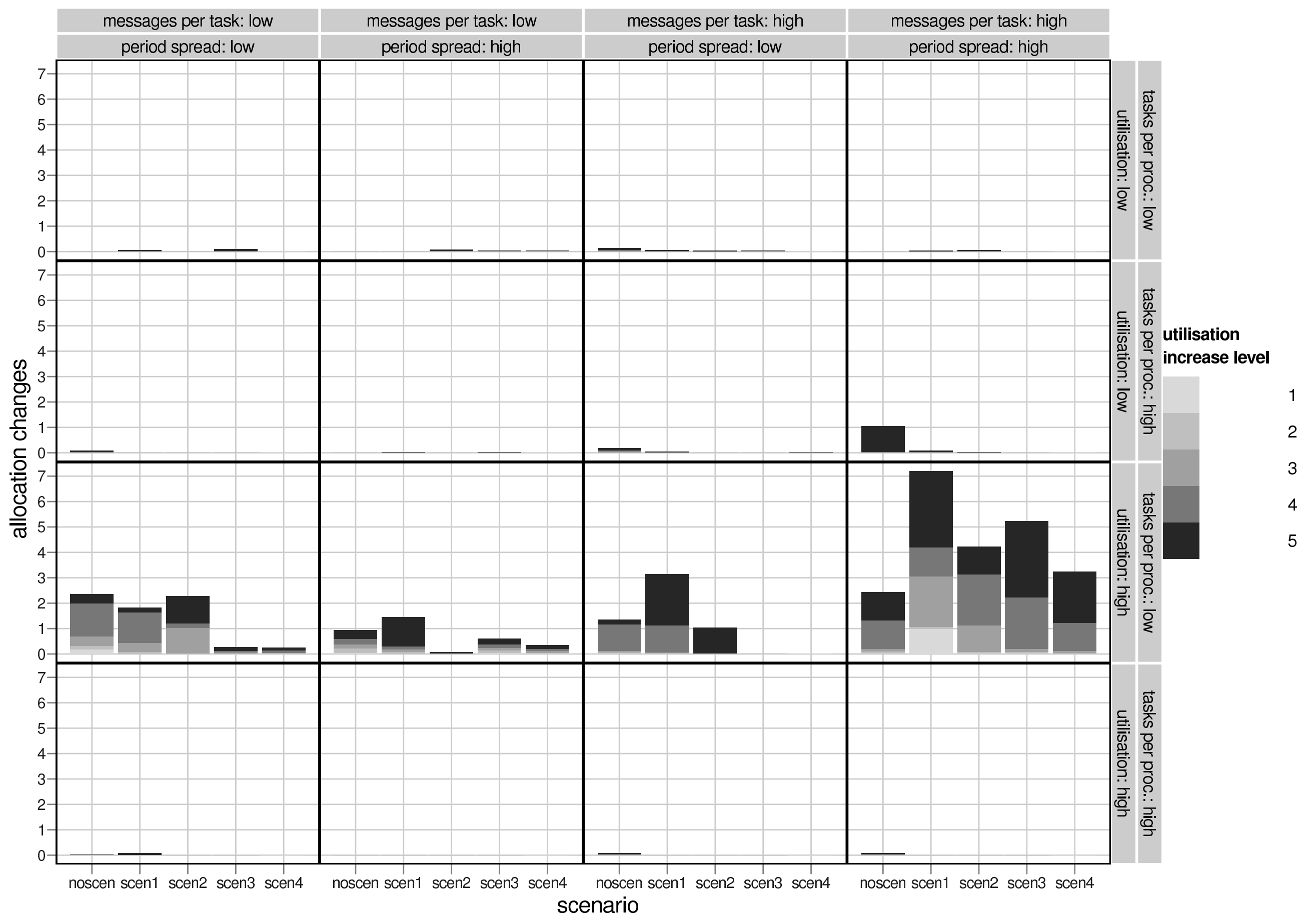


Fig. 10. Changes required for upgrades for each problem class

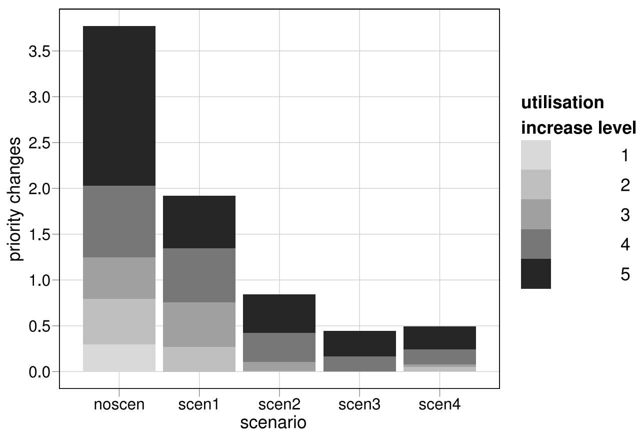


Fig. 11. Priority changes required for upgrades

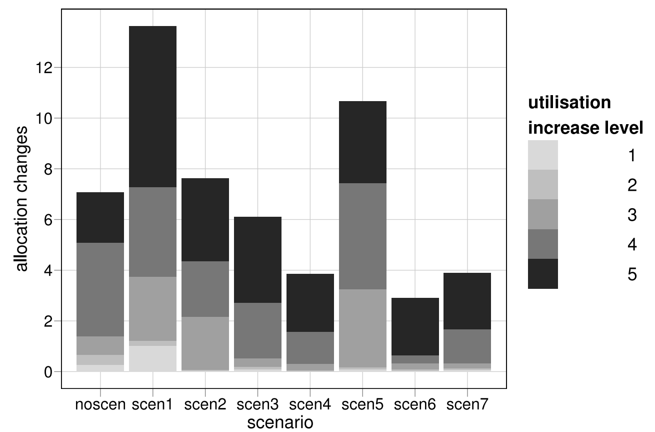


Fig. 12. Results of additional scenario evaluations

characteristics proved to be the most challenging. The range of task periods within a system was the least important of the four characteristics.

ACKNOWLEDGMENTS

This work is funded by the Software Engineering By Automated Search (SEBASE) program, EPSRC Grant EP/D050618/1. We would also like to thank Simon Poulding for his advice on experimental methods.

REFERENCES

[1] L. J. Bass, M. Klein, and F. Bachmann, "Quality attribute design primitives and the attribute driven design method," in *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, 2002, pp. 169–186.

[2] P. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995.

[3] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 2005, pp. 109–120.

[4] P. Koopman, "Embedded system design issues (the rest of the story)," in *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*. IEEE Computer Society, 1996.

[5] L. Sha, "Real-time virtual machines for avionics software porting and development," in *Real-Time and Embedded Computing Systems and Applications*, 2004, vol. 2968, pp. 123–135.

[6] R. Racu, A. Hamann, and R. Ernst, "Automotive system optimization using sensitivity analysis," in *Embedded System Design: Topics, Techniques and Trends*, 2007, vol. 231, pp. 57–70.

[7] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *ICSE '76: Proceedings of the 2nd International Conference on Software engineering*, 1976, pp. 592–605.

[8] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd, "SAAM: A method for analyzing the properties of software architectures," in *International Conference on Software Engineering*, 1994, pp. 81–90.

[9] R. Kazman, M. Klein, and P. Clements, "Evaluating software architectures for real-time systems," *Annals of Software Engineering*, vol. 7, no. 1-4, pp. 71–93, 1999.

[10] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *IEEE Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[11] G. Attiya and Y. Hamam, "Task allocation for maximizing reliability of distributed systems: A simulated annealing approach," *Journal of Parallel and Distributed Computing*, vol. 66, no. 10, pp. 1259–1266, October 2006.

[12] P.-Y. Yin, S.-S. Yu, P.-P. Wang, and Y.-T. Wang, "Multi-objective task allocation in distributed computing systems by hybrid particle swarm optimization," *Applied Mathematics and Computation*, vol. 184, no. 2, pp. 407–420, January 2007.

[13] I. Bate and P. Emberson, "Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems," in *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006, pp. 221–230.

[14] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, Lumkin, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings - Software*, vol. 150, no. 3, pp. 161–175, June 2003.

[15] M. Harman, "The current state and future of search based software engineering," *FOSE '07: Proceedings of Future of Software Engineering*, pp. 342–357, 2007.

[16] E. Alba and F. J. Chicano, "Software project management with GAs," *Information Sciences*, vol. 177, no. 11, pp. 2380–2401, June 2007.

[17] M. A. Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation methods," in *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference*, 2004, pp. 309–318.

[18] K. Tindell, A. Burns, and A. Wellings, "Allocating hard real-time tasks: An NP-hard problem made easy," *Real-Time Systems*, vol. 4, no. 2, pp. 145–165, 1992.

[19] J. Beck and D. Siewiorek, "Simulated annealing applied to multicomputer task allocation and processor specification," in *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, 1996, pp. 232–239.

[20] A. Metzner and C. Herde, "RTSAT– an optimal and efficient approach to the task allocation problem in distributed architectures," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 2006, pp. 147–158.

[21] D. T. Peng, K. Shin, and T. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *Software Engineering*, vol. 23, no. 12, pp. 745–758, 1997.

[22] P.-E. Hladik, H. Cambazard, A.-M. Deplanche, and N. Jussien, "Solving a real-time allocation problem with constraint programming," *Journal of Systems and Software*, vol. 81, no. 1, pp. 132–149, January 2008.

[23] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[24] J. N. Hooker, "Testing heuristics: We have it all wrong," *Journal of Heuristics*, vol. 1, pp. 33–42, 1995.

[25] A. E. Eiben and M. Jelasity, "A critical note on experimental research methodology in ec," in *CEC '02: Proceedings of the 2002 Congress on Evolutionary Computation*, 2002, pp. 582–587.

[26] P. Emberson and I. Bate, "Minimising task migration and priority changes in mode transitions," in *RTAS '07: Proceedings of the 13th Real Time and Embedded Technology and Applications Symposium*, 2007, pp. 158–167.

[27] D. C. Montgomery, *Design and Analysis of Experiments*, 6th ed. Wiley, December 2004.

[28] E. Alba and J. M. Troya, "Analyzing synchronous and asynchronous parallel distributed genetic algorithms," *Future Generation Computer Systems*, vol. 17, no. 4, pp. 451–465, January 2000.

[29] S. Poulding, P. Emberson, I. Bate, and J. Clark, "An efficient experimental methodology for configuring search-based design algorithms," in *Proceedings of 10th IEEE High Assurance System Engineering Symposium*, 2007, pp. 53–62.

[30] J. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Proceedings of the IEEE Real-Time Systems Symposium*, 1998, pp. 26–37.

[31] R. I. Davis, A. Zabos, and A. Burns, "Efficient exact schedulability tests for fixed priority real-time systems," *IEEE Trans. Comput.*, vol. 57, no. 9, pp. 1261–1276, 2008.

[32] W. Zheng, Q. Zhu, M. Di Natale, and A. S. Vincentelli, "Definition of task allocation and priority assignment in hard real-time distributed systems," in *Proceedings of 28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 161–170.

[33] P. Emberson and I. Bate, "Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems," in *RTSS '08: Proceedings of the 29th IEEE International Real-Time Systems Symposium*, December 2008, accepted to appear.

[34] "White Rose Grid at York," http://www.wrg.york.ac.uk, accessed August, 2008.

[35] "Berkeley Open Infrastructure for Network Computing," http://boinc.berkeley.edu/, accessed August, 2008.

[36] "LINDO Systems," http://www.lindo.com/, accessed August, 2008.

[37] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, January 1998.

Paul Emberson is a research associate in the Real-Time Systems group at the University of York. His research interests include local search methods, software engineering processes and software architecture analysis. He previously worked as a professional software engineer in the field of mobile telecommunications.

Iain Bate is a lecturer in Real-Time Systems. His research interests include scheduling and timing analysis, design and analysis of safety-critical systems, and systems engineering. He is the Editor-in-Chief of the Journal of Systems Architecture, a frequent member of programme committees for distinguished international conferences, and a member of the Scientific Advisory Board for the Progress research centre at Maladarlen University, Sweden, which specialises in Component-Based Software Engineering.