Extending A Task Allocation Algorithm For Graceful Degradation Of Real-Time Distributed Embedded Systems [*]

Paul Emberson and Iain Bate
Department of Computer Science
University of York
York, YO10 5DD
{paul.emberson, iain.bate}@cs.york.ac.uk

Abstract

Previous research which has considered task allocation and fault-tolerance together has concentrated on constructing schedules which accommodate a fixed number of redundant tasks. Often, all faults are treated as being equally severe. There is little work which combines task allocation with architectural level fault-tolerance issues such as the number of replicas to use and how they should be configured, both of which are tackled by this work. An accepted method for assessing the impact of a combination of faults is to build a system utility model which can be used to assess how the system degrades when components fail. The key challenge addressed here is how to design objective functions based on a utility model which can be incorporated into a search algorithm in order to optimise fault-tolerance properties. Other issues such as how to extend the local search neighbourhood and balance objectives with schedulability constraints are also discussed.

1 Introduction

The reliability of computer based systems has become more important as their use as key components of critical systems has escalated. Examples of such systems are automotive and avionic control system applications which are both safety and mission critical. These systems are also real-time, embedded and often distributed systems. In the context of this work, real-time systems are considered to be hard real-time where every task must meet its deadline. Due to the severe consequences of these systems failing, it is essential that they can provide an acceptable level of service and system safety even when components have a permanent fault. A system which can continue to run safely in the presence of faults but with a reduced level of service is referred to as a gracefully degrading system [10].

There are a number of strategies which can be used to allow a system to continue to function in the presence of faults. The most prevalent is replication. A system with replication runs redundant versions of some or all of its tasks and will usually require additional processors to do so. Systems using replication can use cold, warm or hot back up [7] strategies. The difference lies in whether all task replicas are always running and being kept up to date to allow an almost instantaneous fail-over or if they are left dormant until needed. This work assumes a hot backup strategy, commonly used in avionics [21].

The choice between dynamic or static redundancy also distinguishes fault-tolerance mechanisms [7]. Static redundancy uses redundant tasks to mask faults whereas a dynamically redundant system waits for the system to begin to error or give an indication an error is about to occur before taking steps to recover. This paper is only concerned with static redundancy which is commonly used in critical systems [21] and is also complementary to the off-line nature of the task allocation problem.

The task allocation problem is an exercise in deciding how to assign tasks to processors so that timing requirements are met [14]. However, real-time scheduling not only involves allocating tasks to processors but also the assignment of scheduling attributes, such as priorities. This means that for systems comprising a significant number of tasks, only automated methods can feasibly be used to find solutions which meet all constraints [20]. Real world task allocation problems are often complex and impose many constraints which the solution must adhere to. This has led to continuing research in developing new solution methods which consider a larger range of constraints and objectives. Examples include heterogeneous processors [22] and design flexibility [2]. It has been noted that distributed scheduling and fault-tolerance are not orthogonal [11] since fault-tolerance schemes have a timing overhead. Therefore, it is natural to extend task allocation algorithms with a fault-tolerance objective.

In this paper, the TOAST task allocation tool, developed by the authors and previously described in [2, 5], is extended to support fault-tolerance. The algorithm is capa-

[*] This work is funded by the Software Engineering By Automated Search (SEBASE) program, EPSRC Grant EP/D050618/1.

ble of solving task allocation problems for heterogeneous systems with precedence constraints. By allowing the algorithm to vary the number of task replicas and their allocations, it is able to alter the fault-tolerance properties of the system. There are a number of challenges in doing this. When task and message replicas are added to the system, they must also be allocated and scheduled, increasing the size of the problem. In addition to this, the optimisation algorithm now has an additional axis of variation in determining how many replicas should be used. Pure task allocation for hard real-time systems is a constraint satisfaction problem where any solution in which all deadlines are met is acceptable. In order to assess fault-tolerance qualities of a system, a fault-tolerance objective function needs to be developed. This is non-trivial since there are many ways in which a system can fail. In particular, a system can degrade in different ways for increasing numbers of faults and the preferred degradation behaviour will differ according to system requirements. Therefore, objective functions which can be adapted to different situations are needed.

The chosen method of evaluation of fault-tolerance quality uses a measure of system utility designed by Shelton [19]. This metric does not assume that the system has failed after all versions of a task have failed but instead provides a measure which describes the reduced level of service. Shelton provides some examples of systems which exhibit this behaviour. One of these is a braking system which has partially failed. Its usefulness depends on which of the four wheel brakes are working and the way in which the system degrades will depend on the order in which they fail. This example is included as part of the evaluation in section 6.

Fault-tolerant systems are designed with the aim of being t-fault-tolerant and/or having a certain mean time between failure (MTBF)[18]. A t-fault-tolerant system is one that can withstand t faults before failure. Objective functions are built upon the system utility measure to support both the t-fault-tolerant and MTBF paradigms.

To summarise, the primary aims of this paper are:

- Extend the search algorithm in the TOAST tool so that the number of task replicas can be varied in addition to configuring their allocation and scheduling attributes.
- Investigate the additional complexity of including task replicas and also the balance between fault-tolerance objectives and schedulability constraints.
- Creation of objective functions which use Shelton's system utility model and are suitable for a range of system fault-tolerance requirements

The structure of this paper is as follows. Section 2 describes related research. Sections 3 and 4 describe the existing system model and task allocation search algorithm. The extensions to the previous work which deal with fault-tolerance issues are given in section 5. These are evaluated in section 6 before conclusions are drawn in section 7.

2 Related Work

Previous work combining automated real-time task allocation with fault-tolerance has tended to concentrate on modifying the scheduling analysis to accommodate the overheads needed for fault tolerance [6, 11, 15]. Often the work will be specific to a computational model such as a static cyclic schedule and optimise this schedule so that the system can handle a single processor failure [6, 15]. The emphasis of this work is in making higher level architectural decisions, in particular the number of replicas to use and where they will be allocated. Previous work on static redundancy and task allocation has been concerned with a fixed number of processor faults, and often just a single fault. To our knowledge, this is the first piece of work which deals with task allocation in the context of a gracefully degrading system and has the ability to differentiate systems which can handle the same numbers of faults before complete failure but degrade differently.

Oh and Son [11] discuss the need to consider schedulability and fault-tolerance simultaneously. They prove that finding a schedule to handle a single processor failure is NP-hard and give an algorithm to solve this problem. The system model is non-preemptive and does not include precedence constraints.

Girault et al. [6] give an approach similar to ours in that they adapt an algorithm which generates distributed static schedules to handle processor failures with fail-stop behaviour. However, the number of replicas is pre-determined and allocated by making a copy of task sets on existing processors to redundant processors. Qin and Hong [15] build on the work of Girault et al. The system model includes precedence constraints and a more heterogeneous environment. They introduce the concept of performability which is a combination of schedulability and fault-tolerance. They allow for reliability heterogeneity by including a failure rate for processors in their model.

Echtle and Eusgeld [4] also use search, specifically a genetic algorithm, to find fault-tolerant system designs. However, the approach is not aimed at real-time systems and schedulability is not taken into account. Of some interest is the fitness function used by the search algorithm. It considers how combinations of faults lead to failures and in this sense has some commonality with the utility model introduced in section 5. Bicking et al. [3] take a similar approach to [4], also using a genetic algorithm, but once again the system model is not targeted at real-time.

Kany and Madsen have written a high quality study on design optimisation for fault-tolerant real-time systems [8]. They use the WCDOPS++ [16] scheduling model which a more recent form of the WCDOPS [12] algorithm used in

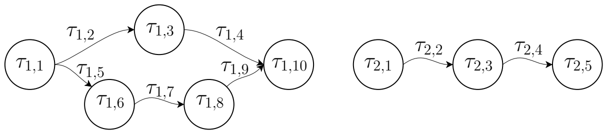

Figure 1. Example of two transactions

TOAST. Their fault-tolerant design choice is the decision between trying to re-execute or a single task replication to mask faults. This is in addition to selecting an allocation. Task priorities are pre-defined, unlike TOAST which searches for a priority ordering in addition to allocation and fault-tolerance decisions. The cost function is based upon the difference between task response times and task deadlines under a number of fault scenarios. There is no notion of how system utility reduces over increasing numbers of failures.

In [10], as part of the RoSES (Robust Self-configuring Embedded Systems) project, Nace et al. outline a framework for providing graceful degradation using a combination of feature subsets, utility model and task allocation. However the main results from the project provide only the utility model [19], and it is generally assumed that each feature subset is resident on its own processor. In this paper we combine our previous work with that of Shelton [19] to contribute to the overall framework envisioned by Nace et al.

3 System Model

The distributed real-time system model used in this work is as follows. Each task has a worst case execution time (WCET), period and deadline. Precedence relations are formed between tasks by way of sending and receiving messages. The dependencies form directed acyclic graphs called transactions as shown in figure 1. Each message has a size, a period and a deadline. The period is set to be the same as the task sending the message and the deadline is set to be the same as the deadline of the task receiving it.

The hardware architecture is modelled as a set of processors connected with network links. Each link has a communication speed and latency which is used along with the size of a message to determine the worst case communication time for a message. All processing nodes have a link to themselves. This models any communication overhead for intra-processor communication though this can be set to be negligible. Figure 2 shows four processing nodes connected with two network links and an additional four links for intra-processor communication. The example hardware platform in figure 2 shows that it is not necessary for all processors to be directly connected to each other so the availability of networks for message allocation is dependent upon where the sending and receiving tasks are allocated.

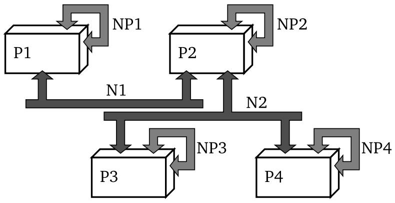

Figure 2. Example hardware architecture

The cost function and search algorithm, described in section 4, base design decisions on response times generated by a form of distributed scheduling analysis. The analysis used by toast is the WCDOPS (Worst Case Dynamic Offsets with Priority Schemes) method by Palencia and Harbour [12] which extends fixed priority scheduling analysis to distributed systems.

4 Search Algorithm

The TOAST optimisation algorithm for finding solutions to task allocation problems has evolved over the course of previous research [2, 5, 13]. This section describes this algorithm so that it can be built upon to address fault tolerance issues in the following section. Particular emphasis is placed on the objective function which has undergone some simplifications since that presented in [13].

In this work the task allocation problem is cast as a minimisation problem where the number of missed deadlines must be reduced. The core of the search algorithm is an implementation of the simulated annealing [9] meta-heuristic. This is a local search algorithm with the ability to probabilistically escape local minima. To make a local search meta-heuristic appropriate to a particular problem, it is necessary to instantiate it with a neighbourhood function and a cost function.

4.1 Neighbourhood Function

The neighbourhood function, which describes how a new candidate solution is generated by mutating the current solution, acts on the configurable attributes of tasks and messages. The neighbourhood function, with equal probability, selects between an allocation change and a priority change. An allocation change randomly picks a schedulable object (task or message) and moves it to a new processor or network as appropriate. Message allocations are restricted to those networks connected to a processor of at least one of the source and destination tasks. Priority changes involve selecting a new priority for a schedulable object and shuffling priorities of other objects up or down as required so that all priorities remain unique.

4.2 Cost Function

The cost function value is calculated as a weighted sum of the results of lower level functions. These lower level functions which make up the cost function are *cost function* *components*. In addition to a component for counting how many deadlines are missed, other components are used as heuristics to improve the performance of the search. Each component returns a value in the range $[0, 1]$ and the overall function is normalised by the sum of the weights so that it too returns a value in the same range. This is shown in equation (1).

$$f(\omega) = \frac{\sum_i w_i g_i(\omega)}{\sum_i w_i} \tag{1}$$

Weightings must be set in order to optimise the algorithm for performance and solution quality. The best values will be dependent upon the problem or set of problems to be solved. The technique used for setting weightings is the systematic experimental method for setting search parameters given by Poulding et al. in [13].

The sets of tasks, messages, processors and network links are denoted as $\mathcal{T}$, $\mathcal{M}$, $\mathcal{P}$, and $\mathcal{N}$ respectively. $\mathcal{S} = \mathcal{T} \cup \mathcal{M}$ is the set of schedulable objects. The notation for the number in a set X is $|X|$. Directly dependent (DD) tasks are a pair of tasks which have a message sent between them. Indirectly dependent (ID) tasks appear in the same transaction but are not necessarily adjacent in the task graph. Functions labelled g_i are the cost function components and functions labelled h_i are helper functions for g_i.

The first component assesses the number of unschedulable objects, by comparing the calculated response time for each schedulable object, R_τ, with its deadline D_τ.

$$h_1(\tau) = 1 \text{ if } R_\tau > D_\tau \text{ else } 0 \tag{2}$$

$$g_1 = \frac{1}{|\mathcal{S}|} \sum_{\tau \in \mathcal{S}} h_1(\tau) \tag{3}$$

The following component counts how many DD tasks are allocated to processors not connected by a bus. Let al map a task to its allocated processor and V map a processor to the set of processors to which it is connected.

$$c(\tau, \upsilon) = 1 \text{ if } V(\mathrm{al}(\tau)) \cap V(\mathrm{al}(\upsilon)) = \emptyset \text{ else } 0 \tag{4}$$

$$g_2 = \frac{1}{|\mathcal{M}|} \sum_{\rho \in \mathcal{M}} c(\mathrm{src}(\rho), \mathrm{dest}(\rho)) \tag{5}$$

The next component penalises objects which cannot receive their input or send their output due to their allocation. Firstly, two functions are defined which give the input and output of a schedulable object. The definitions are conditional on whether the object is a task or message. The functions src and dest give the sending and receiving task of a message.

$$\mathrm{in}(\tau) = \begin{cases} \{\rho \in \mathcal{M} : \mathrm{dest}(\rho) = \tau\} \text{ if } \tau \in \mathcal{T} \\ \{\mathrm{src}(\tau)\} \text{ if } \tau \in \mathcal{M} \end{cases} \tag{6}$$

$$\mathrm{out}(\tau) = \begin{cases} \{\rho \in \mathcal{M} : \mathrm{src}(\rho) = \tau\} \text{ if } \tau \in \mathcal{T} \\ \{\mathrm{dest}(\tau)\} \text{ if } \tau \in \mathcal{M} \end{cases} \tag{7}$$

$$\begin{aligned} h_3(\tau) = |\{\upsilon \in \mathrm{out}(\tau) : \mathrm{al}(\upsilon) \notin V(\mathrm{al}(\tau))\}| + \\ |\{\upsilon \in \mathrm{in}(\tau) : \mathrm{al}(\upsilon) \notin V(\mathrm{al}(\tau))\}| \end{aligned} \tag{8}$$

$$g_3 = \frac{\sum_{\tau \in \mathcal{S}} h_3(\tau)}{\sum_{\tau \in \mathcal{S}} \big[|\mathrm{in}(\tau)| + |\mathrm{out}(\tau)|\big]} \tag{9}$$

The following component measures priority assignment which are incompatible with precedence constraints. $\mathrm{pre}(\tau)$ is the set of all objects preceding τ and $\mathrm{post}(\tau)$ is the set of all objects that follow τ.

$$g_4 = \frac{\sum_{\tau \in \mathcal{S}} |\{\upsilon \in \mathrm{post}(\tau) \text{ and } P_\upsilon < P_\tau\}|}{\sum_{\tau \in \mathcal{S}} |\mathrm{post}(\tau)|} \tag{10}$$

A sensitivity component calculates the largest factor by which execution/communication times can be scaled and for the system to be schedulable. This value can be found using a binary search and will be less than 1 while the system is unschedulable.

$$g_5 = e^{-\lambda \mathrm{SCAL}_{\mathcal{S}}} \tag{11}$$

where $\mathrm{SCAL}_{\mathcal{S}}$ is the largest value of a scaling factor s such that the system is schedulable when the WCETs, C_i of objects in the set $\mathcal{S}$ are set to $\lfloor sC_i \rfloor$.

A load balancing component is based upon the variance of the utilisations of processors:

$$g_6 = \sqrt{\tfrac{\sum_X (U_X - \mu^2)}{(|\mathcal{P}|-1)\mu^2 + (U-\mu)^2}} \tag{12}$$

where U_i is the utilisation of processor i, and μ is the mean utilisation.

Grouping communicating objects onto the same scheduler reduces overheads. Let the set of all transactions be TRANS and the set of schedulable objects contained in transaction r be TRANS_r. V_r is the set of tasks in TRANS_r. For each $\tau_i \in V_r$, the number of tasks allocated to the same scheduler as τ_i and also in V_r is a_{ri}. A grouping value and its maximum for each transaction is:

$$\gamma_r = |V_r| - \sum_{i=0}^{|V_r|-1} \frac{a_{ri}}{|V_r|} \qquad \gamma_{r_{\mathrm{MAX}}} = \tfrac{|V_r|(|\mathcal{P}|-1)}{|\mathcal{P}|}$$

The theoretical maximum value occurs when tasks are equally spread among processors and $a_{ri} = |V_r|/|\mathcal{P}|$ for all i. Similar formulae can be defined for messages with W_r being all messages in TRANS_r. Using $\gamma_{r_{\mathrm{MAX}}}$ to normalise γ_r and then summing over all transactions, the component formula is

$$\begin{aligned} g_7 = \tfrac{|\mathcal{P}|}{2|\mathrm{TRANS}|(|\mathcal{P}|-1)} \left[|\mathrm{TRANS}| - \sum\nolimits_r \sum\nolimits_i \frac{a_{ri}}{|V_r|^2}\right] \\ + \tfrac{|\mathcal{N}|}{2|\mathrm{TRANS}|(|\mathcal{N}|-1)} \left[|\mathrm{TRANS}| - \sum\nolimits_r \sum\nolimits_i \frac{a_{ri}}{|W_r|^2}\right] \end{aligned} \tag{13}$$

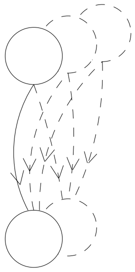

Figure 3. Messages sent by task replicas

Component g_7 groups ID tasks but messages between DD tasks may still go back and forth between processors. The following penalises messages sent between DD tasks on different processors.

$$g_8 = \frac{1}{|\mathcal{M}|} |\{\rho \in \mathcal{M} : \mathrm{al}(\mathrm{src}(\rho)) \neq \mathrm{al}(\mathrm{dest}(\rho))\}| \tag{14}$$

An additional penalty is given to solutions containing processors with over 100% utilisation.

$$g_9 = \frac{|\{l \in \mathcal{P} \cup \mathcal{N} : U_l > 100\}|}{|\mathcal{P} \cup \mathcal{N}|} \tag{15}$$

5 Extensions For Fault Tolerance

In order to embrace fault-tolerance as a core part of the automated architecture design process, it is necessary to extend the system model, including the computational model, as well as both the neighbourhood and cost functions used in the search algorithm.

5.1 Extensions To System Model

An extra attribute is added to the problem specification which indicates the maximum number of replicas for each task. Since a hot backup strategy is being assumed, if a message passes between two tasks, then an equivalent message must be passed between all replicas of those tasks. Figure 3 shows a task with two replicas which sends a message to a task with a single replica. If at least one version of each task is on a processor which has not failed, then the functionality provided by these tasks will still be present. When multiple versions of a task are present in the system other tasks will receive messages from each of the replicas. In this paper, the computational model assumes that the first message received is used. Other models are feasible. For instance, tasks could wait for all messages of working replicas to arrive in order to compare results.

5.2 Extensions To Neighbourhood

In addition to configuring the original set of tasks and messages in the system, allocations and priorities must be found for all replica tasks and their messages. For this purpose, replicas are simply treated as extra schedulable objects in the system so no changes need to be implemented. However, the replicas do increase the size of the neighbourhood making the possible solution space much larger. In order for the algorithm to decide how many task replicas should be used, a third axis of variation (in addition to allocations and priorities) must be added. The neighbourhood function is given the option of enabling and disabling replica tasks. A disabled task is effectively removed from the system for the purposes of evaluating the system's schedulability and fault-tolerance qualities. When a task is disabled, all of the messages it sends and received are also disabled as these will not be needed as part of a design with fewer replicas. On completion of the search, any disabled objects present in the solution judged to be the best will not be included in the output.

During the evaluation it was found that it is better to favour enabling a disabled task as opposed to disabling a currently working one. Therefore, the function chooses to change the status of a currently working task, chosen at random, with probability 0.1 and changes the status of a currently disabled replica with probability 0.9. It should be noted that this is another parameter whose optimal value will be dependent upon the problem to be solved.

5.3 The System Utility Metric

The system utility metric is taken from Shelton [19]. This section explains how it is implemented efficiently and used as part of the search objective function.

For a system, where some components may have failed, the *utility* of the system is a measure of the functionality that the system is still providing. Calculating such a value is difficult since failures are generally not independent. For example, consider an automotive braking system with a brake on each of the four wheels. The loss in utility can be considered equivalent for any single brake failure. If two brakes fail, configurations where both failed brakes are on the same side of the car can be considered to have less utility since this will cause the car to swerve when braking.

Assuming a fail-fast, fail-stop [17] model, where each component can only have a status of *working* or *failed*, there are 2^N failure configurations for a system with N components. Assigning a utility value to every one of these configurations is not a scalable solution. To overcome this, Shelton [19] developed a method which uses hierarchical decomposition to reduce the number of utility values required. Shelton's method takes advantage of the existing design decomposition already present in the system to group individual system components into feature subsets. These feature subsets containing system components are grouped to form higher level feature subsets.

Figure 4, a simplified version of Shelton's diagram in [19], shows four feature subsets for the left front brake subsystem of an automotive braking system. The utility of each feature subset depends on the status of the components in it

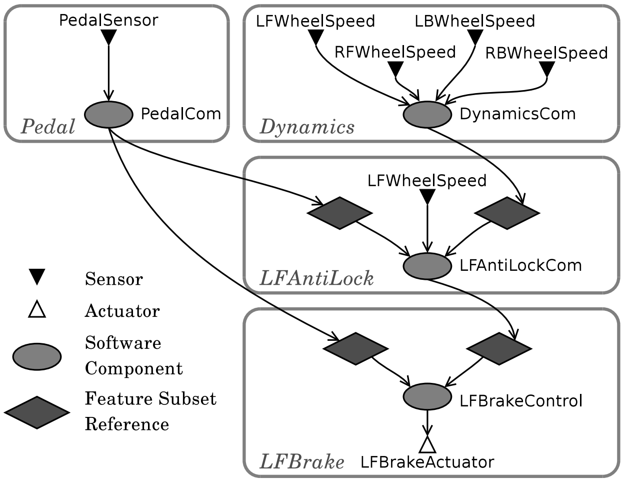

Figure 4. Example feature subsets from a braking system (based on Shelton [19])

Feature Subset	Configuration	Utility
LFAntiLock	LFAntiLockCom, LFWheelSpeed, Pedal, Dynamics	$0.7 + 0.3 * U(Dynamics)$
	LFAntiLockCom, LFWheelSpeed, Pedal	0.7
	Others	0

Table 1. Example utility values for the *LFAntiLock* feature subset

and that of any feature subsets it references. Shelton gives tables of example utility values for the feature subsets. The values for the *LFAntiLock* feature subset are shown in table 1. The utility value in each row of the table corresponds to one or more failure configurations of the components in the feature subset. All components listed in the configuration column must have a working status in order to use the corresponding utility value. The utility value for a configuration is written as a formula which can refer to the utility value of another feature subset using the function $U(\cdot)$. This is shown in the first row of table 1 which uses the utility value of the *Dynamics* feature subset.

In order to involve utility as part of our objective function for performing task allocation, it is necessary to create both an internal representation for efficiently calculating system utility and also an external representation for providing input data which describes the utility model for a system.

The internal representation for the utility of a particular feature subset is a decision tree. Each node of the tree is one of the components of the feature subset. At the leaves of the tree are the utility values. In order to calculate the utility value for the feature subset, the tree is descended from the root selecting either the *working* or *failed* branch from each node depending on the failure status of the node. For low level components, the failure status is decided by whether the component has failed or not. For higher level feature subset components, the failure status is *failed* if the utility value of the referenced feature subset is 0 else the failure status is *working*.

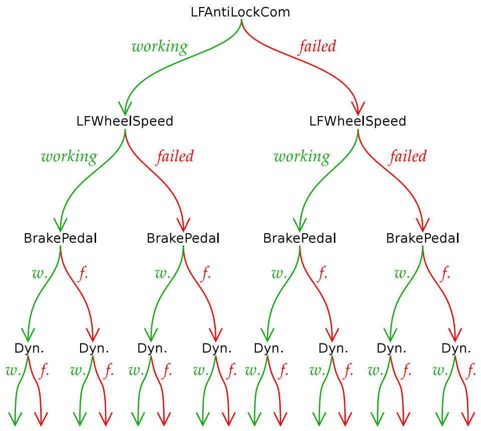

Figure 5. Fully expanded utility decision tree

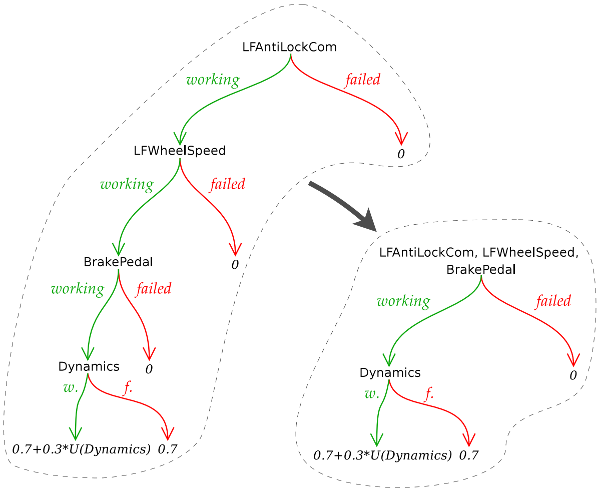

Figure 6. Pruned and compacted trees

In general, for a feature subset with k components, a utility tree will have a depth of $k + 1$ and have 2^k utility values on the leaves of the tree. An example of such a tree for the *LFAntiLock* feature subset is shown in figure 5. However, it is clear from table 1 that many configurations have the same utility value. In particular, for configurations dependent on multiple components, it is not necessary to check the status of every component if any have failed. This allows the tree to be pruned as shown in figure 6. The tree can be compacted further by allowing a decision node to depend upon multiple components as shown by the transformation in figure 6. This form has the advantage of being a more direct mapping from Shelton's table of utility values in table 1. It also forms the basis of an XML format for utility

Task	WCET	Period	Max Replicas
A	7	10	2
B	7	10	2
C	2	10	2
D	2	10	2

Table 2. Example Problem

decision trees which is not overly verbose. All of the lowest level components are linked to a task in the system specification. Each decision node lists feature subsets on which to base the decision between following the *working* or *failed* branch. Every node also has a *require* attribute which can be set to *any* or *all* which indicates how many of the features must be working in order to take the *working* branch. Every utility model input must include a feature subset named *System*. This is assumed to be the feature subset at the top of feature subset hierarchy and its utility value is used to calculate the utility value for the system under a particular failure configuration.

The derivation of utility values for each feature subset is not dealt with by this work. One way in which to interpret these values is to map a loss in utility to the expected monetary value [1] which will be the cost incurred by any failures caused by the faults. When combined with probabilities of the faults occurring expected monetary value is one method of quantifying risk.

Two cost function components which both use the utility metric are described below. The first is based upon keeping the system utility above an acceptable threshold for as many faults as possible. The other is motivated by the concept of expected monetary value and uses probabilities of processor faults to minimise the expected loss of utility.

5.4 Extensions To Cost Function

The system utility model can generate several values for all numbers and combinations of faults. The challenge of using this model in a search algorithm is to create a function which can map all of these values to a quality metric for fault-tolerance. Two approaches are introduced here. The choice of which to use will depend on the fault-tolerance requirements for the system.

The first is based on the utility values of the worst case combinations of processor faults over increasing numbers of faults. This is clarified with the following example. Table 2 shows a small system with four tasks and no messages. Deadlines are set equal to the period of the task. The timing requirements of tasks A and B dictate that they cannot be allocated to the same processor. Each task can be replicated up to two times so there may be up to three versions of each task in the solution. The utility model for this system, given in table 3, uses a simple additive model such that the utility provided by any particular task is independent of whether other tasks have failed. In this instance, the amount of utility provided to the system decreases from task A to task D.

Feature Subset	Utility
System	0.5 * U(A) + 0.3 * U(B) + 0.15 * U(C) + 0.05 * U(D)

Table 3. Utility model for example problem

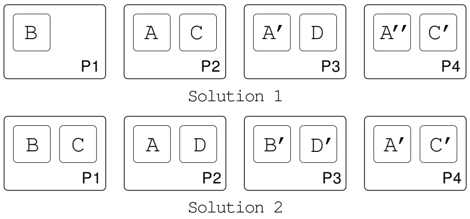

Figure 7. Two solutions to example problem

Two feasible solutions, where all schedulability constraints are met, are shown in figure 7. For Solution 1, the worst case single processor fault which can occur is when P1 fails since there are no versions of task B left in the system. The worst combination of two faults is when processors P1 and P3 fail which removes all versions of both tasks B and D from the system. The worst case system utility values for increasing numbers of faults for both Solution 1 and Solution 2 are plotted in figure 8. The answer to the question of which degradation profile is preferable will depend on requirements. In order for the cost function component to be able to select which solution is best, a threshold parameter, which is specified as part of the utility model, is introduced. It is assumed that once utility falls below this threshold the system in some sense becomes unreliable or unsafe. Therefore, the aim is to withstand as many faults as possible before the utility falls below this level. This is akin to trying to maximise the value to t when designing a t-fault-tolerant system. Using this model, Solution 1 is preferable for the low threshold value marked in figure 8 whereas Solution 2 is better for the high threshold. A cost

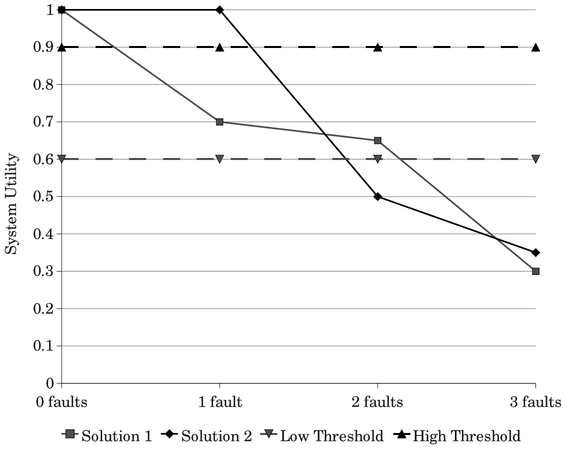

Figure 8. Worst case utility degradation

function component which will order solutions according to this scheme is given in equation (18).

$$M_t = \min_{F \in combs(\mathcal{P},t)} U(System|Ffailed) \tag{16}$$

$$l_t = \begin{cases} M_t \text{ if } M_t \geq L \\ -L \text{ otherwise} \end{cases} \tag{17}$$

$$g_{maxloss} = 1 - \frac{\sum_{t=0}^{|\mathcal{P}|-1} 2^t (l_t + L)}{(1+L)(2^{|\mathcal{P}|} - 1)} \tag{18}$$

In these equations, t is a number of processor faults. The set $combs(X, i)$ is the set of all combinations of i items chosen from X. l_t is the worst case utility value for t faults and L is the threshold utility value. In order not to differentiate between solutions once utility has fallen below the threshold, if $l_t < L$ then l_t is set to $-L$. The 2^t term ensures that systems which can withstand more faults will always have a lower value of $g_{maxloss}$. Note that the initial value of t is 0. The utility is not necessarily maximal for 0 faults since the search algorithm has the design choice of not including any versions of a particular task in the system.

Given an arbitrary utility model, calculating l_t requires every combination of possible processor faults to be evaluated. For systems with more processors, the number of combinations will grow rapidly. One way of limiting this is to only iterate up to a maximum number of faults. A suggested topic for future work is to investigate approximations for l_t for particular restricted utility models.

An alternative cost function component for assessing the fault-tolerance quality of a system is given in equation (20). Let $P(F)$ be the probability that processors in the set F have permanent faults.

$$h_{exploss} = 1 - \sum_{i=0}^{|\mathcal{P}|-1} \sum_{F \in combs(\mathcal{P},i)} P(F)U(System|Ffailed) \tag{19}$$

$$g_{exploss} = {h_{exploss}}^{0.075} \tag{20}$$

This equation calculates the expected system utility by considering the system utility for each possible fault combination across different numbers of faults. $P(F)$ is calculated by combining the probabilities of individual processor faults which must be specified. Processor faults are assumed to be independent though any probability function which maps a set of faults, F, to the probability of them occurring could be used. Since the probability of a processor fault is small, the probability of 0 faults dominates the total probability of 1 or more faults. The 0 fault case will usually coincide with full system utility. Therefore, although $h_{exploss}$ is capable of giving values in the full $[0, 1]$ range, the result will often be very low and the very small variations are insufficient to guide the search. Therefore, raising to a low valued power in the actual cost function component increases the size of the variations in the values which are typically produced without changing the range.

System	Replicas	Tasks	Messages	Procs	Time
evalsys	0	8	12	2	<1s
evalsys	1	16	48	4	2s
evalsys	2	24	108	6	9s
brakes	0	19	29	5	1s
brakes	1	29	90	5	16s
brakes	2	39	183	7	2m 10s
brakes	2	39	183	8	1m 24s

Table 4. Problem sizes with solution times

The final issue in the design of the extended cost function is the balancing of schedulability and fault-tolerance. Since schedulability is being treated as a constraint rather than an objective, the optimisation problem is a single objective problem which can be stated as: *optimise fault-tolerance such that all schedulability constraints are met.* To help achieve this balance a hierarchical weightings structure is introduced. Rather than attempt to balance fault-tolerance against each of the previous cost function components individually, components $g_1, \ldots, g_9$ which were designed for solving schedulability constraints are grouped together and the fault-tolerance cost function components is placed in a separate group. In addition to changing the importance of components within groups, a weighting can be applied to the groups themselves in order to balance the schedulability constraints and fault-tolerance objectives.

6 Evaluation

The evaluation concentrates on the effectiveness of the two cost function components for fault-tolerance given in equations (18) and (20). Two systems are used during the course of the evaluation. The first is a randomly generated 8 task example called *evalsys* and the other is a brake-by-wire example taken from Shelton [19] labelled as *brakes*.

The first experiments were performed to gain an understanding of how adding replicas increases the complexity of the problems. All timings were measured using a machine with an AMD Athlon 64 3500+ CPU clocked at 2.2GHz. The increase in problem size as replicas are added is shown in table 4. Of particular note is the number of extra dependencies which are generated. In the *brakes* problem, sensors, software components and actuators are represented but only software components are replicated. The problems were solved with all replicas enabled as straightforward task allocation problems with no fault-tolerance objectives. Additional processors were introduced to the system requirements to accommodate replicas. The *brakes* example has separation constraints on some tasks so a minimum of 5 processors were required. The time taken to solve each problem variation is also shown in table 4. Adding replicas significantly increases the time required to solve problems

Feature Subset	Configuration	Utility
System	τ_1, τ_2, τ_3	$0.25 * U(\tau_4) + 0.25 * U(\tau_6) + 0.25 * U(\tau_7) + 0.25 * U(\tau_8)$
	τ_1, τ_5	$0.25 * U(\tau_7) + 0.25 * U(\tau_8)$
	Others	0

Table 5. Utility values for *evalsys*

Num Replicas	Procs	L=0.8	L=0.4
1	3	1 (1.00)	1 (1.00)
2	3	unschedulable	unschedulable
2	4	1 (1.00)	2 (0.75)
2	5	2 (1.00)	2 (1.00)

Table 6. Fixed number of replicas

but all problems could still be solved within a few minutes. The final experiment shows that increasing the number of processors can make the scheduling problem easier.

In order to conduct experiments using the fault-tolerance components, utility models were needed for both systems. The model for the *brakes* system is taken from Shelton [19]. The utility model for the *evalsys* system was constructed as shown in table 5. It shows that task τ_1 is critical to the system and that the system can only run at full utility if tasks τ_2 and τ_3 are also present.

The next experiment used the *evalsys* problem to evaluate the worst case loss fault-tolerance component. It compared two strategies of replication. The first used a fixed number of replicas whilst the second allowed the algorithm to vary the number of replicas used. Separation of replicas is not enforced but solutions where replicas are allocated to the same processor should be heavily penalised for poor fault-tolerance quality.

Table 6 shows results for systems with a fixed number of replicas. The number of processors and threshold values were varied across different runs. The values in the final two columns give the number of faults which could be tolerated before the system utility fell below the threshold. The values in parentheses are the worst case system utility after that many failures. The original problem requires two processors to schedule all tasks. Therefore, a solution which duplicates this would require 4 processors and handle a single fault. However, the results show that it is possible to find a solution which achieves the same degree of fault-tolerance with only 3 processors. These runs took about 25 minutes to complete. Although there is some overhead from having to calculate worst case system utility values, this was insignificant compared to the additional time spent by the search in finding a schedulable solution since it had to balance this constraint with the objective of improving fault-tolerance.

Table 7 shows results when the search was able to vary the number of replicas. These runs took longer still, taking up to an hour to find good solutions. Finding a schedulable solution is now easier for the search because it is able to remove replicas. To compensate for this and maintain a good level of utility, it was necessary to adjust the balance of weights away from the group of schedulability cost components and in favour of the worst case loss component. For fixed numbers of replicas, schedulability was weighted 10 times higher than fault tolerance but for this latter table of results, fault tolerance was weighted more highly in a ratio of 2 to 1. This difficulty is emphasised by the fact that the result achieved for a threshold of 0.4 and 5 processors is slightly worse than that of the equivalent result with a fixed number of replicas. However, the benefit can be seen in that it was able to withstand an additional fault with only 3 processors when the threshold was set at 0.4. Table 8 shows the number of replicas included in each solution for the results in table 7. This shows that the algorithm correctly favoured the critical task, τ_1, and increased the number of replicas used when extra processors were available.

Max Replicas	Procs	L=0.8	L=0.4
2	3	1(1.00)	2 (0.50)
2	4	1(1.00)	2 (0.75)
2	5	2(1.00)	2 (0.75)

Table 7. Variable number of replicas

Task	0.8,3	0.8,4	0.8,5	0.4,3	0.4,4	0.4,5
τ_1	1	2	2	2	2	2
τ_2	1	1	2	1	2	2
τ_3	1	1	2	1	2	2
τ_4	1	1	2	1	1	2
τ_5	0	1	0	1	2	1
τ_6	2	1	2	1	1	1
τ_7	2	1	2	1	1	1
τ_8	1	1	2	2	1	1
Total	9	9	14	10	12	12

Table 8. Number of replicas used

The expected utility loss component was tested with the *brakes* system. Since this system did not have redundant sensors included in the example, it will not withstand any faults in the worst case but the expected utility loss can still be improved. The probability of each processor failing in a given time frame was set to 0.001 and the maximum number

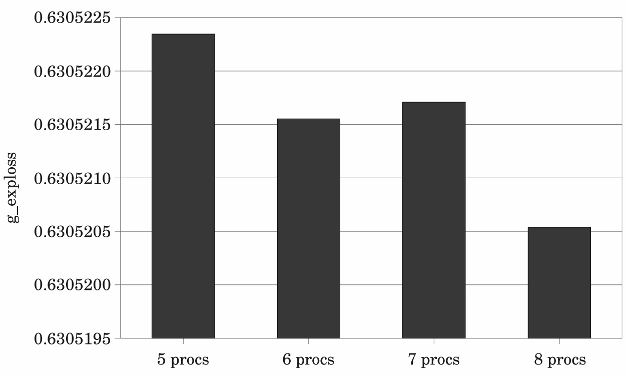

Figure 9. Expected loss

of replicas was 2. The results are shown in figure 9. The solutions showed a general pattern of decreasing expected loss of utility as the number of processors increased though, on this set of runs, the solution for 6 processors was slightly better than that for 7.

The examples presented here are limited in size though grow rapidly as replicas are added. The largest performance problem has been found to be the growth of the solution space caused by having variable numbers of replicas. However it is hard to extrapolate performance based on size alone. For problems which are large but are easily schedulable there will be fewer issues balancing schedulability constraints with fault-tolerance objectives.

7 Conclusions And Future Work

This paper has presented extensions to a task allocation search algorithm to make architectural decisions influenced by fault-tolerance requirements. Suitable objective functions were presented which allowed for gracefully degrading systems to be generated. The pattern of degradation can be changed by setting a threshold parameter for the worst case utility loss component. An objective function for expected utility loss was also given. The neighbourhood for local search was extended to allow the number of task replicas to be varied.

The are some remaining issues for future research. The probability of a processor failure is rarely independent of other failures. The same optimisation methods and metrics could be used in conjunction with a more complex probability model. To this point, communication failures have not been accounted for. The methods discussed in this paper could be used to complement previous work which constructs schedules for resending messages and/or rerunning tasks when failures occur.

References

[1] J. Abrams and R. Cone. Implementing expected monetary value analysis into risk metrics and assessment criteria. In *Proceedings of the 24th International System Safety Conference (ISSC)*, August 2006.

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

[3] F. Bicking, B. Conrard, and J. M. Thiriet. Integration of dependability in a task allocation problem. *Instrumentation and Measurement, IEEE Transactions on*, 53(6):1455–1463, 2004.

[4] K. Echtle and I. Eusgeld. A genetic algorithm for fault-tolerant system design. In *Dependable Computing*, Lecture Notes In Computer Science, pages 197–213. Springer, 2003.

[5] P. Emberson and I. Bate. Minimising task migration and priority changes in mode transitions. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 158–167, 2007.

[6] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 125–125, April 2001.

[7] W. Jia and W. Zhou. Reliability and replication techniques. In *Distributed Network Systems*, Network Theory and Applications, pages 213–254. Springer US, 2005.

[8] J. P. Kany and S. H. Madsen. Design optimisation of fault-tolerant event-triggered embedded systems. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2007.

[9] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[10] W. Nace and P. Koopman. A product family approach to graceful degradation. In *Proceedings of the IFIP WG10.3/WG10.4/WG10.5 International Workshop on Distributed and Parallel Embedded Systems*, 2000.

[11] Y. Oh and S. H. Son. Scheduling real-time tasks for dependability. *The Journal of the Operational Research Society*, 48(6):629–639, 1997.

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

[13] 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*, pages 53–62, 2007.

[14] C. C. Price. Task allocation in distributed systems: A survey of practical strategies. In *ACM 82: Proceedings of the ACM '82 conference*, pages 176–181, 1982.

[15] X. Qin and H. Jiang. A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing*, 32(5-6):331–356, June 2006.

[16] O. Redell. Analysis of tree-shaped transactions in distributed real time systems. In *16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 239–248, 2004.

[17] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.

[18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[19] C. Shelton. *Scalable Graceful Degradation For Distributed Embedded Systems*. PhD thesis, Carnegie Mellon University, June 2003.

[20] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: An NP-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[21] Y. C. Yeh. Safety critical avionics for the 777 primary flight controls system. In *Digital Avionics Systems, 2001. DASC. The 20th Conference*, volume 1, 2001.

[22] 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*, 184(2):407–420, January 2007.