Efficient Task Allocation to FPGAs in the Safety Critical Domain

Philippa Conmy
Department of Computer Science
University of York
Heslington, YO10 5GH
philippa.conmy@cs.york.ac.uk

Iain Bate
Department of Computer Science
University of York
Heslington, YO10 5GH
iain.bate@cs.york.ac.uk

Abstract—**Field Programmable Gate Arrays (FPGAs) are highly configurable programmable logic devices. They offer many benefits over traditional micro-processors such as the ability to efficiently run tasks in parallel and also highly predictable timing performance. They are becoming increasingly popular for use in the safety critical domain where predictability is essential. However, concerns about their dependability, principally their reliability and difficulties in assessing the impact of an internal failure means that current designs are inefficient and conservative. This paper discusses these issues in depth. It also presents an FPGA task allocation method using simulated annealing to balance efficiency and reliability requirements. This can be used to improve designs of safety critical FPGA based systems.**

Keywords- safety, dependability, reliability, FPGAs, simulated annealing.

I. INTRODUCTION

FPGAs consist of millions of simple, programmable logic cells. When these are configured and connected they can perform many different complex tasks in parallel to one another. As the underlying technology is simple they have very predictable timing and power performance. This is a strong advantage in the high-integrity and safety critical domains where guarantees must be made to ensure a system meets its requirements. However, it can be difficult to predict the effect of a fault within the network of logic cells, and it can also be difficult to gather accurate data about the probability of faults. Hence, Safety Critical Systems (SCSs) using FPGAs tend to be very conservative, with physical replication of boards and/or external monitors of their outputs. If we could better exploit their inherent parallelism to run more tasks on fewer boards then there are potential savings in terms of weight, power requirements and overall cost. Some small SCSs, such as Unmanned Aerial Vehicles (UAVs) or missiles would benefit considerably from these savings.

This paper contributes a discussion of the many issues which can be involved in assuring that an FPGA is acceptably safe. We also contribute a Simulated Annealing (SA) method which can be used to allocate tasks on FPGAs. The allocation process determines how to spread out various tasks, and their duplicates, in order to balance resource usage and reliability requirements for safety critical systems. These two properties can be in contention with one another – for example a larger number of FPGAs reduces the likelihood of total failure but is wasteful in terms of power and weight requirements. Our method attempts to balance these. We also present a detailed discussion of the issues involved in assessing fitness of a solution and also provide suggestions for improvements.

This paper is laid out as follows. Section II provides background information on FPGAs and safety and explains our motivation for the work in detail. Section III describes the SA development process and parameters. Section IV describes our results and section V presents conclusions and areas for further work.

II. BACKGROUND INFORMATION

This section describes three different sets of background information. First we provide basic background information to FPGAs and how they work. Second we describe design and development principles for SCSs as these shape and constrain the task allocation we are permitted to attempt. Third we discuss principles of dependability, as these help shape the goals of the task allocation process. Throughout the section we discuss various fault mitigation strategies which are used in the domain.

A. Introduction to FPGAs

As noted in the introduction, FPGAs have millions of programmable logic cells which can be configured and programmed to perform a variety of tasks. Typically an engineer will describe the required FPGA logic using a Hardware Description Language (HDL). This is then put through a two stage synthesis process, first to describe the hardware components which are needed to implement the HDL, and a second Place and Route (P&R) stage which describes where on the FPGA the hardware components will be located. P&R can be manually adjusted by the developer, e.g. using tools such as Xilinx's PlanAhead [1]. Using a modular approach to HDL development, and some manual intervention, individual tasks can be grouped onto defined sets of logic cells. As previously noted, the tasks can then run in parallel without any detriment to each other's performance.

The authors would like to thank the Software Systems Engineering Initiative (SSEI) for their support and funding.

We use these basic principles as motivation for our idea that multiple safety critical tasks can run together on the same FPGA.

FPGAs can be susceptible to various types of hardware failure (as classified in [2]). These include bit flips (which may be transient or permanent, Single Event Upsets (SEUs) or Multiple Bit Upsets (MBUs) [3]), degradation of components over time, or failures on interconnects. The effects of these can be extremely hard to predict. For example, a bit flip within a Boolean look-up-table may only be triggered within certain sets of inputs, giving the illusion of a transient failure. This makes it extremely difficult to track down the source as well as predict the effect.

A failure on an interconnect may disconnect part of a circuit, and create and unwanted connection between others. This can cause power problems as well as stale/incorrect values being propagated through the device. Again, faults may be propagated intermittently and so are hard to track down.

There are various methods which can be used to protect or mitigate against these and their effects including the following:

- Use of antifuse devices – these fuse the interconnects and so are not susceptible to SEUs, however they are not reconfigurable.
- Reflashing/configuring reprogrammable FPGAs – this has the advantage that temporarily stuck bits can be fixed but may cause the device to wear out more quickly
- Physical redundancy – use of multiple FPGAs to protect against hardware failures
- Logical redundancy – use of multiple copies of functions on the same FPGA, as these will run in parallel there will be no performance problems using this approach.
- Place and route strategies – ensuring that different parts of the design have minimum separation between them to prevent single failures causing data/power interference
- Dissimilar hardware – many FPGAs have built in micro-processors so the same function could be run on both the logic cells and the micro-processor, this is discussed further in the next section.
- Appropriate use of HDL – certain types of hardware problems can be avoided by careful use of HDL, e.g. to avoid timing problems caused by latches [4].

B. Design of safety critical systems

SCSs are those whose malfunction could cause death, injury, damage to property or to the environment [5][6][7]. They are generally subject to stringent development guidance which is designed to eliminate, mitigate or otherwise manage the risk inherent in the system. Risk is typically assessed as the combination of the likelihood of an event as well as its severity. The more severe the outcome of a malfunction is, then the more effort that must be put in to reduce its likelihood. It is generally impossible to remove all aspects of risk from a SCS whilst keeping it fit for purpose, therefore qualitative and quantitative measures are used to assess whether an SCS is acceptably safe.

Architectural design is one area which must be considered as a potential source of failure. A SCS must be designed to be robust against:

- Single points of failure – where a single fault in one component could cause a hazardous situation
- Common cause failures – where a fault common to more than one system component could cause a hazardous failure

There are a number of standard architectural and functional design patterns which have been developed to mitigate against these. For example, N-Modular Redundancy (NMR) in which N copies of a software function are used, along with a voting mechanism to compare the outputs. If there is an error in a minority of the outputs then the voter should detect and reject it. Intelligent voting systems can also detect validity of data, either in terms of value ranges or data trends. As noted in the previous section, NMR can be implemented on the same FPGA without loss of performance.

Replicas of components can provide protection against single points of hardware failure, but may still be prone to common cause failures or a fault in the voter. Using diverse components in the design, for example using different types of processors to run replicas or even writing two different versions of the same function, can mitigate common cause failures. However, these will have different performance characteristics or could produce different values (e.g. due to rounding errors). Discrepancies may also occur when components are simply replicated, e.g. due to physical flaws in only one component or clock drift. A robust and well designed voting system can deal with discrepancies, but may be more complex to design and analyse. For example, in [8] the authors describe a robust voting mechanism which can deal with both time and value discrepancies, within certain constraints.

System developers must weigh up the different design options to determine whether the risk (the likelihood and severity) associated with a fault justifies the cost associated with developing and deploying a more complex, and/or replicated design which uses various types of mitigation.

C. Dependability

We have noted that the physical design of an FPGA supports parallel processing. In theory this means that multiple applications, or even multiple copies of the same application could be co-located on the same FPGA with no loss in performance, no interference and no resource contention. Such a design would not be acceptable for a safety critical system as

the loss of the FPGA would mean all allocated applications were lost simultaneously (and this is assumed to be an unsafe state). However, when spread over two FPGAs the applications may be able to meet probability targets for continuous reliability such as those found in IEC 61508 [7], reliability being one important aspect of overall integrity and dependability. Note that we are using reliability and availability as defined by Avizienis et al. in [9]:

- **availability**: readiness for correct service;
- **reliability**: continuity of correct service;

FPGA based designs are often overly conservative, with minimal co-location of functions and multiple replication of boards. In addition, there can be an overestimate of the impact of an SEU within an FPGA, assuming that it would cause the worst possible output in each case, when in fact the likelihood of an SEU is generally very low, and the likelihood of it affecting an area which actually alters the output of the FPGA is even lower (see [2]). The aim of this work is to look for a way of minimising physical resource requirements, whilst ensuring reliability targets can be met. This has been achieved by using a SA algorithm [10], with a fitness function tailored towards assessing resource usage and reliability. The difficulty in developing an appropriate (and generic) fitness function is one of the key findings of this work.

III. HEURISTICS

This section of the paper describes the development of the SA approach.

A. Simulated Annealling

We have developed a SA approach which allocates tasks with an NMR architecture. The basic application structure is shown in Fig. 1, with two replicated tasks and an active and inactive standby voter. The tasks may be replicated and moved to alternate FPGAs. Tasks represent individual functions which are assumed to be independent. In other words they do not need to communicate with one another. An example task might monitor the signal from a sensor, e.g. for gas levels in a mine or as part of a fire protection system. Alternatively, they might process some data, e.g. to remove noise from an audio signal for communications. It has been assumed that the loss of more than one of these tasks in combination cannot cause a more severe system state than the one they have been classified with. For example, suppose that two tasks have been analysed and determined to have a "major" effect on safety if lost. In some systems the combination of loss of those two tasks could have a "catastrophic" effect on safety. We have assumed that this will not be the case for our experiments – further work will investigate this further.

Tasks are allocated and/or replicated to multiple FPGAs within a system. A SA approach to allocation was chosen as it is a well-recognised [11][12] technique for creating designs using search-based techniques. In addition it has been effective for other task allocation problems [12]. SA tends to provide efficient and effective search even when the landscape is complex, e.g. piece-wise discontinuous, large, and problems that feature dependencies [13]. The problems considered in this paper are similar to those in [11][12][13]. Further study of search algorithms may uncover slightly more efficient and effective search algorithms, however here our main interest is understanding the design problem. An alternative approach would be to use an evolutionary or genetic algorithm. However, these require many different random solutions to be generated at each stage, and compared and contrasted. It wasn't clear at the start of the work exactly what acceptable solutions would look like, hence it was decided to use SA to explore and search this problem, examining it stage by stage.

SA is a meta-heuristic method for gradually converging on acceptable solution to a complex problem. Rather than attempt to find an optimum solution to the problem, the SA algorithm will iteratively explore the search space with a random action taken per iteration. Ideally at each step an improvement will be made on the previous result. The improvement and acceptability of the solution found at each step is measured using a fitness function. In order to avoid getting trapped in local minima the SA algorithm is frequently adapted to allow a worse solution to be explored for a few steps. If it becomes clear that the solution is decreasing in fitness, then the algorithm can retreat a number of steps to a previous better solution and attempt to randomly explore the space again from there. The algorithm can be run for a fixed number of steps, or until a certain fitness value is reached. The SA described here uses a fitness function which measures optimum use of resources and also reliability targets.

B. Background architecture for the approach

We now describe the application design shown in Fig. 1 in more detail. The two lanes are on different FPGAs, and a backup voting function is on hot standby. This system is robust against hardware failures internal to one of the functions or voters (such as an SEU), and it is also robust to the loss of a single FPGA. However, it is not robust to logical common cause failures, future work should consider diversity as noted previously. Note that it is assumed that data can be sent to the voter within a required time frame from both functions, and that the voter can handle any discrepancies that may arise in value or time (e.g. using the model from [8]). It is further noted that this architecture can be inefficient, as discussed by the authors of [14], although that paper refers to older FPGA devices which had many fewer gates. In any case, future development of the SA technique is needed to look at different architectures.

One problem with this design is that it is wasteful in terms of resources with many FPGA logic cells unused. Fig. 2 shows another FPGA system, this time with two independent applications sharing two FPGA boards. Again both systems are robust to a single point of failure, but this time the physical resources are shared. There should be no contention for FPGA resources, and as long as a minimum separation is maintained

between them (e.g. of two or more interconnects), no single failure in an interconnect should cause interference. If these two systems were on four instead of two FPGAs then the design would be more expensive, weigh more and have higher power requirements.

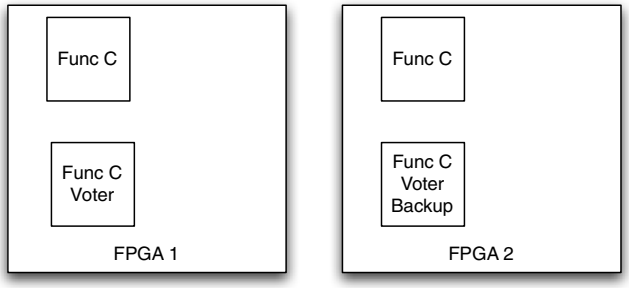

Figure 1. Two Lane Voting System

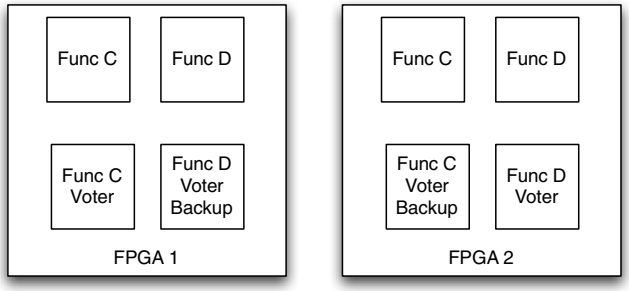

Figure 2. Sharing FPGAs for multiple two lane voting systems

C. System representation and implementation

A C program was developed which models FPGA tasks and their allocations. FPGAs are represented as having a fixed number of cells, upon which zero or more tasks are allocated. Applications are represented as a series of replicated tasks, with two voters – one active and one on hot standby. Voters will always be allocated to different FPGAs to avoid a single point of failure. Tasks and voters are represented by a simple datatype describing the number of logic cells they require on an FPGA and their name. Interconnects are assumed to be part of the cells. They are also assigned a criticality (ranging from catastrophic down to no safety effect). The criticality represents the severity of consequences of their loss, and probability/reliability targets for each application are based on this.

These models are gradually developed and expanded using the SA algorithm, with tasks being replicated, moved and additional empty FPGAs added. A "best" system description is output to a file, with information describing decisions made at every step. The user can alter a random number seed in order to compare and contrast different sets of results. The full method and fitness function are described in the next section.

The main purpose of this tool is to assist system developers in comparing different allocations and to determine how much physical resource is required. At present it is limited to allocating independent applications with a specific design pattern to a series of FPGAs. However, the application model can be extended at a later stage to include inter-dependencies using techniques such as those found in [11], and also to accommodate different design patterns. The experiments using the NMR design patterns have allowed the SA to be developed, increasing our understanding of its sensitivities. This knowledge can also be used when extending application models.

An alternative use of the tool would be to actually embed it upon a dynamically reconfigurable FPGA. It could then be used to re-allocate functions and voters when one had failed. However, it would be difficult to predict the multiple different outcomes and so the tool and SA algorithm would need to be validated. If it is used as a design-time tool then the system designers can verify its output independently and the tool itself wouldn't require such stringent validation [6]. In other words, the proposed layouts from the tool would be examined manually to determine if they are deemed to be acceptable.

D. Development of SA Fitness Function and Algorithm

This section describes the development of the SA method. First we describe the two measured aspects for a system fitness function – reliability and resource usage. Then we describe the algorithm itself and how the fitness measures are used.

1) Resource and Reliability calculations

In order to develop the SA algorithm we need to understand what constitutes an ideal or optimum system allocation. For the types of application examined this would be a system with the minimum set of resources used whilst still maintaining adequate partitioning and meeting reliability targets. As this is a safety critical system which may contain high-integrity applications it is more important to meet reliability and partitioning targets than to completely optimise resource usage. However, that doesn't mean that improvements in resource management can't be made (e.g. to avoid largely empty FPGAs being deployed).

The resources in question are the logic cells of an FPGA. As discussed, each task will require a certain number of logic cells on the FPGA in order to perform its function. In addition, a minimum separation between co-located tasks will ensure that a single hardware failure between tasks cannot allow them to become connected. This minimum separation will alter depending on the target FPGA, for the purposes of this research we assumed a separation of 4 interconnects via cells was required but this parameter can easily be altered in the program. The overall resource usage is therefore simply calculated as the area required for all tasks plus minimum separation from other tasks. We have not at present included a measure for power dissipation, although this can also be a problem, either due to inefficient HDL or use of the FPGA [15]. However, this type of measure could be added to the code relatively simply as part of further work.

TABLE I. DESCRIPTION OF ACRONYMS USED FOR CALCULATIONS

Symbol/Acronym	Explanation
P(I)	Probability of a fault occurring anywhere on the FPGA board over a fixed period of time.
P(IT)	Probability of Internal Task failure. The likelihood that a fault on the FPGA will affect the logic cells of an individual task (e.g. a bit flip within a cell)
P(B)	Probability of failure of an entire FPGA board (e.g. total power loss) over a fixed period of time.
Res (S)	Resource usage of a single System, e.g. how many cells spread over a number of FPGAs that the tasks
Rel (S)	Reliability of a single System as spread over several cells and several different boards. Calculated using P(IT) and P(B) as described above.

Note that a description of the acronyms used in this section and the next is included in Table I. The likelihood of an FPGA hardware failure causing a task to fail can be calculated as the probability of an internal failure per task P(IT) added to the probability of the failure of the FPGA board P(B), as they are mutually exclusive[1].

P(IT) is calculated by taking the overall likelihood of a cell failure on the FPGA over given number of operating hours, P(I), and multiplying it by the percentage of cells used by that task (or by a voter). For example, if a task takes up a quarter of the FPGA then it has a quarter of the likelihood of the whole FPGA to be affected by a failure. It should be noted that this is a pessimistic and slightly simplistic approach, in fact not all internal failures may cause the task to fail, for example, an SEU within a LUT may or may not alter the output of a logic cell [2]. A more realistic model would identify and categorise different failures, e.g. inter-connect, flip-flop, LUT etc. and determine which areas are more critical, for example using the failure propagation analysis described in [16]. In addition, a more complex model would take into account the loss of combinations of the applications. As discussed, even if they are executing independently, it may be that there is a higher system level problem caused by the loss of, for example, two "minor" applications which then becomes a more serious safety concern. However, using a simpler working model allowed us to develop the algorithm, which can be extended to accommodate more complex probability calculations.

P(B) is a fixed figure per FPGA. The figures used in the experiments were partly based on those found in [17]. For the applications within our system there are two concerns. Total loss of functionality which is caused by loss of both voters or all lanes. In addition, we need to be concerned with erratic or plausibly incorrect output. This can be caused by problems such as bit flips in, or losses of, the majority of lanes or the voters. If two lanes are located on different FPGAs as shown in Fig. 2 then the probability of them both being lost due to board failure is P(B)/2 i.e. it is halved. The overall P(I) is P(IT)*P(IT) as they are independent. As even more FPGAs are added to the system, and more replicas created, the likelihood of failure is thus further reduced. However, there is a law of diminishing returns whereby smaller and smaller gains in terms of reliability are achieved beyond a certain number of replicas, but the resource requirements become larger. The fitness function attempts to balance this.

The probability calculation function in our program can be easily altered and updated to include different failure figures and different calculations (e.g. to include combinations or take into account certain classes of failures which could actually be detected by the voter).

We have assumed that a fault, even an MBU, is localized to a single interconnect or cell. The experiments described in [3] found that although up to five bit flips could be caused by a single particle strike these were localised to a single logic cell on FPGAs with smaller components. As components continue to shrink this may not continue to be the case and this would also need to be included in an updated calculation.

Our initial experiments used an NMR design pattern with only a single voter. However, the probability of loss of the voter was so high that the reliability targets could never be met for the highest-integrity applications. This was based on reliability targets for each criticality as given in IEC 61508 [7]. Hence our design pattern was altered to use an additional voter on hot standby. Experiments which reveal this type of information are useful to a safety critical systems developer, as they would indicate that their proposed design pattern was unsuitable.

2) SA Development

The previous section established a base model of reliability and resource usage. Using this the SA was developed, via trial and error, to take a simple system with two FPGAs and a series of single tasks (not replicated), some of which do not reach their reliability targets, and expand it further and further until a certain level of fitness to the optimum is achieved. On each iteration the SA randomly picks one of four options:

- Adding a new, empty FPGA to the system,
- Adding a copy of an existing task to the system (and also a set of two voters if these do not exist for that task),
- Moving a task to an alternative FPGA,
- Deleting an empty FPGA if one is found.

[1] Note that this a measure of only part of an overall system probability target, for example as part of a system fault tree analysis.

The final option was added after early experiments indicated a lot of empty FPGAs were left in output designs or single tasks could be spread over too many FPGAs. This is likely to be due to problems with random number distributions, further work can look at this issue with alternate number generators and distributions. Adding the delete action improved the fitness of the results considerably.

Early experiments also helped determine the weightings given to each option which are as follows:

- 2/19 add an FPGA,
- 6/19 add a task copy,
- 9/19 move a task,
- 2/19 delete an empty FPGA.

Initial results also had large numbers of tasks added. This led to limited gains in terms of reliability improvements, and (as stated) is more wasteful of resources. Therefore the weightings were biased towards moving tasks, which provided better results. A few restrictions were enforced: tasks could only be moved to a different FPGA, voters were not duplicated, voters could be moved but not such that both were on the same FPGA, tasks and voters could only be moved to an FPGA which had enough free cells to contain them and if no such FPGA could be found then that round of actions was skipped. The final restriction prevented the algorithm from staying in an infinite loop. Future tuning of weightings will also look at [12] [13] .

3) Fitness Calculations

Having performed an action on the system its fitness must be calculated. As discussed earlier, the optimum arrangement is to minimise the number of resources used whilst ensuring probability targets are met. The fitness was calculated based on these two attributes as shown below, where Res(S) represented the system fitness based on resource usage for a system (in terms of cells) and Rel(S) represented the fitness based on probability of total loss of functionality for each system (see previous section).

$$F = 0.4*Res(S) + 0.6*Rel(S) \quad (1)$$

F, Res(S) and Rel(S) are all normalised values between 0 and 1, with 0 representing an ideal fitness and 1 the worst possible. As shown, a slight bias has been made in the fitness calculation such that the reliability fitness has more weight than the resource usage. As safety critical tasks are involved in this system, ensuring that probability targets are met is of a higher priority than minimising the resource usage. When presenting a safety case for this system it would be necessary to show that the risk of failure has been reduced to an acceptable level. By taking this approach a claim that risk reduction was prioritised can be justified. Note that Resource usage is calculated per FPGA. Reliability is calculated per application.

Both Res(S) and Rel(S) are calculated by first calculating a maximum overall penalty for either an application or to an FPGA. The actual penalty is then calculated based on whether a number of different criteria are met. The penalties and their criteria are shown in Table VI and Table VII (at the end of the paper due to their size). For example, in a system with two FPGAs where one has 50% of its capacity used and one has 10% of its capacity used the maximum penalty would be 200. The actual penalties would be 25 and 100. The Res(S) would be 125/200 or 0.625.

A similar, but more complex, system is used to calculate the reliability fitness. In this case the current achieved reliability of an application is compared with the target reliability, dependent on its criticality. Applications are penalised dependent on how close they are to the reliability. Higher criticality applications have harsher criteria, for example an application which was 5 times above its reliability target would be penalised by the maximum of 100 if it had a catastrophic consequence of loss but only by 40 if it had a minor consequence. Experimenting with different starting systems helped develop the criteria. It was found that they worked best when closely coupled to first few reliability calculations, i.e. providing the maximum penalties around the starting values was most useful. In addition, these values were very sensitive to single jumps in the search space, adding a single task copy to one FPGA could alter the probability significantly for, in particular, lower integrity or large tasks.

As the penalties may be affected by both task size and criticality it is difficult to tweak the criteria into a standard set for every different combination of applications that may be used. Further work will look at automatic generation of penalty parameters, based on the starting set of tasks.

After each iteration, the current system was compared to a "best" system. If the current systems score was better than the best then the current became the new best system. However, if the current system was worse for more than three iterations then the program replaced the current system with the previous best. Allowing the system to take a few steps before throwing away a solution was necessary to help prevent the program from getting trapped in local minima. Systems with empty FPGAs are penalised heavily and have a poor fitness, but it may be necessary to introduce an empty FPGA to create more space for adding new replicas, ultimately producing a better overall system. It will take a few steps to do this (e.g. adding an FPGA, moving or introducing new tasks to eventually get a better score) hence the need for exploration and retreat if necessary.

IV. RESULTS

This section discusses the results of running the SA algorithm for two different sets of tasks, each set forming a starting system spread over three FPGAs. The first set is shown in Table II, the second set in Table III. The second set has the same five tasks as the first along with two additional tasks. These data sets are intended to demonstrate that the algorithm

is flexible enough to deal with different numbers of tasks, and with different task sizes. They are not based on any existing system data, but were purely experimental.

Summaries of the results for 500 runs of the algorithm for both systems are shown in Table IV and Table V, showing minimum, maximum and average statistics for the final systems. The final number of FPGAs, the fitness scores and the number of replicated tasks is shown.

Scatter graphs in Fig. 3 and Fig. 4 represent the number of resulting systems (vertical axis) which had a given number of replicated tasks (horizontal axis). This helps visualise the spread of results over the 500 runs. The vertical scale is logarithmic to ensure outlying results are incorporated. The far right hand set is the sum of all sets where over 20 tasks were replicated as these were sparsely populated.

The results for each are now discussed in turn, followed by a comparative summary.

1) Results for System 1

As is shown in Table IV, in each case at least one system was produced with only one replica for each task. In addition, a few outlying results had very large numbers of replicas. However, the average number is much lower, and is reasonably proportional to the criticality of the task, for example, Task 2 has only minor criticality if it ceases to function and the average is 1, whereas Task 4 which is labeled as having a catastrophic consequence of failure is higher at nearly 8 replicas average. However, Task 1 and Task 3 with lower criticalities have higher averages than this. If the size of the tasks is taken into consideration this makes more sense, as more replicas may be needed to mitigate against internal failures (such as a particle strike) which are more likely to hit a larger task. Nevertheless it's possible improvements could be made here.

TABLE II. STARTING SYSTEM 1

Task	Size	Criticality
1	256*256	Major
2	100*266	Minor
3	200*365	Hazardous
4	150*275	Catastrophic
5	150*50	Hazardous

TABLE III. STARTING SYSTEM 2

Task	Size	Criticality
1	256*256	Major
2	100*266	Minor
3	200*365	Hazardous
4	150*275	Catastrophic
5	150*50	Hazardous
6	125*150	Minor
7	135*70	Catastrophic

TABLE IV. SUMMARY OF 500 RESULTS FOR STARTING SYSTEM 1

	Minimum Number	Maximum Number	Average
Number of FPGAs in system	3	8	3.36
System Fitness Score	0.36	0.56	0.390
Task 1 replications	1	46	8.324
Task 2 replications	1	8	1.014
Task 3 replications	1	42	12.48
Task 4 replications	1	47	7.804
Task 5 replications	1	54	5.31

As can be seen from the scatter graph in Fig. 3 there is a fairly even spread in terms of numbers of replicas, rather than a spike indicating a common consensus. This does mean that solutions with varying numbers of task replicas are produced. However, these are still only located on a small number of FPGAs (366 systems had 3 FPGAs, 103 had 4 and only 31 (6.2%) had over that), and in fact there is no detrimental performance effect by having a few more replicas than is strictly necessary to meet reliability targets. Harsher penalties on systems which overreach their targets could improve these figures. 311 of the solutions produced systems for which every task met its reliability requirements.

2) Results for System 2

The results for the second system (Table V) show similar characteristics. In each case a system with a single replica for each task was produced, and again there were occasional high numbers of replicas produced. The average number of replicas is again reasonably in proportion with respect to both the size and criticality of the tasks.

TABLE V. SUMMARY OF 500 RUNS FOR SYSTEM 2

	Minimum Number	Maximum Number	Average
Number of FPGAs in system	3	7	3.346
System Fitness Score	0.343	0.629	0.394
Task 1 replications	1	42	8.262
Task 2 replications	1	2	1.004
Task 3 replications	1	40	9.57
Task 4 replications	1	48	6.6
Task 5 replications	1	28	3.818
Task 6 replications	1	13	1.044
Task 7 replications	1	48	7.474

In terms of the spread of results in terms of numbers of replicas (Fig. 4), the results for this system show a more obvious downward trend towards fewer replicas. There is still some variation, however again this was limited to a small number of FPGAs with 372 system solutions having 3 FPGAs, 92 having 4 FPGAs and only 36 solutions (7.2%) with a

greater number. Only 257 of the solutions produced systems for which every task met its reliability requirements.

3) *Overall Results Discussion*

These systems are relatively small and the results obtained were unsurprising given the data sets. In other words, it is likely a designer may have come up with similar solutions albeit with much more manual effort. However, with larger starting sets of tasks the tool becomes increasingly useful as there are many more permutations to consider. Future systems, e.g. UAVs, are moving towards requiring larger numbers of systems but using fewer resources, e.g. to keep the weight and size down. This tool, once developed further, can help meet these aims.

The results demonstrate that using the proposed combination of weightings with the fitness function and SA can drive towards a solution which replicates the more critical functions and thus improve overall safety. However, there is still a relatively high possibility of finding an unacceptable solution if the program is only run once. With improvements to the penalties used improvements can be made to reduce this.

V. CONCLUSIONS

This paper has described the development of a SA algorithm used to optimise task allocation to FPGAs. The method has been relatively successful, producing systems which meet all the reliability requirements of their constituent tasks. However, in a few outlying cases very poor solutions were found and in other cases solutions which couldn't quite meet the criteria were found. In fact we found that developing the fitness function was a complex and finely balanced task. It was discovered that the fitness weightings were very closely coupled with the starting reliability calculations of the system tasks (and hence their size). We can use these findings to possibly automate construction of fitness parameters based on the starting system, thus improving our results. In addition, it is possible that poor solutions could be anticipated and discarded more quickly.

It was discovered that, with the assumed hardware failure rates, higher criticality systems would never meet their targets if only a single voter was used. In fact, further experiments showed that, for some sets of tasks, even dual voting high-integrity applications might never meet their reliability targets. The allocation approach could thus be improved by supporting even more design patterns, with more backup voter lanes if possible. This is perhaps an unsurprising result, as the voter can quickly become a bottleneck as a single point of failure for any NMR system.

Another improvement would be to have a better failure and probability model, which takes into account the likelihood of a fault affecting the output of a task rather than assuming a fault always causes an output failure. It is a relatively simple task to change the probability measures used in the model, but to assess the likelihood some fault analysis would be needed.

Further extensions to the system could take into account diverse implementations and diverse hardware for example, with microprocessors. In addition, dependencies between tasks, both in terms of communication and also combinations of loss could be included.

It is anticipated that this allocation tool (when developed further) could be used during initial design phases or alternatively on board as part of dynamic reconfiguration system – ensuring minimum level of functionality maintained in presence of failure.

REFERENCES

[1] PlanAhead Design and Analysis Tool, Xilinx, http://www.xilinx.com/tools/planahead.htm

[2] P. Graham, et al.," Consequences and Categories of SRAM FPGA Configuration SEUs", in Military and Aerospace Programmable Logic Devices International Conference. 2003. 1-9.

[3] H. Quinn, et al., Radiation-Induced Multi-Bit Upsets in SRAM-Based FPGAs. IEEE Transactions on Nuclear Science, 2005. 52(6): p. 2455-2461.

[4] P.Conmy, C.Pygott, I.Bate, "VHDL Guidance for Safe and Certifiable FPGA Design", IET System Safety Conference, October 2010.

[5] Ministry of Defence, "Safety Management Requirements for Defence Systems, Part 1 Requirements (00-56)," Ministry of Defence, ed., U.K. Ministry of Defence, 2007.

[6] RTCA/EUROCAE, DO-254 "Design Assurance Guidance for Airborne Electronic Hardware", RTCA, 2000.

[7] IEC, "Functional Safety of electrical/electronic/programmable electronic safety-related systems (IEC 61508)", 2010.

[8] H. Aysan, S. Punnekkat, and R. Dobrin, "VTV – A Voting Strategy for Real-Time Systems", 14th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'08), Taipei, Taiwan, December, 2008.

[9] A. Avizienis, J.C. Laprie, and B. Randell. "Fundamental Concepts of Dependability". in Third Information Survivability Workshop. 2000. Boston, Massachusetts, USA: IEEE.

[10] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, "Optimization by Simulated Annealing". Science. New Series 220 (4598): 671–680, 1983.

[11] M. Nicholson, "Selecting a Topology for Safety-Critical Real-Time Control Systems", DPhil, University of York, 1998.

[12] P. Emberson, I. Bate: Stressing Search with Scenarios for Flexible Solutions to Real-Time Task Allocation Problems. IEEE Trans. Software Eng. 36(5): 704-718 (2010)

[13] P. Emberson, I. Bate: Extending a Task Allocation Algorithm for Graceful Degradation of Real-Time Distributed Embedded Systems. IEEE Real-Time Systems Symposium 2008: 270-279.

[14] J. Lach, W.H. Mangione-Smith, M. Potkonjak," Low Overhead Fault-Tolerant FPGA Systems", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 5, 212-221, 1998.

[15] D. Chen, J. Cong, Y. Fan, "Low-Power High-Level Synthesis for FPGA Architectures", Proceedings of the 2003 International Symposium on Low power electronics and design, 134-139, 2003.

[16] P. Conmy, I. Bate, Component-Based Safety Analysis of FPGAs, IEEE Transactions on Industrial Informatics, Vol 6, No 2, May 2010. pp 195-205.

[17] K. Morris, Upset with Neutrons, FPGA journal. 2004
http://www.fpgajournal.com/articles/20040420_seus.htm

TABLE VI. RESOURCE USAGE FITNESS PENALTIES

Category	Penalty	Criteria - %age of cells used on FPGA
Maximum	100	Less than 15% and over 80%
Medium	50	Between 15 and 45
Low	25	Over 45 and up to 60
Minimum	0	Over 60 and up to 80

TABLE VII. RELIABILITY TARGET FITNESS PENALTIES

Category	Penalty	Criticality	Criteria – closeness to required probability
Maximum	100	Catastrophic/Hazardous	More than 5 times the probability target
		Major	More than 10 times the probability target
		Minor/none	More than 15 times the probability target
		Minor/none	0.1 times under the probability target
Large	80	Catastrophic/Hazardous	More than 1.1 times the probability target
		Major	More than 5 times the probability target
		Minor/none	More than 10 times the probability target
		Major	0.005 times under the probability target
Medium	40	Catastrophic/Hazardous	More than 1.01 times the probability target
		Major	More than 1.5 times the probability target
		Minor/none	More than 5 times the probability target
		Catastrophic/Hazardous	0.0005 times under the probability target
Low	20	Catastrophic/Hazardous	More than 1.001 times the probability target
		Major	More than 1.1 times the probability target
		Minor/none	More than 1.6 times the probability target
Minimum	0	All	Between the low over and under targets.

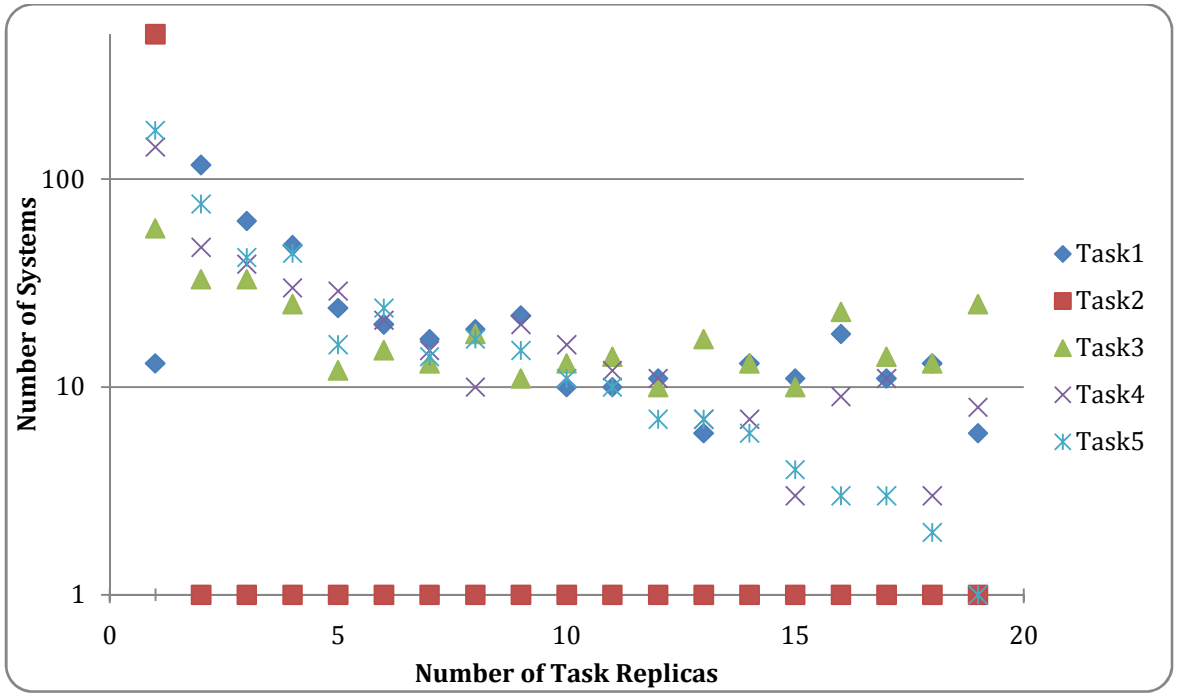

Figure 3. Scatter graph of total numbers of tasks for System 1

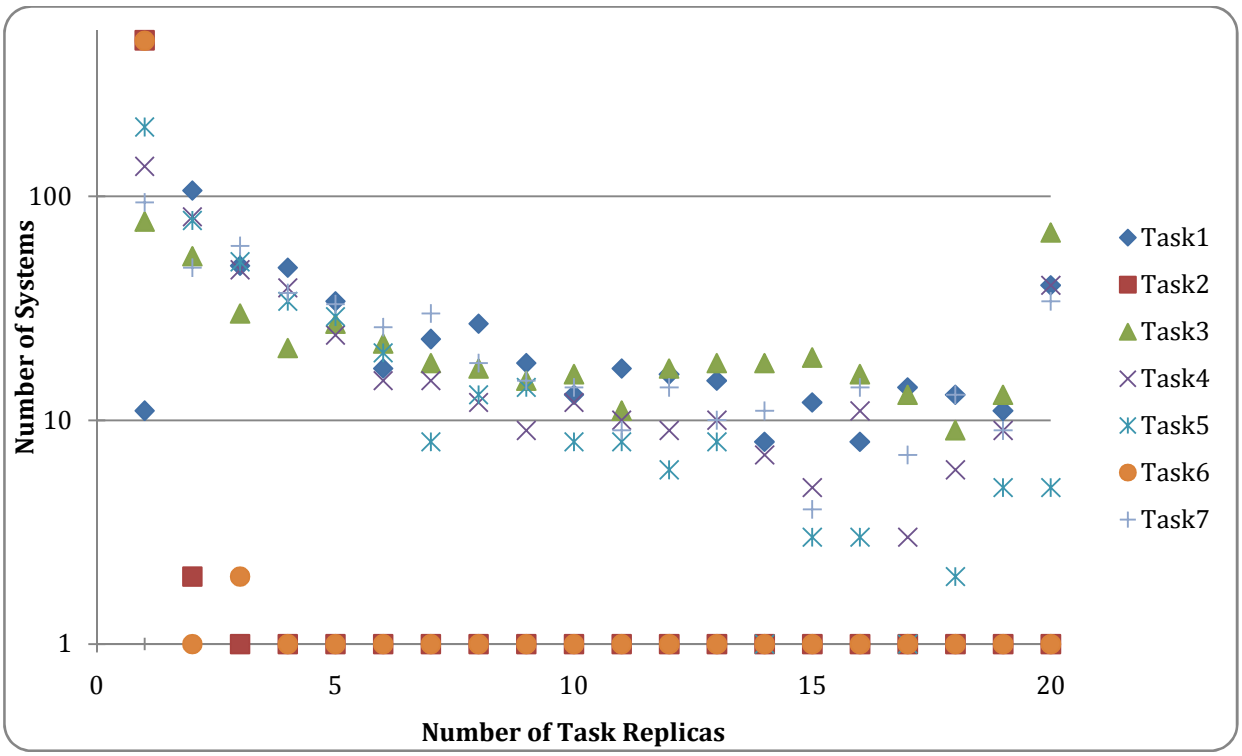

Figure 4. Scatter graph of total numbers of tasks for System 2