A Dependability-Driven System-Level Design Approach for Embedded Systems

Arshad Jhumka
Dept of Computer Science
University of Warwick
England
arshad_jhumka@hotmail.com

Stephan Klaus
Dept of Computer Science
TU Darmstadt
Darmstadt, Germany
klaus@iss.tu-darmstadt.de

Sorin A. Huss
Dept of Computer Science
TU Darmstadt.
Darmstadt, Germany
huss@iss.tu-darmstadt.de

Abstract

The objective of this paper is to introduce dependability as an optimization criterion in the system-level design process of embedded systems. Given the pervasiveness of embedded systems, especially in the area of highly dependable and safety-critical systems, it is imperative to directly consider dependability in the system level design process. This naturally leads to a multi-objective optimization problem, as cost and time have to be considered too. This paper proposes a genetic algorithm to solve this multi-objective optimization problem and to determine a set of Pareto optimal design alternatives in a single optimization run. Based on these alternatives, the designer can choose his best solution, finding the desired tradeoff between cost, schedulability, and dependability.

1 Introduction

Computer systems are becoming increasingly pervasive such that our reliance on their continual provision of correct services, in spite of external perturbations such as node crashes, has correspondingly increased. This means that we, as users, want these systems to be *dependable* [13]. These computer systems are being deployed in a wide range of areas, from safety-critical systems such as nuclear power plants, fly-by-wire systems to consumer-oriented products such as cars, mobile phones, washing machines etc.

Focusing on consumer-oriented products, cost is a critical issue. Where for safety-critical systems, dedicated processors can be used to execute safety-critical applications, in consumer-oriented products such as cars, safety-critical applications, such as the braking system, may well execute on the same processor as a non-safety-critical application, e.g., temperature control, so as to keep the cost factor tractable. Such a design approach is already taking place in so-called Integrated Modular Avionics (IMA), where safety-critical applications are being executed on the same platform as non-safety-critical applications. Designing dependable systems using such a methodology entails ensuring that non-safety-critical applications do not interfere with the safety-critical ones. Rushby [15] pointed out that, for this approach to work, system partitioning along two dimensions is needed: (i) spatial, and (ii) temporal. In this work, we focus on temporal partitioning, and we assume spatial partitioning.

The concern of temporal partitioning is to ensure that activities in one partition do not interfere with the timing of events in other partitions. In this paper, since tasks of different 'criticalities' may execute on a given processor, we assume that each partition has to contain one task. As suggested by Rushby in [15], static scheduling can be used at the level of partitions, while dynamic scheduling can be used within partitions. Since a partition consists of a single task, in this paper, static scheduling is used for temporal partitioning.

Designing for dependability in the latter stages of the design phase may lead to a *complete redesign* of the system in the worst case, or can be very *costly* (as we will show in Sec. 5). This means that dependability aspects must be considered during the earlier part of the design process. Hence, there are certain dependability-driven factors that need to be taken into consideration earlier on. One of these factors is *replication*. Replication of the more critical tasks is needed to ensure that, in case of a failure of a node upon which a critical task resides, there is at least one other replica of that task still executing to take over.

Replication of the more critical tasks makes the system more dependable, however this may come at the expense of cost (e.g., more resources may be needed to accomodate the extra tasks), and schedulability. This leads to a multi-objective optimization problem (between cost, time and dependability). To address this, we propose a genetic algorithm to solve the multi-objective optimization problem, whereas a set of Pareto optimal design alternatives is generated in a single optimization run. The system designer can then choose the solution that best suits the system's re-

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

quirements.

The remainder of the paper is organized as follows. Sec. 2 discusses the related work. In Sec. 3, our system model is presented and the approach to increase the dependability of embedded systems is introduced. Sec. 4 discusses the proposed genetic design space exploration. Sec. 5 provides some experimental results and some insights. We conclude in Sec. 6.

2 Related Work

For reasons of space, we will focus in the sequel on work related to the provision of dependabilty in embedded systems.

It has been shown that making a given system dependable requires that the given system has sufficient redundancy [7]. There are two dimensions to be considered, namely: (i) redundancy in space, and (ii) redundancy in time. For example, to support a single node failure, there should be at least two nodes to support continuous service provision (space), or in order to mask a value failure, a reexecution might be needed (time). In this work, we consider the dependability aspect during the design phase and incorporate the necessary redundancy (spatial and temporal) so as to make the system dependable.

[16] developed a fault containment-based approach, whereby applications are first decomposed into their constituents tasks and procedures, which are subsequently transformed to create fault-containment blocks. These blocks are later re-integrated to form dependable applications, which are then allocated to nodes, and scheduled. This approach works when the source code is available, but it is not readily usable at the system design level. A similar approach has been adopted by [3]. Up to now, very little work has addressed the problem of integrating dependability as an optimization criterion. [9] developed an approach in which processes/tasks are replicated until the overall dependability utility of the system is maximized. However, this comes at the expense of cost, which is an important constraint in embedded system design. Also, they fail to model factors such as importance.

System-level synthesis approaches based on task graphs can be found in [6] and [14]. A comparison of simulating annealing, genetic algorithm and tabu search for the functional HW/SW partitioning using a single optimization criterion is presented in [1]. [4] and [11] address genetic algorithms for multi-objective system-level synthesis. But all these approaches do not consider dependability at all. This is done by [10], who introduces fault tolerance as a parameter for the design of reconfigurable hardware. Overall, there is a dearth of work that addresses dependability as an optimization criterion in embedded systems design.

A good survey and a detailed description of specification models for embedded systems can be found in [2]. Next to state-oriented models and to abstract models like UML, task graphs are a widely spread means for the specification of data-flow dominated embedded systems. Beside the data-flow, most embedded systems contain at least a few control-flow decisions. Therefore, a lot of research was done extending task graphs by control- flow information [5, 12].

3 Models

System Model: We assume the following model in this paper. A system S consists of a set of n applications, $A_1 \dots A_n$, and can be represented by a directed graph. Each node in the graph represents an application. An edge from node A_j to node A_k represents data flowing from application A_j to A_k. Each application A_i consists of a set of k_i tasks, $t_i^1 \dots t_i^{k_i}$. It can be represented by a task graph, where each node in the graph represents a task, and the edges between the nodes represent data dependencies between these tasks. By unwrapping each application node with their respective task graph, the system can be represented as a task graph, where nodes denote tasks and edges represent data between tasks. Thus, S consists of m tasks, $T_1 \dots T_m$, respectively.

We denote by R the set of available resources. A cost factor v_i is associated with every resource $r_i \in R$. Every time an additional resource $r_j \in R$ is used, the overall system cost increases by v_j. Each task t_i can be represented as a tuple (R_i, E_i, C_i), where

- $R_i \subseteq R$ is the set of resources task T_i can use.
- $E_i \subseteq R_i \times Z$ is the execution time of task T_i when using a resource in R_i (modeling heterogeneity).
- C_i represents the importance of the task. Importance is necessary when dependability is taken into account. The higher the importance of a task, the more critical it is for providing correct services.

There are two possible ways to model replication. First, we can associate a *replication factor* with every application of the system. The higher the replication factor, the more critical the application is for the system. Suppose a system designer decides to assign a replication factor of 2 to a given application. We can then duplicate every task of that application. Second, since a system can be represented as a task graph, we can directly associate a replication factor with every task. In this work, we adopt the second approach. Since every task in an application is possibly not critical, it is not necessary to replicate all the tasks of that application. Hence, the ability to selectively replicate tasks is crucial.

When replicating a given task, then the initial task graph (hence called task graph) needs to be extended into a *repli-*

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

cation task graph. The replication task graph is constructed from the task graph as follows (Fig. 1):

- For every edge (n_1, n_2) pointing to a node n_2 in the task graph, there is a an edge (n_1, n_2) pointing to every replicated node n_2 in the replication task graph.
- For every edge (n_1, n_2) starting at node n_1 in the task graph, there is an edge (n_1, n_2) starting at every replicated node n_1 in the replication task graph.

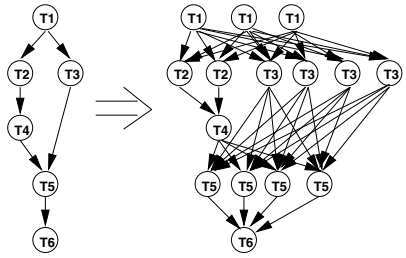

Figure 1. Initial and Replication Task Graph

When replication is introduced in the system-level design process, then another constraint regarding the binding has to be taken into account. If a task is duplicated in order to increase the dependability of the system, then it makes no sense to bind the replicated task to the same resource as the original task. If a failure occurs, then the replicas should not be affected by this failure. So, each replication task of T_i has to be bound to a resource $r \in R_i$ in such a way that no other replica of T_i is bound to the same resource.

Based on failure probability, importance, and replication, the dependability is calculated as follows:

$$Dependability = \sum_{i=1}^{T} C_i * (1 - {p_i}^{f_i}) \qquad (1)$$

whereas:
T: number of tasks; f_i: replication factor of task t_i;
C_i: importance of t_i; p_i: failure probability of t_i.

We assume that the replication factor, and importance of a task to be provided by the system designer. We further assume that the failure probability of a task is equal to the failure probability of the node it is running on, and the probability of failure of the node can be obtained from field data.

Fault Model: Since we are concerned with enhancing dependability via replication, the underlying fault model is resource crashes.

4 System-Level Synthesis

The purpose of system-level design or design space exploration is to determine a set of suited implementation alternatives. This process can be detailed as the selection of the necessary resources (allocation) and the mapping of the functional units onto the selected architecture in space (binding) and time (scheduling). Naturally, this is a multi-objective optimization problem, since const, time, and especially dependabilty are taken into account. Therefore, no universal definition of an optimum can be given. One solution for an appropriate definition of optimality in multi-objective optimization was given by Vilfredo Pareto in 1896. This definition expresses basically that a solution is *Pareto-optimal* if there exists no feasible other solution, which would decrease some objective without causing a simultaneous increase in at least one other objective.

Figure 2 illustrates a solution space S for two optimization objectives f_1 and f_2, respectively. The solution space is restricted by the design constrains (dashed line) and by the Pareto-front (solid line), which consists of all Pareto-optimal solutions. Among the Pareto-optimal solutions S1 to S3, S1 is the best one according to $f1$ and S3 according to $f2$, respectively. S2 dominates S1 with respect to $f2$ and S3 regarding to $f1$.

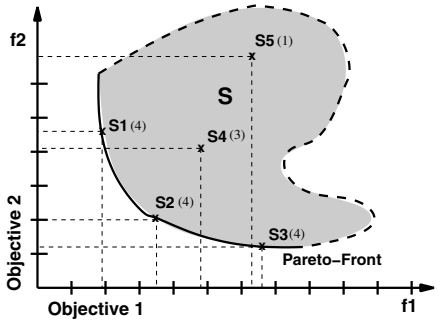

Figure 2. Illustration of Pareto-Front

4.1 Genetic Design Space Exploration

Genetic algorithms are choosen to solve this complex multi-objective optimization problem, since a complete search is in general not feasible due to the size of the solution space. This optimization heuristic [8] is inspired by Darwin's theory of evolution and is very well-suited for problems with large, non-convex search spaces and multi-objective optimization goals.

Basically, a set of possible solutions (the *population*) is modified by *crossover, mutation*, and *selection* operations. Thereby, the selection is controlled by a *fitness function*, which measures the quality of each individual (one possible solution) in the population. The general flow is illustrated in Algorithm 1. First, the population is initialized by assigning randomly some values to each individual. This population is evaluated by a Pareto ranking, and then a subset of

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

the population is mutated according to a given probability. Subsequently, new individuals are generated by crossover operations, which replace randomly individuals in the current population.

Algorithm 1 Genetic Design Space Exploration

```
1:  for (j=0;j<PopulationSize;j++) do
2:     newInd.Init
3:     P = P ∪ newInd
4:  end for
5:  while STOP=false do
6:     PerformParetoRanking(P)
7:     for all ind ∈ P' ⊂ P do
8:        ind.Mutate()
9:     end for
10:    for all i1, i2 ∈ P' ⊂ P do
11:       newInd.Crossover(i1, i2)
12:       newGen=newGen ∪ newInd
13:    end for
14:    P ∪ newGen \ worstIndividuals
15: end while
16: Print Pareto Solutions P
```

4.1.1 Representation

Each individual represents one possible solution for the optimization problem of embedded system design. Therefor, a coding has to be found, which is suited to represent all design alternatives of the envisaged solution space. First of all, an individual must hold information about a complete implementation of an embedded system (i.e., allocation, binding, and scheduling). Allocation is being represented by a set of resources on top of which the system can be implemented. The binding assigns a resource instance to each task considering both the current allocation and the possible binding alternatives of the task. The schedule is represented by unique priorities, which are utilized in the evaluation phase to determine the start and end times for all tasks. In addition, information about replication factors of each task is assigned to each individual.

Figure 3 depicts the representation of the replication factors for the example of Figure 1. It can be easily seen, that the list contains a replication factor for each task.

4.1.2 Genetic Operators

The genetic operators for initialization, mutation, and crossover are divided into parts for allocation, binding, scheduling, and replication factors too. The operators for allocation, binding, and scheduling are working on the extended task graph including the replicas according to the replication factors.

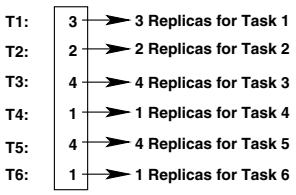

Figure 3. Genetic Representation of Replication Factors

The population is initialized by randomly generating some individuals, whereas only feasible solutions are generated. For the initialization of the allocation a subset of all avaliable resources is chosen. The binding assigns randomly a resource instance to each task considering the current allocation. The schedule is initialized by simply giving each task an unique priority and the replication factors recive a value between 1 and a maximal replication count.

The mutation and the crossover operators for the replication factors are illustrated in Figure 4. Mutation chooses randomly a position and changes the replication factor for this task. For the crossover the lists of replication factors are splitted into to sets: In the first set, the factors according to the first parent are taken. The second set is generated accordingly from the data of the second parent. For the second child the roles of the parents are swapped.

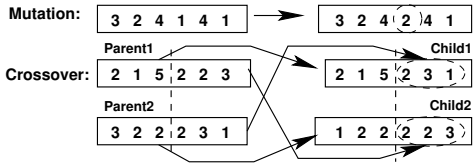

Figure 4. Genetic Operators

The allocation is mutated by removing or adding randomly some resources. As a result, the allocated set may be too small, so that tasks exist, for whom no resource is avaliable. In contrast to "punishing" such individuals, the allocation is repaired by adding randomly missing resources. The binding is mutated by assigning a different resource from the allocation to some tasks, if possible. For mutating the schedule, the priorities of two tasks are randomly exchanged.

The crossover of the allocation is done by choosing randomly resources from one parent and by completing this set by resources from the other parent. For the crossover of the binding the sets are divided into two parts: In the first set, it is tried to bind each task according to the binding of the first parent. If this is not possible, then the second parent is taken. In the second part the same is done with swapped

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

roles of the parents. Crossover of schedules is done by dividing the tasks into three parts. Hence filling the middle with the priorities of one parent, and the first and last part with the priorities of the other parent, if they were not used yet. The rest of the data structure is filled with the remaining priorities.

4.1.3 Evaluation

In the evaluation phase the costs, the dependability and both the start and end times of all tasks are calculated. The cost of a single solution is determined by summing up the costs of all used resources. The dependability is calculated according to eq. (1) considering importance, failure probability, and replication factors of the tasks. A scheduling based on the determined binding and on the calculated priorities results in exact execution times for each task. So, the overall execution time and possible deadline misses of a single task can be derived. The scheduling is done by globally ordering the tasks on each resource according to the data dependencies and the priorities. This circumvents the disadvantage of list scheduling algorithms. These algorithms unveil the problem of *timing anomalies*. Such a anomaly occurs in the case that the overall performance of a system increases, if a task is not started immediately after this tasks becomes ready and the resource is available.

The evaluation is based on a Pareto ranking, which sorts all solutions according to the amount of individuals which are not dominated by this solution. The resulting scores of each solutions are depicted as attributes in Fig. 2. E.G., $S5$ is not dominated by $S3$, but by all other solutions. At the end of the design space exploration the complete set of Pareto-optimal solutions is passed to the designer, the so-called human decider, who can now select *his best* design solution.

5 Results

This section demonstrates the figures of merit of the proposed approach, by means of a small but expressive example. Here, we take $|R| = 1$. The advantages of directly considering dependability in the system level design process are illustrated by means of the example of Figure 5. The results of a classical design approach are directly compared in the following to our comprehensive system-level synthesis approach.

Figure 6 depicts an implementation variant generated by a classical design space exploration without considering dependability at all. This implementation exploits the resources $R1$ and $R2$ resulting in overall costs of 270. Faster, but more expensive solutions exist when exploiting $R2$ and $R3$.

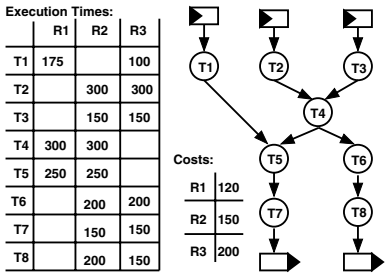

Figure 5. Example Task Graph

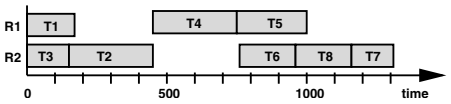

Figure 6. Implementation resulting from Classical System Synthesis

Late replication v/s our approach: Dependability issues are being addressed by many designers of new systems. Conceptually, dependability is treated as an "add-on", e.g., it is accomplished by doubling the complete implementation. The resulting system has a dependability value of 6 according to eq. (1), but the cost is doubled too. This situation occurs when dependability is considered late in the system design phase.

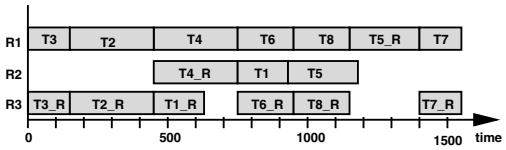

Figure 7. Dependable Implementation

On the other hand, Figure 7 shows an implementation that is derived from applying our proposed approach, assuming all tasks to have the same importance value (hence, same replication factor). A dependability value of 6 is achieved too, but by exploiting only three resources (rather than 4). This results in system cost of 470 instead of 540 for the naive, i.e., doubled version. All Pareto optimal solutions generated by the proposed genetic algorithm are presented in Table 1. As it can easily be seen, the solution space spans from of expensive and highly dependable systems to cheap and fast implementations without any dependability. The implementation presented in Figure 6 is not part of this list, since a dependability value of 4.25 can be reached without increasing the costs or the overall execution time of the system (alternative 9). *This result shows that it pays off to*

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

consider dependability issues early in the design phase, as opposed to *a-posteriori* add-ons.

Alt.	Execution Time	Cost	Dependability
1	1550	270	4.5
2	1025	670	5.5
3	1000	390	4.50
4	1150	590	5.50
5	1150	390	5
6	1350	440	5.75
7	1300	540	6.0
8	1550	470	6.0
9	1300	270	4.25
10	1550	320	5.25

Table 1. Implementation Alternatives

Different replication factor: A design space exploration was performed by applying a maximal replication factor of 4. The importance parameter C_i of tasks $T4$ and $T5$ is set to 2, as these tasks are considered as the most critical once. and therefore should have a higher replication factor. The replication factor for each task are given in Table 2. The resulting implementation exploits four resources, i.e., two $R1$, one $R2$ and $R3$. The overall cost sums up to 570, the dependability value results in 6.375, and the overall execution time is 1300. Compared to previous cases (for the same resource cost)we only pay an extra 13% time for a gain of more than 16%. *This result shows again that early consideration of dependability is important, as well as an individual assignment of replication factors to tasks, rather than to applications (finer grained). This holds especially for safety-critical systems*, where resources are abundant. Obviously replicating non-critical tasks does not necessarily lead to a significant increase in dependability.

Task	1	2	3	4	5	6	7	8
Replication Factor	2	1	2	3	4	2	3	3

Table 2. Replication Factors

6 Conclusion

This paper presents a novel system-level design method for embedded systems which considers dependability as an optimization criterion. Replication is introduced at task level to increase the dependability of the system. Further, instead of determining a single solution only, a set of Pareto optimal design alternatives is calculated by the proposed genetic algorithm in a single optimization run to help the system designer in finding the required tradeoff between runtime, costs, and dependability. As demonstrated by the detailed example, taking dependability directly into account leads to superior implementations. Currently, we allow tasks of different applications to be potentially distributed across different processors. In other cases, this may not be feasible, since it does not provide fault containment. So, a subject of future work will be system design under more complex failure models.

References

[1] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies. In *Proc. of 5th International Workshop on Hardware/Software Co-Design*, Germany, March 1997.

[2] L. A. Cortés, P. Eles, and Z. Peng. A survey on hardware/software codesign representation models. Technical report, Linköping University, June 1999.

[3] B. P. Dave and N. Jha. Cofta: Hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance. In *Proc. FTCS*, pages 417–441, 1997.

[4] R. P. Dick and N. K. Jha. MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-synthesis of Hierarchical Heterogeneous Distributed Embedded Systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, 1998.

[5] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *IEEE/ACM Proc. Conference on Design, Automation and Test in Europe*, pages 132–139, 1998.

[6] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search,. *IEEE/ACM Proc. Conference on Design Automation for Embedded Systems,*, 2:5–32, 1997.

[7] F. Gaertner and H. Volzer. Redundancy in space in fault-tolerant systems. Technical Report TUD-BS-2000-06, Department of Computer Science, TU - Darmstadt, 2000.

[8] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[9] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and Krithi Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proc. IEEE RTSS*, pages 79–89, 1997.

[10] C. Haubelt, D. Koch, and J. Teich. ReCoNet: Modeling and Implementation of Fault Tolerant Distributed Reconfigurable Hardware. In *Proc. of 16th Symp. on Integrated Circuits and Systems Design*, pages 343–348, São Paulo, Brazil, September 2003.

[11] C. Haubelt, S. Mostaghim, F. Slomka, J. Teich, and A. Tyagi. Hierarchical Synthesis of Embedded Systems Using Evolutionary Algorithms. In *Evolutionary Algorithms for Embedded System Design*, Genetic Algorithms and Evolutionary Computation, pages 63–104. Kluwer Academic, 2003.

[12] S. Klaus and S. A. Huss. A Novel Specification Model for IP-based Design. In *Proc. of EUROMICRO Symposium on Digital System Design*, pages 190–196, Turkey, September 2003.

[13] J. C. Laprie. *Dependability: basic concepts and terminology*. Springer Verlag, 1992.

[14] M. López-Vallejo and J. C. López. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Trans. on Design Automation of Electronic Systems*, 8(3):269–297, July 2003.

[15] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, NASA Contractor Report CR-1999-209347, 1999.

[16] N. Suri, S. Ghosh, and T. Marlowe. A framework for dependability driven sw integration. In *Proc. Distributed Computing Systems*, pages 405–416, 1998.

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE