Hardware-Software Co-Synthesis of Fault-Tolerant Real-Time Distributed Embedded Systems*

Santhanam Srinivasan and Niraj K. Jha

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

Abstract

Distributed systems are becoming a popular way of implementing many embedded computing applications, automotive control being a common and important example. Such embedded systems typically have soft or hard performance constraints. The increasing complexity of these systems makes them vulnerable to failures and their use in many safety-critical applications makes fault tolerance an important requirement. This paper is the first to address the problem of automatic hardware-software co-synthesis of fault-tolerant embedded distributed real-time systems in a generalized scenario. We present an algorithm which takes as input the specification of the data-flow information in the form of a task graph, the performance constraints, the fault tolerance requirements and the available hardware resources in the form of processor, ASIC and link libraries. Our algorithm then synthesizes the required hardware as a distributed system in terms of the component processors, ASICs and interconnection links. The tasks are mapped to this hardware such that the overall system cost is minimized while still meeting the performance constraints and the fault tolerance requirements. Our algorithm uses clustering techniques to perform the synthesis. Fault tolerance is added using CRAFT*, a technique we recently proposed for CRiticAlity based Fault Tolerance in real-time distributed systems.*

1 Introduction

An embedded computing system consists of a hardware engine (typically a distributed architecture) executing application software. The processing elements (PEs) of the hardware engine may be general-purpose CPUs, DSPs, *etc.* which execute the application software, or they may be application-specific ICs (ASICs) implementing portions of the functionality in custom hardware. The different PEs are connected together by an interconnection network of links and/or buses. The functionality of the system is modeled as a set of periodically executing modules which we term *tasksets*. Each taskset is composed of a number of intercommunicating sequential processes or *tasks*. The structure of the hardware engine makes up the *hardware architecture* and the mapping (or *allocation*) of the different tasks and the inter-task communication onto the different components of the hardware engine results in the *software architecture*. The design of embedded systems involves the co-synthesis of the hardware and software architectures.

The specification of the embedded system consists of two parts [1, 2]. The *functional* aspect of the specification describes the tasksets, their component tasks and the inter-task communication. Also specified are the characteristics of the different tasks, like the different PEs capable of executing them and the execution times on these PEs, inter-relationships among tasks and their communication characteristics, like the volume of data transferred from one task to another. The *non-functional* specification includes soft and hard performance constraints or *deadlines* on the tasksets, giving them a certain amount of time to compute the *correct output*. System failures could occur either due to missing a deadline or producing an incorrect result or both. The increasing use of such systems in critical applications means that fault tolerance is a necessity.

In our work, we aim at making the finally synthesized embedded system fault tolerant. To this end, we need to introduce some redundancy into the system. However, in a real-time scenario, care should be taken that the overhead due to fault tolerance does not imperil the meeting of deadlines. For this purpose, we use the concept of *CRAFT* [3], such that we can maximize the amount of fault tolerance while still not violating the performance constraints. We present a comprehensive scheme to accept a very general specification of the embedded system in terms of the functional and non-functional components with arbitrary resource constraints on the tasks. The available resources are specified as processor, ASIC and link libraries. From this description, our algorithm simultaneously synthesizes the hardware and the software architectures such that the overall system has the specified fault tolerance and meets the performance constraints while minimizing the system cost.

Traditional work in real-time and distributed systems [3, 4] assumes that the hardware configuration is specified and tries to find a mapping of the tasks

*Acknowledgments: This work was supported in part by National Science Foundation under Grant no. MIP-9423574.

to the hardware and assign time slots to the tasks and inter-task communication in order to meet the deadlines. Since the hardware engine is yet unspecified, compared to these approaches, the problem of co-synthesis is much more difficult. Related work in hardware-software partitioning has generally assumed a one-CPU-one-ASIC architecture and tried to move operations from hardware to software to minimize cost [5], or *vice versa* to satisfy performance goals [6]. In [2], work which is most relevant to ours, the authors have given an iterative procedure to synthesize an arbitrary distributed embedded system from the specification. However, they assume the PEs to be off-the-shelf processors and do not allow ASICs. Also, they do not consider communication scheduling. They also do not allow resource constraints on the tasks. Finally, they do not address the issue of fault tolerance. To the best of our knowledge, there has been no work in solving the co-synthesis problem in the generalized scenario considered by our algorithm.

2 Definitions and Basic Concepts

In this section we explain the basic terminology and concepts needed to understand the rest of the paper.

Hardware Resource Specification: The final hardware architecture is assumed to be made up a collection of *processing elements* (PEs) interconnected via a network of arbitrary topology. Pairs of communicating PEs are assumed to do so via dedicated *links*. The PEs themselves may be off-the-shelf processors or DSPs or may be one of many types of custom designed ASICs. The different types of links available to build the interconnection network are specified in the form of a *link library*. The link library defines the properties of the set $\Lambda = \{\lambda_1, \cdots, \lambda_v\}$ of available types of links. An entry in the link library specifies: (a) the type of the link (RS232 or I^2C, for example) (b) the cost of such a link, and (c) delay to transmit a unit piece of data along the link (*unit link delay*). The set $\Pi = \{\pi_1, \cdots, \pi_k\}$ of available PE types is specified in a *PE library*. This library contains an entry for each available type of processor or ASIC which specifies (a) the type of the PE (*e.g.* Pentium, Xilinx *etc.*) (b) whether the PE is a processor or an ASIC (c) the cost of each PE (d) the number of ports the PE is capable of supporting, and (e) the types of links supported by each available port of the PE.

The main difference between a processor and an ASIC from the point of view of our co-synthesis algorithm is in the model for their costs and for the execution of tasks that are allocated to them. Consider tasks t_1 and t_2 that have been allocated to PE p. Assume that t_1 and t_2 do not have any precedence constraints between them. If p were a processor, its cost would be independent of the tasks that are allocated to it. Also, if we consider only uniprocessors, the execution of the two tasks cannot go on simultaneously and only one of them can execute at a given moment on p. In contrast, if p were an ASIC representing a custom hardware implementation of the two tasks, then the cost of each ASIC would be dependent on the *total* functionality implemented and hence a *function of all the tasks allocated to it.* Besides, the ASIC can be implemented such that the execution of the two tasks can proceed in parallel.

In view of the above distinctions, for the ASIC entries, the PE library also has to specify the variation of the cost of the ASIC as a function of the functionality implemented, *i.e.* the tasks allocated to the ASIC. In our experiments, as a first order approximation, we assume the cost of the ASIC to be proportional to the sum of the execution times of all the tasks allocated to the ASIC. Hence, the PE library can specify the cost function by supplying the appropriate proportionality constant for the different ASICs. Note, however, that our co-synthesis algorithm itself does not depend on this assumption.

Software Specification: The functional specification of an embedded system consists of a set of modules called *tasksets*. All the tasksets are collectively referred to as the *job*. Each taskset is composed of a number of communicating *tasks*. The communication pattern among the tasks imposes a precedence constraint on the tasks. In this work, the division of the tasksets into their component tasks is assumed to have been completed in the *task partitioning* step. Partitioning is done to expose the parallelism inside a taskset. Each taskset is assumed to execute periodically. The tasks and their intercommunication are represented by a task graph [7, 3], with nodes representing tasks and directed edges representing intercommunication. The task graph supplies the worst case execution time $\mathcal{E}_{t_i,\pi_j}$ of each task t_i on PE type π_j in the PE library. Allocation constraints on the different tasks are represented using *task preference* and *task exclusion* matrices, Pr and Ex, respectively [7]. $Ex[t_i, t_j]$ is '1' if task t_i cannot be allocated to the same PE as task t_j ('0' otherwise) and $Pr[t_i, t_j]$ is '0' if task t_i cannot be allocated to PE type π_j ('1' otherwise). The *comp_cost* of a task is the average execution time of the task on all the PE types it can be allocated to. We consider only periodic tasksets in this paper. Each taskset has a number of parameters associated with it. The *earliest start time* (*est*) is the time before which the taskset is not allowed to start, the *period* is the length of time after which the taskset repeats execution, and the *deadline* is the latest time by which the taskset has to complete execution for the system to perform correctly. All tasks inherit the period and the deadline from the taskset to which they belong. The *est* of a task t is the earliest time at which all the fanins of t can complete and t can start. The *latest start time* (*lst*) of t is the latest time when t can start and still enable all succeeding tasks in its parent taskset to finish before their deadline. *lst* is a function of all the tasks that depend, directly or indirectly, on t to complete before they can start executing. Note that both *est* and *lst* of a task are a function of the particular allocation since the actual execution time of each task and the cost of all inter-task communication are known only after the allocation is done. However, using best, average and worst case values for task execution times and communication costs, we can get a range of values for *est* and *lst* which are useful to the

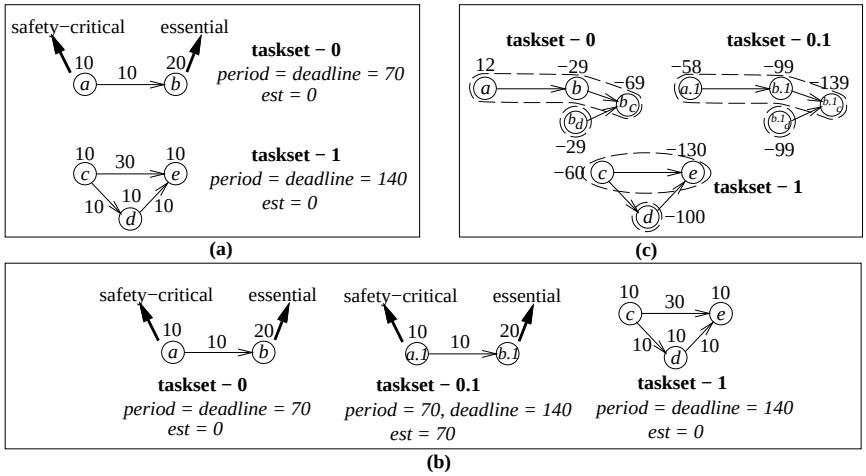

Figure 1: Example: (a) software specification (b) CTG formation (c) Fault tolerant CTG and clustering

co-synthesis algorithm, as described later. To obtain best-case values, for each task, the best computation time from among all the PE types in the system is used. All inter-task communication is ignored, as if every pair of communicating tasks were assigned to the same PE. For computing the average-case *est* and *lst* we use average task computation times and an average value for the link delay, considering all possible link types in the link library for the inter-task communication. For determining the worst-case values, worst-case task execution times and the worst link delay from the link library are used. Both *lst* and *est* are computed using a depth-first search on the task graph of each taskset.

Example 1 Figure 1(a) shows an example software specification. Taskset 0 has tasks a and b, and a deadline, period and *est* of 70, 70 and 0 respectively. Taskset 1 has tasks c, d and e, and a deadline, period and *est* of 140, 140 and 0 respectively. □

PE Graph: The PE graph is a convenient abstraction of a snapshot of the hardware architecture as it is being synthesized, with nodes representing the PEs already in the system and edges representing inter-PE links. The weight on the edge e_{ij} between PEs p_i and p_j, w_{ij}, is the delay of link e_{ij} represents. We assume that a PE can perform either a *send* or a *receive* to or from its neighbor at any one time, but not both. We assume that tasks co-allocated to the same processor do not incur any overhead when they communicate with each other. Given this scenario, in order to minimize inter-PE communication (IPC), we would like to ensure that heavily communicating tasks are allocated to the same PE. Our synthesis algorithm tries to do this by forming *clusters* of heavily communicating tasks and ensuring that all the tasks of a cluster are allocated to the same PE.

Assertions: We assume that a fault in a PE will result in an error in at least one task computation and that the failure of a link will corrupt at least one data item being sent along that link. In order to introduce safety in the system, each task in the task graph is either checked, whenever possible, by an assertion task or else is subject to duplication and comparison. For assertion checking, the original task is modified by encoding its data elements using a system-level code. The encoded output data elements are given to the assertion task which checks their encoding. If the encoding is not satisfied, it indicates the existence of an error and correspondingly, a faulty PE or link. The advantage in using assertions over full duplication is that assertion checking is usually much less expensive than duplication [7].

Clusters: A cluster is a collection of tasks, typically grouped together because they communicate heavily with one another. Our algorithm forms a cluster of tasks in order to reduce IPC by allocating all the tasks of a cluster to the same PE. The number of clusters can be an order of magnitude smaller than the number of tasks in the system. Using clusters, therefore, may drastically reduce the size of the search space to be explored by the co-synthesis algorithm.

3 Hardware-Software Co-synthesis

The steps of our co-synthesis algorithm are described next.

Feasibility Check: First the best case *lst*'s and *est*'s for all tasks are determined. Denoting the best case *lst* and *est* of task t by l_b^t and e_b^t, respectively, if for any task t, $e_b^t > l_b^t$, then the job is declared to be infeasible.

Compound Task Graph (CTG) Formation: In [8], it is shown that there exists a feasible schedule for a job if and only if there exists a feasible schedule for the least common multiple (LCM), or *hyperperiod*, of the taskset periods. So, each taskset is replicated as many times as is required to fill up the hyperperiod. Let taskset s be a copy of taskset r. Let t_i^s and t_j^s correspond to tasks t_i^r and t_j^r of taskset r. Then $Ex[t_i^s, t_j^s] = Ex[t_i^r, t_j^r]$. Also, $Pr[t_i^s, \pi] = Pr[t_i^r, \pi]$ for every PE type π. In addition, t_i^s and t_j^s have the same criticality as t_i^r and t_j^r, respectively, as defined next.

Fault-Tolerant CTG: Tasks in real-time system are categorizable as *safety-critical*, *essential* and *non-essential* depending on the importance of the function they perform [4]. Failure of safety-critical tasks can be catastrophic and of essential tasks leads to performance degradation. Failure of non-essential tasks has

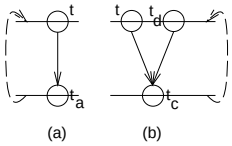

Figure 2: Recovery block using (a) assertion check (b) duplication and comparison

no immediate effect but has a long term impact.

For the safety-critical tasks we target transient faults and apply backward error recovery as follows. If an assertion t_a exists for the safety-critical task t then we modify t to produce encoded output [7]. Assertion task t_a is introduced into the system to verify the correctness of the encoding. If the assertion fails, t and t_a are re-executed. The number of such retries, r, is specified by the user. If the task is not assertible, a duplicate t_d and a comparison task t_c which checks the output of t and t_d for equality are introduced into a similar *backward recovery loop*. This is depicted in Figure 2. In order to avoid communication delays as part of the loop, to be able to derive a worst case bound on the execution time of the loop, we constrain all the tasks belonging to the recovery block to be allocated to the same PE. Under this constraint, if the tasks are scheduled preemptively, the recovery block is equivalent to a compound task t_r with a worst case execution cost equal to $\mathcal{E}_{t_r,\pi} = \mathcal{E}_{t,\pi} \times (r+1) + \mathcal{E}_{t_a,\pi} \times r$, on a PE of type π, when t is assertible. If not, the worst case bound becomes $\mathcal{E}_{t_r,\pi} = \mathcal{E}_{t,\pi} \times (2r+1) + \mathcal{E}_{t_c,\pi} \times r$. For r equal to 1, these expressions reduce to $2\mathcal{E}_{t,\pi} + \mathcal{E}_{t_a,\pi}$ and $3\mathcal{E}_{t,\pi} + \mathcal{E}_{t_c,\pi}$. So, for each safety-critical task t, our algorithm modifies the execution costs to the appropriate value. Note that these are worst case execution times, and since failures occur only infrequently, the actual execution times in most cases would be $\mathcal{E}_{t,\pi} + \mathcal{E}_{t_a,\pi}$ or $2\mathcal{E}_{t,\pi} + \mathcal{E}_{t_c,\pi}$.

We aim to make the execution of the essential tasks fault-secure. We do this by considering them in the order of a user defined priority and for each task t we introduce an assertion t_a or duplicate t_d and comparison task t_c, as shown in Figure 2, but without the backward error recovery loop, and allocate t to a different PE than t_a, or t_c and t_d. Under these circumstances, it can be shown that t's computation is fault-secure [7]. The cost of fault tolerance, $\texttt{ft\_cost}(t)$, is defined to be the sum of all the extra computation and communication to be added to the system to make task t fault-secure [3]. The user specifies a parameter, `frac_ft`, that controls the amount of fault tolerance added. Let us define `total_ft_cost` to be $\sum_{\text{essential tasks } t} \texttt{ft\_cost}(t)$. Those essential tasks t are made fault-secure, such that $\sum_t \texttt{ft\_cost}(t) \leq \texttt{frac\_ft} \times \texttt{total\_ft\_cost}$.

Deadline Based Cluster Formation: Each task is assigned a *deadline based static level* which is the length of the longest path (in terms of average execution costs of the tasks along the path and the communication) from the task to a sink task in its taskset minus the deadline. To start with, all tasks are marked `UNCLUSTERED` and repeatedly tasks which are exclusion and preference compatible with each other and lying on the longest unclustered path in the task graph (identified using static levels) are grouped into a cluster and marked `CLUSTERED`. The process terminates when all tasks are marked `CLUSTERED`.

Example 2 Figure 1(b) shows the result of CTG formation, which results in a new taskset 0.1 with *est*, period and deadline equal to 70, 70 and 140 respectively. Tasks $a.1$ and $b.1$ are safety-critical and essential respectively. After the application of backward error recovery, the execution cost of tasks a and $a.1$ are modified to 21 (assuming that the cost of the assertion task is 1). For tasks b and $b.1$ duplicate and comparison tasks are added. (10 units of data are assumed to be transferred from b and b_d to b_c.) The deadline based static levels are computed and the tasks are grouped into clusters. The result of static level assignment and clustering in shown in Figure 1(c). □

Preemption Priority Assignment: We adopt a fixed priority scheduling strategy [9]. Since the problem of determining the optimal priorities is hard for distributed systems, we use heuristics to assign priorities. Deadline based static levels, as defined earlier, are also an indication as to how early a task needs to be scheduled. So, we sort the tasks in the decreasing order of static levels and tasks earlier in the list are assigned a higher priority than later ones.

Worst Case Finish Time Estimation after Partial Synthesis: Suppose during the co-synthesis process we are trying to determine the best allocation for a cluster c. We would like to allocate c to that PE p that maximizes the probability of meeting the deadlines of all the tasksets. For this purpose, we make some pessimistic assumptions about the unallocated clusters and estimate the finish times of all the tasks as follows. We assume that each unallocated cluster c is allocated to that PE type π, denoted by $\texttt{worst\_PE\_type}(c)$, which maximizes $\sum_{t \in c} \mathcal{E}_{t,\pi}$. Also, all unallocated inter-cluster communication is assumed to be done over the slowest link in the link library. Under these assumptions we can compute worst case static levels and hence priorities for all the tasks. Then the worst case release time (the time at which t is ready to execute) $\tau_r^T(t)$ can be expressed as: $\tau_r^T(t) = \max_{e_i,t_j}(\tau_f^E(e_i \in \text{fanin edge}(t)), \tau_f^T(t_j))$, where $\tau_f^T(t_j)$ is the finish time of task t_j, which is of higher priority than t and is allocated to the same PE as t, and $\tau_f^E(e_i)$ is the finish time of the *send* of edge e_i which is computed as $\max_{e_j}(\tau_f^E(e_j)) + send(e_i) + recv(e_i)$, where e_j is an edge which may delay the scheduling of e_i. Finally, we can write $\tau_f^T(t) = \tau_r^T(t) + \mathcal{E}_{t,\pi_t}$. where π_t is either the PE type of the PE to which the cluster c_t, to which t belongs to, is allocated or is $\texttt{worst\_PE\_type}(c_t)$.

Delay Penalty Calculation: We use the concept of penalties [2] to handle soft and hard deadlines in our co-synthesis algorithm. Consider a task t with an *est* of τ_e, a soft deadline of $\tau_s \geq \tau_e$ and a hard deadline of $\tau_h \geq \tau_s$. Assume that t completes at time τ_f. Every task t has an associated *penalty function*, $\phi_t(\tau_f, \tau_e, \tau_s, \tau_h)$ which is a measure of the "goodness" of the finish time τ_f. Typically, $\phi_t(.)$ is 0 for $\tau_e \leq \tau_f \leq \tau_s$, as there is no penalty if the task meets its

soft deadline. Its value is positive and non-decreasing for $\tau_s \leq \tau_f \leq \tau_h$ (usually increasing moderately). Since finish times which fail to meet the hard deadlines are unacceptable, in this work, we set $\phi_t(.)$ to ∞ for $\tau_h < \tau_f < \infty$, but a function which rises very steeply beyond the hard deadline may be used.

Given the finish time $\tau_f(t)$, the delay functions $\phi_t(.)$, and the soft and hard deadlines, $\tau_s(t)$ and $\tau_h(t)$, of all tasks t in the system, the *delay penalty* of the system, `delay_cost`, is computed as `delay_cost` $= \sum_t \phi_t(\tau_f(t), \tau_e(t), \tau_s(t), \tau_h(t))$.

Co-synthesis Procedure: Our co-synthesis procedure CO-SYN() takes as input the clustered CTG, the preference and exclusion constraints of all the tasks and the PE and link libraries and synthesizes the hardware architecture as a PE graph and the software architecture as a mapping of the tasks in the CTG to the PEs in the system.

Suppose the CTG consists of the set $C = \{c_1, \cdots, c_k\}$ of k clusters. At any given point, the algorithm maintains a partial view of the system in terms of a PE graph with nodes representing the set $P = \{p_1, \cdots, p_m\}$ of the m PEs selected so far and a mapping to these PEs of a subset $C' \subseteq C$ of allocated clusters. Initially, both P and C' are set to $\emptyset$. Then for each unallocated cluster $c \in C$, CO-SYN() considers each of the existing PEs $p \in P$ as a potential choice for allocating c. PE p is examined for compatibility with c as follows.

- First p is examined for *link compatibility*. Let P_i be the set of PEs, $P_i \subset P$, $p \notin P_i$, which communicate with p, *i.e.*, $\forall p_i \in P_i$, there exists a link between p and p_i in the PE graph. Let P_j be the set of PEs not including p to which have been allocated tasks communicating with tasks in c. We define c to be link compatible with p iff for each $p_k \in (P_j - P_i)$[1], there exist free ports on p and p_k which can support a common link type.
- Next CO-SYN() verifies that all tasks in c are preference compatible with p and exclusion compatible with all tasks allocated to p.

If p passes both of the above tests, CO-SYN() proceeds to the next step; otherwise it examines the next PE in the system. Let the dollar cost of the partially synthesized system at this point be `cost_so_far`. The increase in `cost_so_far`, `cost`(c,p), if c is allocated to p, is 0 if p is a processor. Otherwise, it is the cost of the cluster c (sum of execution costs of all the tasks in c on p) times the unit cost of the ASIC type of p.

Assume that p^c_{best} is the best choice so far of a PE to allocate c to. Let `cost`(c, p^c_{best}) be the increase in `cost_so_far` if c is allocated to p^c_{best}. As long as no PE p^c_{best} has been found, `cost`(c, p^c_{best}) is ∞. The algorithm considers a new PE p for allocating c, only if `cost`$(c,p) \leq$ `cost`(c, p^c_{best}). If this is so, the worst case finish times of all the tasks in the system are estimated as explained earlier. The delay cost of the system, `delay_cost`, is computed. If `delay_cost` is less than ∞ (all hard deadlines are met), then, first CO-SYN() compares p and p^c_{best} based on the dollar costs and if they are tied in this respect, it compares them on the basis of the corresponding delay costs. This procedure is repeated for all already existing PEs in the system.

At this point, if a PE has been found to allocate c to at no extra dollar cost (*i.e.* c can be allocated to some existing *processor* and still meet the system deadlines), then c is allocated to this PE, the system cost is updated and CO-SYN() moves on to the next unallocated cluster. Otherwise, it goes through all existing PEs and tries to upgrade their PE types and their link configuration and attempts to allocate c to this upgraded PE. This is done as follows. For each existing PE, CO-SYN() upgrades the PE type. Then, for each possible combination of link settings it examines to see if the current settings are link, exclusion and preference compatible. If so, the dollar cost change and the delay cost of the system are computed. It keeps track of the best solution seen so far.

Finally, CO-SYN() tries to examine the possibility of adding a new PE to the system to allocate c to. It again examines each different type of PE and creates a dummy PE p_d of that type. Then for each possible combination of link settings, it marks c as temporarily allocated to p_d and again runs these settings through the link, exclusion and preference tests. Then based on the dollar and delay costs, it compares the current setting with the best choice see so far, and updates the best solution if necessary.

At this point, if a PE and its corresponding link settings have been found such that c is compatible with it, then c is allocated to that PE which is best in terms of dollar and delay costs. Then CO-SYN() moves on to the next unallocated cluster. If no suitable PE can been found, CO-SYN() declares failure.

4 Experimental Results

We generated random examples, each with 3 tasksets containing 3, 6 and 9 tasks respectively. The task graph connectivity (the probability of an edge in the task graph) was fixed at 0.4. In each task graph, 50% of the tasks were chosen to be assertible. Of the assertible tasks, 10% were chosen to be costly (where the assertion task requires all the inputs of the original task). The execution cost of the tasks was chosen uniformly in the range $[5, 195]$ (an average value of 100) and the communication weights in the task graph were chosen uniformly in the range $[1, 10]$. The deadline of each taskset was chosen as a multiple of the sum of the average execution time of the tasks in the taskset, an approach that has been taken earlier [4]. The amount of slack in the system can be varied by adjusting this multiplying factor. Note that in our case, since the fault tolerance tasks are not part of the initial specification of the task graph, they do not figure in the computation of the deadlines. In contrast, in [4], these tasks are part of the users specification and contribute in deciding the deadline. Due to this key difference, the multiplying factors used in the two cases are not comparable. We used a factor of 5 in our experiments. The task graphs were chosen from (a) binary trees (BTREE), (b) randomly

[1] For sets A and B, $(A - B) \doteq \{x : x \in A, x \notin B\}$.

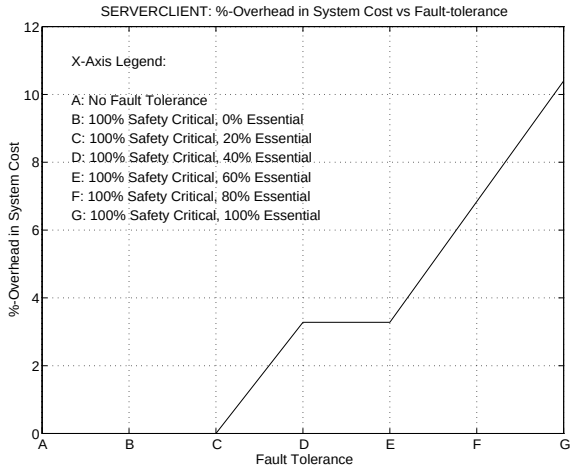

Figure 3: Variation of %-overhead in system cost versus fault tolerance for SERVERCLIENT graphs

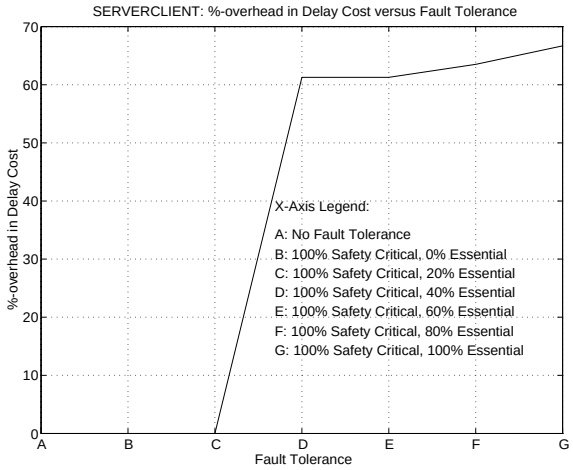

Figure 4: Variation of %-overhead in delay cost versus fault tolerance for SERVERCLIENT graphs

interconnected chains (CHAIN), (c) server-client formations (SERVERCLIENT), (d) graphs formed by interconnecting structures of types (a), (b) and (c) (BASIS), and (e) fully random graphs (BRAID). 200 random graphs were generated for each category. For each random example, a PE library consisting of two processors and two ASICs was randomly generated. For the purpose of experimentation, a simple link library with one link was chosen. Note, however, that the algorithm can handle a link library of arbitrary size. 10% of the tasks were chosen to be safety-critical and another 10% to be essential.

On an average, over all categories of task graphs, our algorithm found a solution with fault tolerance for 100% of the safety-critical and essential tasks for around 97.5% of the examples. For the cases in which it declared failure, due to the size of the examples (with 54 tasks in the CTG, not including the tasks added for fault tolerance), it was not possible to verify if there did actually exist a solution which was missed by our algorithm or whether it was an inherently infeasible system.

Figures 3 and 4 show the variation of the %-overhead in system and delay costs with respect to various levels of fault tolerance, for SERVERCLIENT graphs.[2] From these and other results, the following was observed.

- For every type of task graph, the system cost overhead for 100% fault tolerance (*i.e.* for 100% of the safety-critical and essential tasks) is marginal – around 24% in the worst case and less than 20% on an average. This reasonable penalty makes our technique for introducing fault tolerance in the system an attractive solution.
- It is not necessary that we have to pay a penalty in terms of system cost or delay cost for achieving fault tolerance. In some cases, the available slack in the system can be exploited at no extra cost to introduce fault tolerance and to maintain the performance (measured in terms of the delay cost) simultaneously. For example, in the case of SERVERCLIENT graphs, we can handle 100% of the safety-critical tasks and 20% of the essential tasks for no extra system or delay cost.

[2] Other results are omitted due to lack of space.

5 Conclusions

In this work we have presented the first algorithm capable of synthesizing generalized fault tolerant distributed hardware and software architectures from a real-time system specification. We have experimentally shown the efficacy of our scheme.

References

[1] W. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, pp. 967–989, 1994.

[2] T.-Y. Yen and W. Wolf, "Sensitivity driven co-synthesis of distributed embedded systems," to be presented at *Int. Symp. System Synthesis*, Sept. 1995.

[3] S. Srinivasan and N. K. Jha, "CRAFT: CRiticAlity based Fault Tolerance for distributed systems," tech. rep., CE-J95-02, Dept. of Electrical Engg., Princeton University, Princeton, NJ 08544, 1995.

[4] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," in *Proc. Int. Conf. Distr. Comput. Syst.*, pp. 108–115, June 1990.

[5] R. Ernst, J. Henkel, and T. Benner, "Hardware-software co-synthesis for microcontrollers," *IEEE Design & Test of Computers*, pp. 64–75, Dec. 1993.

[6] R. Gupta and G. De Micheli, "Hardware-software co-synthesis for digital systems," *IEEE Design & Test of Computers*, pp. 29–41, Sept. 1993.

[7] S. Yajnik, S. Srinivasan, and N. K. Jha, "TBFT: A task based fault tolerance scheme for distributed systems," in *Proc. ISCA Int. Conf. Parallel & Distr. Comput. Syst.*, pp. 483–489, Oct. 1994.

[8] E. Lawler and C. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Lett.*, vol. 12, no. 1, Feb. 1981.

[9] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, Jan. 1973.