

Design of Fault-Tolerant Distributed Control Systems

Vincenzo Piuri, *Member, IEEE*

Abstract—Real-time control systems, involving continuous measuring and monitoring of physical quantities, need often to take into account fault-tolerance capabilities to guarantee a correct behavior (even in the presence of a given amount of hardware faults and software errors) or, at least, to provide gracefully degradable performances.

In this paper, hardware dimensioning, the optimum allocation of the computation, and the fault-tolerance issues are afforded contemporaneously, with specific attention to the design of dedicated distributed control systems. A single optimization frame is defined to identify a globally optimum solution with respect to these conflicting goals.

I. INTRODUCTION

DISTRIBUTED systems are the best accepted solution to implement advanced dedicated control systems, involving both continuous observation of the controlled process through measurement and tracing of characteristic physical quantities, and execution of the related activities providing the proper behavior of the whole controlled process. Two problems must be solved in control system design: hardware dimensioning of each component, and software partitioning and allocation onto the architecture. A higher number of processing elements allows for exploiting computation concurrency despite a greater cost of the whole architecture; on the other hand, software partitioning, allocation and timing performances depend on the number of available processing units.

Fault tolerance issues have also to be faced whenever these systems are used in mission-critical applications (in particular, in real-time environments). In these cases, it is almost mandatory to detect the presence of faults and errors, to correct the erroneous outputs, and to survive (possibly with degraded performances) after some processing units have been declared faulty.

In the current literature, these problems are usually considered separately, e.g., by adopting graph theory [1], integer programming [2], and heuristic methods [3]. An integrated approach should consider all the above aspects at the same time to achieve the globally optimum solution. A first proposal in this direction was presented in [4] for integrating the system dimensioning and the software allocation. Contemporaneous solution of all the three design goals is discussed in [5]–[7], for different computational models and optimization algorithms. Fault tolerance and system survival are achieved by duplicating the software components in different processing units: when a unit becomes faulty, it is excluded from the

active computation and the spare copy of each software component running onto it is started on another processor. Only one fault at the time can be treated since the redundant allocations are statically precomputed. After each fault, the system is reconfigured and computation restarted; no subsequent fault can be managed until the system has been repaired. If more than one fault has to be tolerated, more than one spare copy of each software component must be preallocated. Static preallocation does not consider real-time evaluation of processing unit workload; computational capabilities are wasted when negative peaks occur, while overdimensioned capabilities (required to deal with positive peaks) are not exploited during the average behavior of the system.

In this paper, a general frame for optimum system dimensioning and dynamic reallocation of the computation is considered, so that a higher number of faults can be tolerated, possibly with degraded performances and functionalities, at low costs. Computation is modeled by using concurrent communicating processes in Section II, while the hardware structure based on multiprocessor distributed systems is analyzed in Section III. Stochastic evaluation of software performances is concerned with the capabilities of dealing with external events within a given maximum time. Hardware dimensioning and software allocation are considered contemporaneously to achieve a globally optimum solution. Fault-tolerance characteristics (error/fault detection, error masking, dynamic reallocation, graceful degradation) are integrated in the optimization model, as discussed in Section IV. The overall optimization procedure for static dimensioning and the dynamic reallocation techniques are finally discussed and evaluated in Section V.

II. THE SOFTWARE MODEL

The software application is modeled by using concurrent processes which communicate by message exchanging at mailboxes [8], [9]. This general model is particularly suited to describe many real-time applications, e.g., in manufacturing control and in real-time control systems. The sequence of operations of each process is cyclically performed, as it is typical of the above applications. Current literature points out three kinds of message passing: asynchronous, synchronous and buffered message passing store respectively infinite, zero, and a finite nonzero number of messages [9]. In order to simplify the evaluation of the characteristic figures for software and without loss of generality, in this paper we consider only asynchronous message passing ([4], [6], [7]). Under this assumption, send operations are nonblocking with respect to the calling process, while receive operation is blocking whenever no message is available in the mailbox. We assume also that process rescheduling onto the processing units is

Manuscript received May 18, 1993; revised October 3, 1993.

The author is with the Department of Electronics and Information, Politecnico di Milano, I-20133 Milano, Italy.

IEEE Log Number 9216002.

allowed only during receive operations, i.e., each sequence of instructions between two receive operations is executed as atomic entity. Starvation is avoided by introducing a receive operation from the operating system at the beginning of the process cycle. Dead-lock must be prevented by a correct design of the application.

Mapping of processes and mailboxes onto the hardware architecture defines the *allocation*. The *system configuration A* is the pair of the hardware architecture of the distributed computing system, and the allocation of processes and mailboxes on such architecture.

To design an optimum system, we must evaluate the efficiency and the effectiveness of computation partitioning into concurrent communicating processes with respect to the application itself and of the allocation of processes and mailboxes onto the architecture. In particular, we must analyze process performance and capabilities of matching the application requirements on response time to the external events. To such purpose, we introduce the process *cycle time* τ_i as the average time required to complete one iteration of the operation cycle of process P_i [4]. This figure depends on the allocation of processes and mailboxes since it is influenced by the time spent in receive operations and by the processor's workload. A detailed evaluation of the cycle time is discussed in [7], by using a graph model of the process operations and queueing theory.

Constraints may be imposed on the cycle time of each process to guarantee a fast response of the control system to the external events (e.g., user commands, sensor signals, error signals) and, thus, a correct and safe behavior of the whole system. This constraint is given by an upper bound for the cycle time: $\tau_i(A) \leq \tau_i^0(A)$. The software system behaves correctly if all constraints on the process cycle time are fulfilled.

Since static approximation [7] gives only the average value of the cycle time, the obtained system is not definitely guaranteed to meet very hard real-time constraints. Experiments in [7] have shown that our technique leads usually to satisfactory solutions; however, to deal more safely with very hard constraints, we can impose slightly harder constraints during the design phase so that the actual ones will be satisfied during the whole system life. In most of our experiments, the error in cycle time evaluation does not exceed 4% of the approximated value; the upper bound may therefore be reasonably decreased by such percentage. A different solution consists of relocating processes and mailboxes whenever the processor workload increases so much that proper cycle time cannot be provided; allocation is dynamically modified according to the actual workloads of all processors and to the available hardware resources required by each process.

Progressively relaxed constraints for the process P_i can be considered in the presence of an increasing number f of faulty processing units to model time degradation of the system behavior: $\tau_i(A) \leq \tau_i^f(A)$.

Overall evaluation of timing constraints on software components can be formally defined by introducing the *timing cost function* $C_T(A)$ for the given allocation. In a fault-free architecture, the timing cost function $c_T^0(A)$ is a two-valued function: it is null if all timing constraints are satisfied;

otherwise, it is infinite. Similarly, we define the timing cost function $c_T^f(A)$ in the presence of f faulty processing units, by considering the relaxed constraints and any distribution of the faulty units. In some applications, it is appreciated that the cycle time is minimum. This additional design goal is described by the minimum timing function $c_{T,M}^f(A) = \sum_i \tau_i(A)$, as a further term in the timing cost function in the presence of f faults. In general, the timing cost function $C_T(A)$ is given by

$$C_T(A) = \sum_{f=0}^N c_T^f(A) + \sum_{f=0}^N c_{T,M}^f(A)$$

where N is the number of processors in the architecture. The first summation may be truncated at the term related to the maximum number of faulty units that should be tolerated in the architecture, while the second summation should at most consider the first term ($f = 0$) since experiments have shown that the additional complexity of the optimization is not satisfactorily balanced by performance benefits in the application.

III. THE HARDWARE ARCHITECTURE

The architectures for real-time control applications may be modeled by using distributed systems composed of heterogeneous processing units connected by heterogeneous interconnection networks. Within this frame it is possible to consider a wide variety of computing structures, from the strictly coupled units of multiprocessor systems to the loosely coupled local area networks, and a wide organization of interconnections, from bussed systems and networks to point-to-point connections.

The distributed network may be decomposed in the following basic components: processors, memory storage, input/output devices, and interconnection links. By taking into account these components, the *total hardware cost* $C_H(A)$ of the architecture for a given allocation of processes and mailboxes is globally given by

$$C_H(A) = c_P(A) + c_S(A) + c_D(A) + c_L(A).$$

The *processor* includes not only the processing unit, but also all devices required to support its operation (e.g., cabinet, power supply, cooling system). For a given allocation of processes, the cost $c_P(A)$ of the processors is strongly nonlinear since it depends on the computational characteristics and power of the considered units [4]. An infinite value is assumed if no processor is available with the considered characteristics.

The *memory storage* may be permanent (e.g., hard disks, floppy disks, tapes, ROM), semipermanent (e.g., RAM with backup batteries), and volatile (RAM). It may be used for storing processes (executable code plus data area), communication mailboxes, and the operating system. The total cost $c_S(A)$ of the memory storage for the considered node is obtained by adding the contributions due to the three kinds of devices:

$$c_S(A) = c_{SP}(A) + c_{SS}(A) + c_{SV}(A).$$

Each of these components is a nonlinear function of the total amount of memory required by each specific use in the given

allocation:

$$c_{Sx}(A) = f_{Sx} \left(\sum_{P_j \in A} s_{P_j}(A) + \sum_{M_j \in A} s_{M_j}(A) + s_O(A) \right)$$

where x represents any of the previous kinds of memory (Permanent, Semipermanent, Volatile) and s is the space required by the corresponding entity (Process, Mailbox, Operating system) within a given kind of memory. They become infinite whenever technological constraints forbid further expansion.

Input/output devices are used to connect the control system to the controlled plant through sensors and actuators, and to the factory personnel through monitors, terminals, buttons, switches, and other command and presentation devices. The cost $c_D(A)$ of such components includes also the cost of sensors and actuators connected to the i/o control boards.

Each *communication* mechanism between processes is implemented by a suited hardware support. If a mailbox and all processes communicating through it are allocated on the same node, only memory storage is required [it is taken into account in $c_S(A)$]. If at least one process communicating with the considered mailbox is on a different processor, we must introduce interconnecting devices and cables. Let $c_L(A)$ be the total cost of the interconnection links over the memory storage; it is given by

$$c_L(A) = \sum_{\text{each pair}(P_i, M_k)} c_{Lk,i}$$

$c_{Lk,i}$ is null if the process P_i is on the same processor of the mailbox M_k ; otherwise, it is the cost of the interconnection. In the case several pairs of *process-mailbox* share the same hardware, the cost must be equally distributed over all the pairs.

IV. THE FAULT TOLERANCE ISSUES

Mapping of software components onto the fault-free architecture constitutes the *basic allocation* A_b . If such an allocation exists, it satisfies the basic constraints onto the cycle time, while the minimum hardware structure is derived during the optimization process together with the allocation itself. To deal with faulty components, we must add redundant hardware resources to the basic architecture. Some incompletely used components are intrinsically present in the basic architecture due to the quantized size of the components themselves (e.g., memories and processors); however, such redundancy is not sufficient to provide fault tolerance since it is a side-product of the optimization phase for the fault-free system, but it is not intentionally introduced in the architecture and strategically placed.

Position and quantity of the redundant items directly influence the system capabilities to detect faults, to correct errors, and to survive at nominal or degraded behavior. Redundancy definition must be performed after a detailed analysis of the application criticality and of the fault tolerance strategies. Efficient and effective solutions need high-redundancy and high-computing software supports, thus increasing the system

costs. An optimal design can be achieved by integrating the addition of redundant capabilities into the overall design process, instead of considering fault tolerance as an add-on feature that can be plugged onto the basic structure. In this section, modelling of fault tolerance within the global optimization frame is discussed.

The first step of any fault tolerance strategy is *fault and error detection*. Fault detection is directed to identify the presence of a faulty component: fault localization and diagnosis are adopted to identify the faulty component and to assert (if it is possible) the kind of fault. Error detection operates at a higher abstraction level, comparing the expected and the actual results of the system computation. Incorrect operation is possibly identified according with check granularity; if small-granularity mapping of behavioral errors onto physical faults has been provided by the system designer, the faulty component inducing the error can be localized precisely; otherwise diagnosis must be applied at a lower level to characterize the fault.

Many techniques have been proposed in literature for fault and error detection (for a basic introduction on these topics, see [10]). The main approaches are the following:

- 1) *modular redundancy*. Component duplication with output comparison can be used to identify the erroneous computation due to a single fault; an even higher number of identical components with output voting points out a higher number of faults.
- 2) *data coding*. Nominal data are replaced by applying the code transformation rules; modified computation is performed in the codeword space, and results are checked to verify if they belong to the output code. Specific codes have been developed to protect optimally different classes of hardware components (e.g., arithmetic codes, parity-based codes, Hamming codes).
- 3) *indirect checkers*. Additional components are added to check the behavior of the system indirectly. For example, watchdogs are used to verify that the computation is completed within a maximum time.
- 4) *signature analysis*. Compression of output results and comparison with reference values are used to verify the correct transformation during nominal computation.
- 5) *software replication*. Additional copies of the processes are allocated onto different processors and are executed in parallel; different outputs point out possible faults.
- 6) *periodic testing*. At the hardware level, the computation on nominal data is suspended, and testing is performed by using predefined data. At software level, we can add a process, in each processing unit, that tests all features of the unit concurrently with the other processes (i.e., the nominal computation is not suspended).

All the above approaches can be directly described in our system models. If detection is afforded at hardware level, hardware overhead can be included in the evaluation discussed in Section III. For example, if duplication is envisioned, the cost of each processing unit capable of detection is at least twice the cost of the nominal unit (additional costs for comparison circuits must be taken into account). In the case of

data coding, the cost of nominal units must be replaced by the cost of units operating in the coded space, including checker costs; memory storage must be increased to deal with larger data. In the case of software techniques, the cost increase of the redundant architecture is due to the higher number of software components and to redundant allocations; influence on timing function is due to additional operations for error checking and to the higher number of concurrent processes and mailboxes. If detection is performed by the operating system (managing output comparison and parallel execution of spare copies of processes), redundant resources are introduced even if they are not directly required by application processes; the operating system can in fact be viewed as a standard process of the application.

Error correction is more complex since redundant information must be able to support regeneration of the expected results. In the literature, extensions of the detection techniques have been considered to deal also with correction both at hardware and software levels (e.g., odd-modular redundancy with voting, enhanced coding techniques, odd copies of software components) [10]; all of them can be modeled in our optimization framework. If hardware approaches are envisioned, the cost of nominal components must be replaced by the cost of components capable of correction; voting circuits can be included in the proper system component. In the case of software approaches, additional hardware must be considered for the additional processes, and timing constraints must be modified to incorporate the additional operations.

Fault location can be performed by running the complete diagnosis procedure of the system or by exploiting redundant information generated during the computation. A number of approaches have been presented in literature [10]; the corresponding overhead in hardware and software components may be taken into account to describe their influence on system structure and performances. Besides, if off-line techniques are used, the main drawback is availability reduction of the system operation.

The ultimate strategy is directed to guarantee system *survival* after fault occurrence. To achieve this goal, faulty components must be excluded from the active operation, and unused spare resources must be exploited to supply the required computational capabilities. Dynamic redundancy for system survival is implemented by introducing spare interconnections and components at different levels: if enough redundancy is available, reconfiguration of interconnections allows the system to continue the nominal operations. Fault/error detection and, then, fault location are the preliminary steps which provide the necessary information for dynamic reconfiguration. Hardware overhead can be modeled by including the cost of spare components and reconfiguration structures into the hardware costs analyzed in Section III. Timing constraints are derived for the software allocation by taking into account the modified structure and performances due to dynamic redundancy. Verification of these constraints must be performed for all allowed allocations, since *a priori* the fault distribution is not known.

Application restarting after reconfiguration must be carefully considered to minimize the restart time and the loss of

data. Some approaches [4]–[7] propose static preallocation of one spare copy for each process in order to tolerate single faults (*redundant allocation*); one copy is active in the system computation, while the other one is standing by: the second copy will be activated as soon as the processor associated to the first one becomes faulty. The system behaves correctly if the repair time is short enough that a second fault does not occur before its completion. This static redundancy approach can be included in optimization by imposing that two copies of each process and mailbox are allocated onto different processors and that the corresponding time constraints are satisfied; as a consequence, interconnection links between processing units and input/output devices are replicated to guarantee the operation of each process. To deal with f faults, preallocation and dimensioning must consider f spare copies in the system, possibly leading to highly redundant

A different approach based on dynamic relocation of software components (processes and mailboxes) is here envisioned to overcome this drawback without greatly constraining both system survival and performances. Each time that a processing unit is declared faulty, the redundant processes and mailboxes (allocated on different processing units) are activated to allow the system survival, possibly with reduced timing performances. At the same time, to guarantee fast restart after a further fault, a new copy of these processes and mailboxes is created and allocated according to the actual processing workload and so that two copies of the same process are never in the same processing unit. Such novel redundant copies have the same role of the redundant copies preallocated in the redundant allocation for dealing with the first fault: they are quiescent until a fault destroys the active copies. Besides, dynamic reallocation will guarantee an optimal balancing of workload and, thus, of timing performances of all software components. After creation, suited operating system functions provide a realistic evaluation of the actual workload and of the actual cycle time of each process; from such information, possible transfer of processes (or mailboxes) from a processing unit to another is started by the operating system, according to the constraints on the specific hardware devices required by the processes themselves. Additional hardware must be introduced and considered during static dimensioning and optimization; in particular, as many specific devices (i/o devices and interconnection channels) as the tolerated faults must be added to the system, and as much memory storage must be introduced in the processing units as required to allow dynamic creation of redundant copies.

Minimization of restart time is also achieved by introducing checkpointing during computation. In this way, the current state of all spare copies of processes is updated to the current state of the running copies. Such operation is performed by a system process (the *checkpoint process*), which is always running in each processing unit. Each application process defines its check points by calling a *checkpoint procedure* of the operating system. Whenever a checkpoint is reached by an application process, the checking process transfers the value of the process variables to the spare copy. Data transfer from the source processing unit to the destination one is automatically managed by the checking processes

of such units. This technique guarantees a fast restart of processes which were running on the failed processing unit since operations do not need to be repeated from the beginning of the whole computation, but only from the last checkpoint; a careful choice of checkpoints may assure a very high coherence between actual and stored control system states.

All the above solutions try to guarantee the correctness of the system behavior at the nominal level. When redundancy for such level is exhausted, the system cannot operate normally in the presence of further faults. Often, reduced operation may be envisioned by considering partial execution of some nominal activities. The system behavior can be *gracefully functionally degraded* by allowing some operation to continue, some other operation to be partially performed, and some other activities to be stopped. Control and coordination of the progressively reduced system capabilities can be managed by hardware, or by operating system, or by application software; possible overhead (in hardware and time) for such activity can be included in the system cost. Graceful degradation must be designed carefully from the software point of view to avoid dead-lock and starvation.

V. DIMENSIONING AND ALLOCATION OPTIMIZATION: THEORETICAL FRAME AND EXPERIMENTAL RESULTS

On the hardware and software modeling introduced above, we can now define the optimization problem involving allocation of resources and processes in the distributed control system to satisfy computational and fault-tolerance requirements while minimizing the cost of the whole architecture.

This is a combinatorial optimization problem specified by a set of system configurations (i.e., the hardware architecture and the software allocation which are possible solutions of the problem) and by a cost function defined on all configurations. Increasing the number of processing units (i.e., enhancing the hardware architecture) and the software allocation can distribute processes and mailboxes to achieve higher computational performances. However, the costs of the system also increase.

Evaluations of the hardware costs and of the computational capabilities may be taken into account by defining the *system cost function* $C(A)$: in the configuration A , it is the figure of merit both of the allocation and of the hardware structure. Therefore, two components are considered in $C(A)$:

$$C(A) = c_H(A) + \gamma c_T(A)$$

where $c_H(A)$ and $c_T(A)$ are the costs of the hardware and the capability of satisfying the timing constraints in the allocation, respectively; γ is a scale factor to adjust the dimensions of the two terms. Evaluation and approximation of the hardware and timing costs have been already discussed in the previous sections.

This optimization problem is known to be an NP-complete problem [11]. Greedy algorithms could be adopted, but they usually achieve only a good solution (not the optimum one) since they do not consider all relationships among application requirements and available resources. In our experiments we used a classic branch-and-bound technique to explore the space of possible configurations. At each iteration, the optimization

algorithm analyzes the current configuration A and identifies possible enhancement both in terms of hardware reduction and of cycle time. For this new configuration A' , it has to evaluate the cost function $c(A')$: whenever the cost function assumes a better value, the new configuration is assumed as the reference one.

Convergence is assured by the characteristics of the algorithm itself, even if the computational effort and the storage requirements may be quite relevant. To prevent "explosion" of the number of examined configurations (i.e., of the dimensions of the branch-and-bound tree), particular care has been given to the evaluation of the timing constraints.

In our extensive experiments on the use of several fault-tolerant techniques to match the requirements of the issues discussed in the previous section, we defined characteristics and overheads of such techniques, and we evaluated their impact on the general optimization frame and on the resulting architectures. In this second part of the section, we present the main results of our experiments and the balanced design approach that we consider well suited for distributed control systems, even if it must be tailored onto the specific requirements of the application envisioned. Our goal is in fact to create such an architecture capable of high performances and fault tolerance at reasonable costs through statistical observations on the frequency of faults and on the use of hardware resources; besides, efficient support to system restarting after faults is considered with minimum loss of data. From our results, the designer can directly extract suggestions and solutions to design a fault-tolerant control system for a specific application: our experiments provide in fact the general guidelines to support the optimum choice with respect to the actual specific constraints and requirements.

To verify the effectiveness of the integrated approach to system design and to identify the impact of the fault-tolerance strategies and techniques on the hardware architecture and on the timing performances, we created a set of experiments derived from real-life cases. For each of them, we redesigned the distributed control system by applying the integrated approach presented above and alternative solutions to introduce different degrees of fault tolerance. We considered control systems for dyeing processes of yarns and fabrics, for pharmaceutical chemistry, for continuous distillation of aromatic essences, for robotized arms in a mechanical production pipeline, for mobile robots, and for electrical power plants.

Unless differently specified, we consider only the case of single errors since multiple errors usually require too much hardware overhead while their unmasked occurrence probability is often very low with respect to the single-error case. Cost functions of timing constraints and of hardware are thus evaluated as discussed in Sections II and III, respectively. To increase our confidence in the capability of satisfying the timing behavior not only as the average property, but in most of the cases, a detailed system simulation should be performed under realistic assumptions of the external environment; as discussed in Section II and in [7], we reduce the upper bounds by 4% since experiments have shown that actual cycle time often does not exceed the expected average by more than such a percentage.

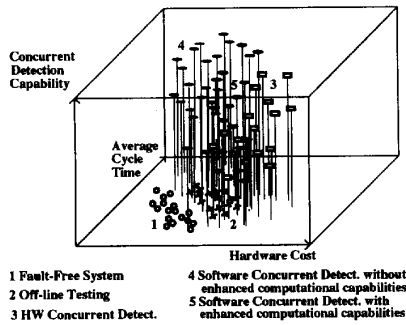


Fig. 1. Concurrent fault/error detection is given with respect to hardware cost and to average cycle time.

As error detection and testing are concerned, we experimented with several solutions among those discussed in Section IV. For off-line solutions we adopted standard testing techniques of the commercial components: testing is executed when the system is powered up and, possibly, as a consequence of a reset signal. For concurrent approaches, we considered the use of hardware fault detection by modular redundancy and by data coding.

At a higher level, concurrent detection was explored also by means of software testing: one additional *testing process* is allocated onto each processing unit to execute periodic verification of local hardware components and interconnection links. By message passing, testing processes exchange information about the overall hardware state and build the map of faulty processing units in each unit. Whenever several testing processes observe that a component is not working properly, they declare it as faulty and update the status map in each processing unit.

Some typical results are shown in Fig. 1; for some study cases we designed a possible solution based on the different techniques. Off-line testing requires low overhead but obviously does not provide any concurrent result validation; software concurrent testing induces an increase of cycle time if no additional computational capabilities are added to the system. Hardware detection needs often a high redundancy, but guarantees early detection and high error coverage.

As error correction is concerned, generally, we do not consider useful and cost-effective to adopt any hardware/software specific support. Only if the application is highly mission-critical a correction technique should be provided (e.g., triplication of hardware or software components with voting). In Fig. 2, the high costs of concurrent correction are shown both by using hardware modular redundancy, data coding, and software modular redundancy.

Careful analysis and experiments have been performed to study efficiency, effectiveness, and costs of several approaches for system survival. We considered the case of single and multiple faults: from statistical analysis, it is usually possible to derive that the probability of multiple faults is very low with respect to single faults. Only if the faulty component is not excluded from the active computation or if repairing time is too long, the multiple-error case could not be neglected since it may be generated by a sequence of single faults.

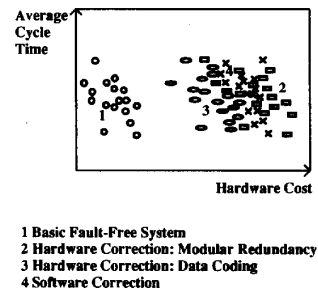


Fig. 2. Concurrent error correction is shown with respect to hardware cost and to average cycle time.

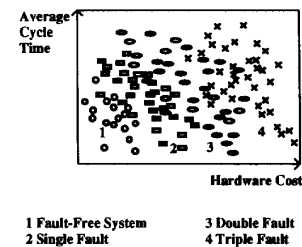


Fig. 3. System survival through static preallocation: increases of hardware cost and of average cycle time to provide system survival in fault-free, single-fault, double-fault, and triple-fault systems.

Survival after faults at the nominal behavior with low restarting time was experimented by overdimensioning the computing system and by static preallocation of spare (quiescent) copies of software components; timing performances have been reduced according to relaxed timing constraints to study the impact of fault tolerance on the architecture. Static optimization identified the optimal hardware architecture (including all redundant hardware components required to support execution of the spare copies after fault occurrences) and software allocations which provide the expected behavior in the absence of faults (basic allocation) and in the presence of faults (redundant allocations corresponding to the considered fault multiplicities and distributions).

In Fig. 3, we show some results of these experiments: hardware costs are given with respect to the average cycle time for fault-free systems, and in the presence of one, two and three faults. Degraded timing constraints are considered for each fault occurrence: timing constraints were increased by 5–15% for each fault. Fig. 4 presents the typical behavior of the relationship among hardware costs and timing constraints in the presence of faults for a study case: accepting time degradation of the system behavior (i.e., relaxing the timing constraints), it is possible to save a considerable hardware redundancy.

Due to the high cost of preallocation of spare software components in the case of multiple faults, we prefer to adopt two different approaches to deal with one fault or with several faults (both for contemporaneous and subsequent). Therefore, one basic copy and one spare copy of the processes are statically preallocated in the system, as discussed above. Subsequent faults will be possibly tolerated by relocating pro-

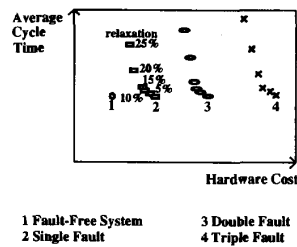


Fig. 4. System survival through static preallocation: increases of hardware cost and of average cycle time for different relaxations of timing constraints.

cesses and mailboxes dynamically, according to the constraints on hardware resources and relaxed timing and by taking into account the actual workload. Hardware requirements and timing impact of redundant dynamic allocations cannot be easily predicted in a static way: a complex simulation of all possible fault distributions and of all possible working conditions (workloads and external environment) should be performed to statistically evaluate them.

To avoid this drawback, we observed from experiments that, in the presence of several faults, it is more efficient to create dynamically the spare copies of processes and mailboxes and to allow dynamic reallocation of them according to the actual workload of each fault-free processing unit. Suited functions of the operating system provide a realistic evaluation of the actual workload and of the actual cycle time of each process; from such information, possible transfer of a process (or mailbox) from one processing unit to another is executed by the operating system, according to the constraints on the specific hardware devices required by the process itself.

To support this policy, additional hardware with respect to those required by single faults must be introduced and considered during static optimization. Dynamic creation of redundant copies of processes and mailboxes implies in fact availability of memory storage in the processing units, interconnection links, and i/o devices.

Since it is not feasible to evaluate *a priori* the amount of additional memory required, we can adopt an heuristic rule supported by experiments. We observed that a reasonable choice is based upon the characteristics of processes. We identify two classes of processes: the ones which are strongly related to specific devices must always be allocated onto processing units containing such devices, while the ones needing only memory and computational capability can be relocated without any loss of functionality.

For processing units containing the special devices, we add at least an amount of memory capable of containing all the processes related to such devices. Satisfaction of timing constraints is more difficult to evaluate in a stochastic way. Experiments have shown that it is effective to increase the computational capability of the processing unit by a percentage equal to the difference between the percentage of memory added to the system (i.e., the operation that must be performed) and the percentage of relaxation of timing constraints for the considered number of faults with respect to the basic allocation.

As the second class of processes is envisioned, we compute the total additional memory required by one copy of all

these processes; experiments have shown that it is effective to distribute such additional memory over the possible fault-free processing units. To compute the amount of memory for each unit, we divide the total additional memory by the minimum number of fault-free units (i.e., the total number of processing units minus the maximum number f of tolerated faults). Anyway, at least we should add as much memory as the average required by the processes of such class. Computational capability is increased as in the other class of processes.

Whenever the cost of processing units becomes infinite or, possibly, too large, it is reasonable to introduce one (or several) additional smaller processing unit; in this case, often, the costs become feasible and acceptable.

Whenever several units agree that another unit is faulty, the *reconfiguration process* of each fault-free processing unit starts the sequence of operations allowing excluding the faulty unit from the computation and redistributing the computation. The reconfiguration process of a processing unit is locally responsible to manage the dynamic allocation of processes and mailboxes.

The spare copies of the application processes running onto the faulty unit are activated by means of message exchanging among the reconfiguration processes: interconnection mailboxes are properly rerouted. Then, the reconfiguration processes spawn a new spare copy of each of these software components, and transfer it into a suited processing unit by interacting with the reconfiguration process of the destination unit. The list of allowable destination units for each application process is initially defined according to the hardware resources required by process itself; it is dynamically updated by the reconfiguration process by deleting the faulty units and by inserting the repaired units. The destination unit is chosen according to the actual and expected values of resource availability, cycle time and workloads. Also these figures of merit are locally evaluated by the reconfiguration processes on the values observed by the scheduling processes of the operating system; reconfiguration processes dynamically distribute them to all fault-free units.

A number of experiments have been performed to observe the influence of the different approach and of the characteristic parameters envisioned above, in the case of dynamic reallocation with static dimensioning for the single-fault case. Some results for selected case studies are presented in Fig. 5 to point out the dependencies among hardware costs and cycle time for different numbers of tolerated faults. In Fig. 6, a typical relationship between hardware cost and cycle time is given for different relaxations of timing constraints and different percentages of hardware redundancy introduced to tolerate the faults occurring after the first one. We can observe that hardware costs are usually smaller than the corresponding cases in the completely static preallocation.

Whenever the workload in a processing unit becomes too high (a reasonable limit is between 95% and 98% of the nominal computational capabilities), dynamic reallocation can be exploited to balance the workload all over the distributed control system: this prevents the timing constraints that are not satisfied due to positive workload peaks in such a unit. In this case, the reconfiguration process identifies and moves

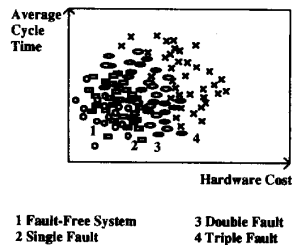


Fig. 5. System survival through dynamic allocation: increases of hardware cost and of average cycle time to provide system survival in fault-free, single-fault, double-fault, and triple-fault systems.

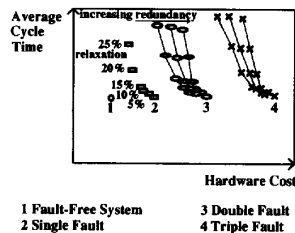


Fig. 6. System survival through dynamic allocation: increases of hardware cost and of average cycle time for different relaxations of timing constraints and for different percentages of hardware redundancy.

one software component onto another processing unit. In our experiments, the following selection criterion has been shown effective and with a low computational overhead: we choose the process with the highest difference between the expected cycle time and the actual value, and with at least one processing unit which is allowed for reallocation. If no new allocation can be found by applying the rules discussed for fault tolerance, another process is examined. After the system has reached a new stable or quasi-stable behavior (typically, after 5 complete cycles of all highly interactive processes), the possibility of dynamic reallocation for workload balancing is considered again.

VI. CONCLUDING REMARKS

In this paper, we have presented an integrated approach to dimensioning the distributed system and to allocating the computation for real-time control applications. We reversed the classical approach of software allocation on a fixed architecture: we allocate the software components onto a virtual architecture having no predefined constraints. The actual hardware structure is assembled following guidelines of optimum software allocation.

Fault tolerance has been included in the design methodology as a basic feature for any critical system. Different strategies have been analyzed: their integration in the homogeneous frame for system design has been introduced. The static optimization problem for single faults has been defined; the extensions of the distributed architecture to support the dynamic reallocation of software components have been analyzed in order to tolerate subsequent faults.

Suggestions and solutions to design a fault-tolerant control system for a specific application can be directly derived from

the approach and the results presented in this paper. Our experiments provide in fact the general guidelines to support the optimum choice of the architectural solutions and of the approach to fault tolerance with respect to the actual specific constraints and requirements. The use of our methodology has been experimented also in some real-life cases concerning chemical industries, robotics, and electrical power plants; in such cases, it has been proved effective in reducing the costs (about 20% less than traditional methodologies) while guaranteeing a high degree of fault tolerance.

Further research should improve the efficiency of the static optimization algorithm and of the components in the operating system for dynamic reallocation. Management of partially faulty processing units could be introduced in the dynamic reallocation strategy to deal with faults that affect specific components without inducing a complete loss of functionalities of the whole unit.

REFERENCES

- [1] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *IEEE Computer*, July 1978.
- [2] O. I. El-Dessoukian and W. H. Huan, "Distributed enumeration on network computers," *IEEE Trans. Comput.*, vol. C-29, Sept. 1980.
- [3] P. R. Ma, "A model to solve time-critical application problems in distributed computer systems," *IEEE Computer*, Jan. 1984.
- [4] F. Distanto and V. Piuri, "Distributed architecture design to match optimum process allocation: A simulated annealing approach," in *IEEE Proc. Real-Time Syst. Symp.*, San Jose, CA, Dec. 1987.
- [5] V. Piuri and E. Tourouta, "About fault-tolerant allocation of tasks in multiprocessor architectures and system dimensioning," *Microprocessing and Microprogramming—Euromicro J.*, vol. 19, 1987.
- [6] [—]{—}, "An approach to fault-tolerant allocation of concurrent communicating processes in multiprocessor architectures and hardware dimensioning," in *Proc. Euromicro 87*, Portsmouth, U.K., Sept. 1987.
- [7] F. Distanto and V. Piuri, "Hill-climbing heuristics for optimal hardware dimensioning and software allocation in fault-tolerant distributed systems," *IEEE Trans. Reliability*, vol. 38, Apr. 1989.
- [8] P. Brich Hansen, "Concurrent programming concepts," *ACM Comput. Survey*, Dec. 1973.
- [9] G. R. Andrews, F. B. Schneider, "Concepts and notations for concurrent programming," *ACM Comput. Survey*, Mar. 1983.
- [10] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*. Bedford, MA: Digital, 1982.
- [11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.



Vincenzo Piuri (S'84-M'86) received the Dr. Ing. degree in electronic engineering in 1984 and the Ph.D. degree in computer science in 1989, from Politecnico di Milano.

He was Assistant Professor in Electrical Engineering and in Computer Programming at Politecnico di Milano; he is presently Associate Professor in Operating Systems at Politecnico di Milano. His research interests include distributed and parallel computing systems, computer arithmetic, neural networks, fault tolerance, CAD for VLSI. Original results were obtained in architectural design of advanced fault-tolerant computing systems for signal processing and measurements. He cooperates in the organization of several national and international research projects, sponsored by the EEC, by the National Research Council, by the Italian Ministry of the University and of the Scientific and Technological Research, and by industries.

Dr. Piuri is a member of AEI and Euromicro; he is presently Secretary of IEEE Computer Society—North Italy Chapter, IEEE Instrumentation and Measurement Society—North Italy Chapter, and AEI—Milan Chapter.