

Hill-Climbing Heuristics for Optimal Hardware Dimensioning and Software Allocation in Fault-Tolerant Distributed Systems

Fausto Distante

Politecnico di Milano, Milan

Vincenzo Piuri

Politecnico di Milano, Milan

Key Words — Hardware dimensioning, Process allocation, Real-time systems, Cycle time, Queueing theory, Simulated annealing.

Reader Aids —

Purpose: Demonstrate optimal design

Special math needed for explanations: Queueing theory, Combinatorial optimization

Special math needed to use results: None

Results useful to: Network & computer designers

Abstract — An optimum distributed architecture with fault-tolerance capabilities for a given software application may be obtained by allowing allocation algorithm to evolve without any *existing-hardware* constraint. Distributed software partitioning and allocation is done using the Simulated Annealing optimization algorithm. To define the cost function used by the optimization algorithm, a model for interacting processes constituting the software application is presented. Tuning of algorithm parameters has been considered to assure convergence at a reasonable cost in term of computation time.

1. INTRODUCTION

Distributed systems are by now the best accepted solution for throughput enhancement of large software applications composed of highly concurrent processes. Intrinsic computation parallelism may be in fact exploited by parallel processing capabilities of such architectures. Two problems have to be solved in designing a multiprocessor system:

- hardware dimensioning,
- software partitioning and allocation onto such architecture.

Solutions of these problems are strictly related: in fact a high number of processing units allows to exploit 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. An integrated approach

should consider both aspects at the same time (during design phase) to achieve the optimum solution. A third problem has also to be faced whenever such systems perform critical control activities: in these cases it is almost mandatory for the system to be able to survive (possibly with degraded performances) after a processing unit has been declared faulty.

Several approaches to solve software partitioning and allocation upon distributed computing systems have been presented in the literature, mainly based upon graph theory, integer programming and heuristic methods. In graph theoretical approaches, tasks and communications are mapped onto a graph representing the application. To minimize total cost a min-cut algorithm was proposed in [1]; this approach becomes very complex and time-consuming as the number of processors increases. Integer programming methods [3], [4] lead to a flexible technique, giving the opportunity to introduce in the model some environmental constraints such as processes' priority. This approach is though limited by the amount of storage and computation time required. Heuristic methods, such as branch-and-bound techniques, are proposed in [5]. Task preference and task exclusive matrices make application constraints fairly visible, but they have to be accurately designed to limit the number of iterations of the algorithm.

Three common characteristics may be identified in all above techniques:

- software partitioning and allocation is optimized with respect to an already defined distributed hardware architecture,
- computation is described by means of a too simple model for some applications as manufacturing control or real-time control systems (often, software is modelled by means of tasks whose interactions may be represented by an acyclic graph and are executed only once),
- fault tolerance is usually not considered.

In the approach presented here, architecture design, software allocation and fault tolerance strategies are dealt with at the same time. The adopted fault-tolerance strategies do not affect the overall philosophy of our approach: they may be considered as an *add-on* which will be included when required. The only restriction we impose for adoption of our fault-tolerance strategies is that no more than one processing unit is faulty at the time and that mean time between failures (MTBF) is high enough to allow maintenance operations (which ultimately could consist in substitution of the offending processing unit) after a fault has occurred.

Given a set of possibly communicating tasks (from now on called *software application*) we will let the allocation algorithm evolve without considering any hardware architectural constraint: that is, we will not try to optimize distribution of a software application upon an existing distributed architecture but we will optimally partition the software application and then assemble the hardware network in order to satisfy allocation requirements. In other words, we will use results of a *non-architecture-constraint* allocation as guidelines for architecture design.

In section 2 we will introduce the models for computation and hardware architecture as well as fault-tolerance strategies. Computation is modelled by means of concurrent processes communicating through mailboxes. Activities of each process are performed cyclically. The distributed architecture is described by a network of processing units connected by communication links (e.g. double-port memory or serial busses).

The algorithm adopted to perform both hardware dimensioning and software allocation is the *simulated annealing*, a well known technique in the VLSI area (mainly in routing and placement problems). Simulated annealing was introduced by Kirkpatrick [6] as a randomized algorithm capable of identifying globally optimum least-cost configurations of a large NP-complete problem whose cost function has local minima [7], [8].

An overview of simulated annealing will be presented in section 3, while in section 4 we will discuss and evaluate the cost function adopted. The cost function has to take into account two major parameters, namely *dispersion costs* and *aggregation costs*. The former are concerned with the trend of allocating processes as sparsely as possible (i.e. one process for each processor). While optimizing workload distribution, dispersion overloads inter processor communication links and increases hardware costs to an unrealistic extent. Conversely, the aggregation costs affect clustering of processes upon processors: inter-processor communications are minimized, but unbalanced workload distribution (or, possibly, processor overload) may result. A formal model of computation based upon queuing theory will be introduced to evaluate aggregation costs.

Finally, we will present some implementation details and experimental results in section 5. Performance of the algorithm is discussed as approximate evaluation of scheduling parameters and rate of convergence are concerned. Alternative implementations of the annealing technique are also considered and compared.

2. A MODEL FOR A DISTRIBUTED SW APPLICATION

The computation performed by processing units is modelled by a set of *concurrent communicating processes*: a communicating process is a sequential computation which may be executed in parallel with other processes and which is able to exchange messages with them [9] [10].

To model the interactions among processes we adopt a message passing philosophy: interacting processes exchange information by sending and receiving messages through mailboxes, i.e. communication points [10]. Current literature points out three kinds of message passing techniques classified by the number of messages that can be stored by the mailboxes: asynchronous, synchronous and buffered message passing store respectively infinite, zero and a finite non-zero number of messages [10].

In this paper we consider asynchronous message passing: a typical characteristic of this approach is that send operations are *non-blocking* with respect to the calling process since spare storage for new incoming messages is always available in each mailbox; conversely, a receive operation is blocking if the queue of available messages is empty.

To simplify the analysis of the characteristics of the model we introduce two additional assumptions:

- only one process can receive messages from a mailbox, while more than one process is allowed to send messages to it;
- a FIFO policy for message enqueueing is assumed.

Results derived here may be easily extended to cover also other kinds of communication protocols (for example: synchronous or buffered message passing, and conditional send/receive operations, complex queuing policies, and multiple communications).

Concurrent communicating processes may be viewed as a cyclic repetition of sequences of operations. This holds both from a general point of view and in the case of particular application fields (see, e.g., manufacturing or process control). Presence of send and receive operations in each sequence of instructions allows communication with other processes. For simplicity sake, we also assume that rescheduling of processes may occur only during receive operations, i.e. each sequence of instructions between two receive operations is executed as atomic entity. To avoid starvation a *receive* from the operating system is always introduced at the beginning of the process cycle; dead-lock is prevented by the correct design of the whole software application.

To evaluate allocation optimality, the following figures of merit are defined:

- the *cycle time* of each process
- the hardware cost.

The cycle time τ_i of process P_i is the average time required to execute a single iteration. This figure depends on allocation since it is influenced by the time spent in receive operations and by the processor's workload. A constraint may be imposed on the cycle time of each process: in fact, a particular process may need to execute the complete cycle in a time no greater than a given bound: $\tau_i \leq \tau_i^*$. In other words, the *response* time of the process might be limited by an upper bound.

Different approaches have been presented to introduce fault tolerance in a distributed system, both via hardware and software techniques involving physical and computational redundancy. In this paper we do not deal with special-purpose hardware and/or software supports which allow the system to survive hardware failures: we consider the problem of achieving a highly available system by means of fast restart of computation after a fault has occurred. Our approach is quite simple and cost-effective: it consists in *pre-allocating* an additional copy of each process in the system. Whenever a processing unit is declared faulty, copies of processes originally running on such unit (and pre-allocated upon the remaining units of the system) will be automatically activated. This allows exclusion of the faulty unit for maintenance operation, while the system (though with degraded performances) is still working. Such approach considers only hardware failures and is not concerned with software fault-tolerance.

The set of the first copy of each process (called *basic process*) constitutes what we call the *basic allocation*: in fact these processes are activated at the nominal start-up and perform the computation satisfying the basic constraints $\tau_i \leq \tau_i^*$ on cycle time. The additional copies (called *redundant processes*, thus constituting the *redundant allocation*) are activated only when a processing unit is declared faulty. In this way it is possible to minimize the failure time of the whole system because the remaining units can continue their computation while processes once running on the now faulty unit are replaced by the redundant copies on other units. System performance is obviously degraded after the failure since processes run on a reduced number of processing units: to allow system survival it is in fact necessary to relax the constraints on the cycle time of active processes.

The approach presented here overcomes the failure of a single processing unit: the system may not survive any more to subsequent faults since redundant copies of the processes running on new faulty processors may not be available.

When continuous operation, even with reduced performance, is a mandatory requirement, it is possible to extend our approach to design a gracefully degradable distributed system. In such a case, we pre-allocate as many redundant copies as the maximum number of faults the system must tolerate without crashing. Obviously, as redundancy increases, so does the cost of the overall system. Experiments have shown that no more than three or four faults should be considered with this technique. If a higher degree of fault tolerance is required, better results are achieved by introducing other hardware redundancies, e.g. in the processors themselves.

3. HILL-CLIMBING HEURISTIC FOR RESOURCES ALLOCATION

Allocation of resources in a multiprocessor system to satisfy computation and fault-tolerance requirements while

minimizing a cost function of the whole computing system is known to be a NP-complete problem [11], [12].

To simplify the allocation process, greedy algorithms may be used [13]; however, they usually achieve only a good solution, often quite different from the optimum allocation: in fact local optimization does not take into account all relationships among application requirements and network availabilities.

In this paper an alternative approach to the allocation problem is proposed by adopting the simulated annealing technique [14], a heuristic method that leads to the optimum solution when algorithm parameters have been properly tuned. The main characteristic of such algorithm is the capability of performing *hill climbing* moves, i.e. at each step a new configuration with increased cost may be accepted. In this way, when a suitable *tuning* of some parameters is provided, it has been proved [14] that it is possible to reach the global minimum even when other local minima are present.

The combinatorial optimization problem is specified by a finite set of configurations (states), i.e. the possible solutions of the problem, and by a cost function defined on all the states. In our problem each state S is a mapping of basic and redundant processes and related mailboxes upon processing units in the distributed computing system.

For each of these configurations, the cost function $c(S)$ is the figure of merit of the allocation performed. Two different components are accounted for in such function to consider both hardware cost and timing constraint

$$c(S) = c_H(S) + \gamma c_T(S)$$

The first term $c_H(S)$ considers the cost of the hardware required by the allocation, i.e. the costs of processors, storage and communication channels. The second component $c_T(S)$ considers the capability of the computing system to satisfy performance requirements both in the fault-free and faulty case. γ is a scale factor to adjust dimensions of these two terms. Details on the evaluation of these components will be given in the next section.

At each iteration of the algorithm, a new solution S' is randomly selected among the neighbors of the last accepted solution S : two solutions S and S' are *neighbors* if they differ only for the allocation of a single process and related mailboxes. The new allocation is accepted not only if the cost function has decreased with respect to the previous one, but also if it has had an increase limited in a given range. Hill climbing is performed according with a random acceptance rule: it is controlled by the current value of a parameter T (which eventually fixes the acceptable range of increase), which behaves as the temperature of the annealing process. The new allocation, even if worse than the last accepted one, will be accepted if the function

$$f(S, S', T) = \min(1, \exp\left(-\frac{c(S') - c(S)}{T}\right))$$

is greater than a random number with uniform probability distribution in the range $[0, 1]$. As T decreases, the probability of accepting hill climbing moves decreases since function f will have higher probability of being smaller than the generated random number.

Simulated annealing may be modelled by means of Markov chains: on this model some properties of the algorithm may be derived. In particular, it is possible to prove that the algorithm surely generates the global optimum, whenever given conditions on the number of iterations at each value of the parameter T and on the updating rule for T are satisfied.

To assure the convergence of the algorithm, T must be updated accordingly with the following rule [14]:

$$T = \frac{rL}{\ln(n + n_0 + 1)}$$

where r is the radius of the graph underlying the Markov chain, L is the Lipschitz constant of the cost function, n is the current number of iteration and n_0 is a parameter not less than 1. Under this assumption of Markov chain is strongly ergodic, thus implying that any state probability vector converges component-wise to the constant vector of the optimum solution.

The basic structure for an hardware dimensioning and software allocation system using the simulated annealing algorithm is presented in figure 1.

```

Simulated Annealing Allocator ()
{
  S = Initial Allocation();
  T = Initial Temperature();
  while(stopping criterion is not satisfied)
  {
    while (inner loop criterion is not satisfied)
    {
      j = New Allocation Generator(S);
      if (New Alloc. Accept Test(c(j), c(S), T))
        S = j;
    }
    T = Temperature Update(T);
  }
}

```

Fig. 1 Simulated Annealing Allocator

4. COMPUTATION MODELING AND COST FUNCTION EVALUATION

Graph modeling may be used to represent the process activities and to evaluate timing performances (ie the cycle time). The computation graph of a process consists of a set of nodes and labelled oriented arcs. Three types of nodes may be identified:

- the initial node, ie the node from which the computation of the process begins;
- the computation nodes, which represent the beginning and the end of each operation belonging to the process;
- the end nodes at which the computation of the process ends.

As far as arcs are concerned, we may subdivide them into two classes:

- communication arcs, representing communication primitives among processes;
- computation arcs, representing any other process operation.

Every arc is labelled by:

- the executive time t_I of corresponding operation I ;
- the selection probability p , ie the probability that the arc is selected when execution reaches the node from which the arc starts.

The computation graph of a process may be built from its source code by associating arcs to the instructions of the process and then properly connecting such arcs to represent the execution flow of the process. To label such arcs, average execution time of both computation and communication operations should be evaluated. Execution time of any computation and *send* operation can be deterministically evaluated. Things are not so straightforward when evaluation time of *receive* operations is afforded, since it depends also on the rate of availability of messages, i.e. on the characteristics of the whole computation itself. The ideal execution time t_R of a receive operation may be computed by assuming that a receive never stops the calling process, i.e. constant availability of incoming messages; the actual execution time is the sum of the ideal time t_R and the waiting time t_W during which the process is waiting for a message. Arcs representing receive operations are labelled with the ideal execution time.

The second figure in the label relates to selection probability. It depends on the topology of the computation graph and on input data: in a sequence of operations without branches, selection probability of each operation (possibly except the first one) is always equal to 1 since no alternative may be chosen; conversely, in a *branch on condition* statement each branch has a different selection probability depending on the actual data. An example is shown in figure 2.

An exact computation or experimental measure of cycle time cannot be performed in static conditions, i.e. without executing the processes in the system. Cycle time may be obtained from simulation of execution of processes, using the process computation graphs and assuming a realistic distribution of input data: simulation should be performed so that the main cycle in each process is repeated enough times to obtain meaningful statistics. The main drawback of such approach is the great computational effort required to evaluate the cost of each allocation: complete simulation of processes must be executed whenever the allocation changes since the cycle time depend in this case on

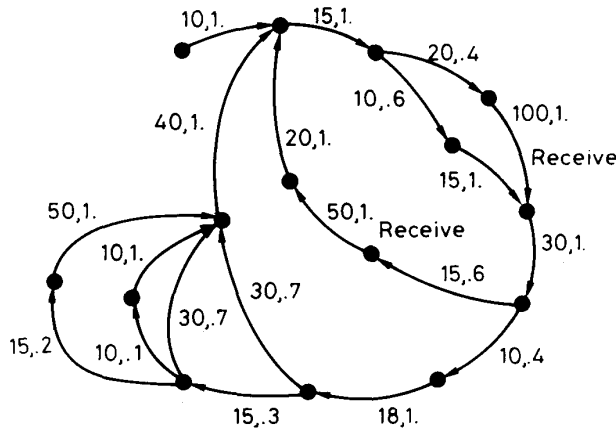


Fig. 2 Process Computation Graph

the execution time of receive operation and, ultimately, on the allocation itself.

A better solution consists in introducing an approximation which allows a statistical evaluation of the cycle time, at least as first approximation: our extensive experiments have shown that good results may be achieved by assuming the waiting time of each receive operation executed by process P_i on a given mailbox equal to the average cycle time of processes P_j performing send operations on the same mailbox. To avoid recursion (since again cycle time of processes P_j should be computed by taking into account the waiting time of their receive operations), we assume the waiting time for receive operations of processes P_j equal to the ideal receive time: in other words, we suppose that a message is always available for all these receive operations. Obviously, this technique gives only a lower bound of the cycle time; nevertheless, experiments have outlined that the difference of the exactly simulated values τ_{eff} and the estimated ones τ_{apx} is lower than 3% in most cases and not greater than 7% (some important examples are shown in table 1).

TABLE 1

Process	Example					
	#1		#2		#3	
	τ_{apx}	τ_{eff}	τ_{apx}	τ_{eff}	τ_{apx}	τ_{eff}
P1	30.5	31.6	8.3	8.4	15.2	15.6
P2	12.9	13.3	27.5	28.1	28.3	29.1
P3	17.3	17.4	89.2	90.1	8.1	18.8
P4	25.5	25.9	34.1	34.8	36.5	37.9
P5	37.9	38.2	22.1	22.7	49.7	51.1
P6	29.2	30.1	91.3	93.1	22.3	22.7
P7	87.7	88.3	67.4	68.6	37.5	38.9
P8			62.1	63.5	18.6	19.8
P9			12.3	12.7	7.1	7.4
P10			14.5	15.1	11.3	11.5
P11			45.7	47.1	26.9	27.7
P12			51.6	53.2		
P13			78.3	79.2		
P14			10.1	10.3		
P15			7.5	8.1		

Given the above model, we will now introduce a technique to estimate the cycle time of a basic or redundant process and, thus, to evaluate the timing performance of the considered allocation. Let's recall some assumptions required to simplify the evaluation, even if not affecting the overall method and final results.

First, we suppose that processes may be rescheduled only during receive operations; second, to avoid starvation, it is assumed that the process receives a message from the operating system at the beginning of the cycle of the process itself (thus *forcing* a compulsory rescheduling point in each process).

Under these assumptions, the process may be viewed as a cycle of non-interruptable sequences of instructions each starting after a receive operation and ending before another receive: in the sequel we will refer to such sequences as *blocks*. Each block is waiting for a message to complete the receive operation at its beginning; as soon as a message is available, the block becomes ready to run on the processing unit upon which the process is allocated.

In the subgraph representing block $B_{i,k}$ of process P_i , we can find h different paths $R_{i,k,h}$ (due to h conditional statements): we call *route* each of these paths.

Route probability $p_{R_{i,k,h}}$ is the probability of executing route $R_{i,k,h}$ among the different routes in block $B_{i,k}$: it is given by the product of all selection probabilities of the arcs belonging to the route and following the arc R related to the receive operation at the beginning of the route itself.

$$p_{R_{i,k,h}} = \prod_{arc_m \in (R_{i,k,h} - \{R\})} p_m$$

Ideal route execution time $t_{R_{i,k,h}}$ of route $R_{i,k,h}$ is the minimum time required to execute all instructions of that route; it may be obtained by considering the execution time of each instructions I_j belonging to such route and by neglecting the waiting time of the receive operation:

$$t_{R_{i,k,h}} = t_R + \sum_{I_j \in (R_{i,k,h} - \{R\})} t_{I_j}$$

The *ideal block execution time* $t_{B_{i,k}}$ of each block $B_{i,k}$ of process P_i is the minimum time required to execute the block itself; it may be computed as:

$$t_{B_{i,k}} = \sum_{R_{i,k,h} \in B_{i,k}} p_{R_{i,k,h}} t_{R_{i,k,h}}$$

For each block $B_{i,k}$ we can also compute the block probability $p_{B_{i,k}}$: it is the probability that block $B_{i,k}$, among all blocks of process P_i , enters the *ready-to-run* state. To such purpose we identify each *mesh* $M_{i,n}$ in the computation graph of the process P_i , i.e. each closed path whose arcs are executed at most once each cycle. Among all possible meshes of process P_i , a mesh $M_{i,n}$ may be executed with probability $p_{M_{i,n}}$, called *mesh probability*, given by the product of all choices performed along the mesh:

$$p_{M_{i,n}} = \prod_{arc_m \in M_{i,n}} p_m$$

For each mesh $M_{i,n}$ the *ideal mesh execution time* $t_{M_{i,n}}$ is defined by neglecting waiting time of receive operations: the average of the actual executing time of the mesh cannot be smaller than this ideal value.

$$t_{M_{i,n}} = \sum_{B_{i,k} \in M_{i,n}} t_{B_{i,k}}$$

Considering mesh probability, block probability $p_{B_{i,k}}$ may be computed as follows by recalling that receive operations delimit blocks. We identify the meshes in the computation graph of the process and we evaluate the mesh probability for each of them. Then, we identify the receive operations in each mesh $M_{i,n}$. The probability of executing block $B_{i,k}$ among the blocks belonging to mesh $M_{i,n}$ is given by $1/N_{i,n}$, where $N_{i,n}$ is the number of receive operations in the mesh, which is equal to the number of blocks in such mesh. Therefore, the probability of executing block $B_{i,k}$ of process P_i may be estimated by considering all the meshes containing the block itself, that is:

$$p_{B_{i,k}} = \sum_{M_{i,n} \ni B_{i,k}} \frac{p_{M_{i,n}}}{N_{i,n}}$$

From various figures previously obtained, it is possible to study the probability distribution of ideal block execution time for each process P_i . In particular, we obtain that average value t_{B_i} of ideal block execution time is:

$$t_{B_i} = \sum_{B_{i,k} \in P_i} t_{B_{i,k}} p_{B_{i,k}}$$

while its variance $s_{B_i}^2$ is:

$$s_{B_i}^2 = \sum_{B_{i,k} \in P_i} (t_{B_{i,k}} - t_{B_i})^2 p_{B_{i,k}} = \sum_{B_{i,k} \in P_i} t_{B_{i,k}}^2 p_{B_{i,k}} - t_{B_i}^2$$

Last figure we need to evaluate process cycle time is the waiting time of receive operations performed by the process itself: this figure may be evaluated by using a proper model of the activities in the processing units based upon the queuing theory [17] [2]. Adopting such theory we can think of the processing unit as a server, while blocks $B_{i,k}$ of each process P_i are the transactions that have to be served: if the unit is busy executing a block, the other blocks are enqueued into the waiting queue of the server. Service time t_{s_i} is the time required to execute the whole block $B_{i,k}$ and may be computed as shown above.

The frequency of transaction arrivals is λ , while the interarrival time (ie the time between the arrival of two subsequent transactions into the queue) is $t_{\alpha} = 1/\lambda$. In our model, the interarrival time t_{α_i} (referred to transactions belonging to process P_i) is the mean time between two subsequent send operations performed by any process $P_j \neq P_i$ on the mailboxes from which P_i receives messages: let S_i be the set of such processes P_j . We can estimate in first approximation the interarrival time t_{α_i} by means of the following formula:

$$t_{\alpha_i} = \frac{\sum_{P_j \in S_i} T_{I_j}}{\sum_{P_j \in S_i \wedge \text{Send}_{j,k} \in P_i} p_{\text{Send}_{j,k}}}$$

where

- T_{I_j} is the *instruction cycle time* of the process P_j , i.e. the time required to perform all operations by neglecting the waiting time of receive operations:

$$T_{I_j} = \sum_{M_{j,n} \in P_j} t_{M_{j,n}} p_{M_{j,n}}$$

- $p_{\text{Send}_{j,k}}$ (probability of executing the send operation k of process P_j) may be evaluated in the same way as the block probability.

Queuing theory may again be used to evaluate some parameters of the service supplied by the processing units of this ideal network in which receive operations have null waiting time. The *partial use of the server* ρ_i of the process P_i is the ratio between the time t_{B_i} in which the server executes a block of P_i and the interarrival time t_{α_i} of its blocks:

$$\rho_i = t_{B_i}/t_{\alpha_i} = \lambda_i t_{B_i}$$

Such figure is the average fraction of time unit in which the processing unit executes process P_i . The *use of the server* ρ can thus be defined as the sum of the partial use of the server of each process allocated on the considered processing unit.

$$\rho = \sum_i \rho_i = \sum_i \lambda_i t_{B_i}$$

It is the average fraction of time unit in which the processing unit is working.

To simplify evaluation of waiting time, we assume as first approximation that the probability distribution of interarrival and service time is exponential: moreover, we consider FIFO policy for mailboxes and for queues of processes ready to run; population of incoming transactions (blocks) is assumed infinite.

Under these assumptions, probability q_i that one of blocks $B_{i,k}$ of process P_i is executed by the processing unit on which P_i is allocated is:

$$q_i = \lambda_i \left(\sum_{P_j \in A} \lambda_j \right)^{-1}$$

where A is the set of processes allocated on the considered processing unit. The *mean service time* t_s of the processing unit and its variance are thus given by the following formulas:

$$t_s = \sum_{P_i \in A} q_i t_{B_i}$$

$$s_s^2 = \sum_{P_i \in A} q_i (t_{B_i} - t_s)^2 = \sum_{P_i \in A} q_i t_{B_i}^2 - t_s^2$$

The *total transit time* τ_R of a transition in the system is the mean time that a block spends in the waiting queue plus the time required by its execution. Fig. 3 shows the ratio between total transit time τ_R and service time t_s

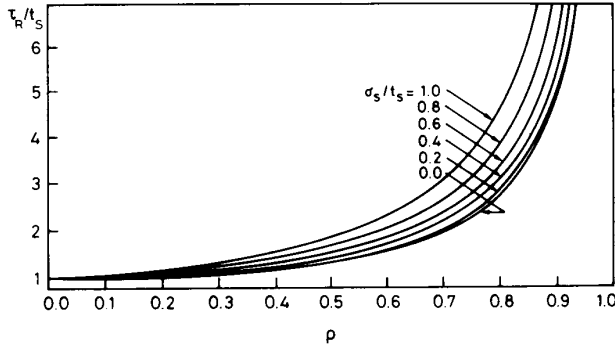


Fig. 3 Total Transit Time

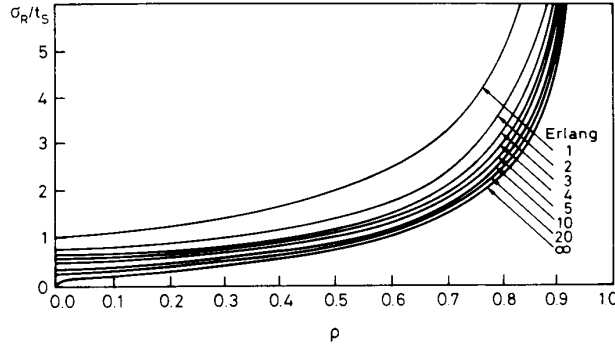


Fig. 4 Variance of Total Transit Time

versus the use of the server ρ , while fig. 4 shows the ratio between the standard deviation σ_R of the total transit time and the service time t_S versus the use of the server.

A lower bound for the cycle time τ_i of the process P_i is given by the *ideal cycle time* t_i : it is the cycle time that could be reached if waiting time of receive operations was null. From previous results we can obtain:

$$t_i = \sum_{M_{i,n} \in P_i} \tau_R N_{i,n} P_{M_{i,n}}$$

The average value of the waiting time t_{W_k} of the receive operation in block $B_{i,k}$ may be estimated as first approximation by observing that the worst case for a receive operation implies an empty mailbox and that the processes sending messages to such mailbox have just past the send instruction: in this case the subsequent message will arrive after the completion of the process' cycles. Therefore, t_{W_k} is given by:

$$t_{W_k} = \frac{\sum_{P_j \in S_{i,k}} t_j}{\sum_{P_j \in S_{i,k} \wedge \text{Send}_{j,h} \in P_j} P_{\text{Send}_{j,h}}}$$

where $S_{i,k}$ is the set of processes $P_j \neq P_i$ sending messages to the mailbox on which P_i performs a receive in block B_k .

Using the waiting time of each receive operation, it is possible to estimate route, block and mesh execution time more accurately than we have done before. The *route execution time*, including the waiting time for receive operations, is then given by:

$$\tau_{R_{i,k,h}} = t_{W_k} + t_{R_{i,k,h}}$$

Let's note that this assumption could lead to consider as running a process which is waiting for a message since rescheduling is introduced only at the beginning of each receive operation; in this case a considerable amount of processing time would be wasted. A better choice consists obviously in rescheduling a waiting process until a message becomes available: the performance of the whole system will take advantage of such faster turn around and, probably, the cycle time of some processes might be reduced. Exact evaluation of this behavior requires an accurate simulation of the system by means of a suited queueing model; therefore, for sake of simplicity, we do not further consider this second technique.

The *block execution time* $\tau_{B_{i,k}}$ and the *mesh execution time* $\tau_{M_{i,n}}$ become respectively:

$$\tau_{B_{i,k}} = \sum_{R_{i,k,h} \in B_{i,k}} P_{R_{i,k,h}} \tau_{R_{i,k,h}}$$

$$\tau_{M_{i,n}} = \sum_{B_{i,k} \in M_{i,n}} \tau_{B_{i,k}}$$

As we have done for ideal block execution time, we can obtain the probability distribution for each process P_i ; the average value of the block execution time and its variance are:

$$\tau_{B_i} = \sum_{B_{i,k} \in P_i} \tau_{B_{i,k}} P_{B_{i,k}}$$

$$\sigma_{B_i}^2 = \sum_{B_{i,k} \in P_i} (\tau_{B_{i,k}} - \tau_{B_i})^2 P_{B_{i,k}} = \sum_{B_{i,k} \in P_i} \tau_{B_{i,k}}^2 P_{B_{i,k}} - \tau_{B_i}^2$$

These figures characterize the behavior of instruction blocks: by means of queueing theory and by using the same queueing model adopted for evaluating the waiting time of receive operations, we can estimate as first approximation the cycle time of each process. In this case the interarrival time t_a and the use of the server ρ must consider the real evolution of the processes:

$$t_{\alpha_i} = \frac{\sum_{P_j \in S_i} \tau_j}{\sum_{P_j \in S_i \wedge \text{Send}_{i,k} \in P_j} P_{\text{Send}_{j,k}}} = 1/\lambda_i$$

$$\rho_i = \tau_{B_i} / t_{\alpha_i} = \lambda_i \tau_{B_i}$$

$$\rho = \sum_i \rho_i = \sum_i \lambda_i \tau_{B_i}$$

To simplify the evaluation of cycle time, we introduce the same assumptions adopted for ideal cycle time on probability distribution of interarrival time, handling of

queues and mailboxes and on population of incoming transactions. These allow to estimate the mean service time and its variance as:

$$t_S = \sum_{P_i \in A} q_i \tau_{B_i}$$

$$\sigma_S^2 = \sum_{P_i \in A} q_i (\tau_{B_i} - t_S)^2 = \sum_{P_i \in A} q_i \tau_{B_i}^2 - t_S^2$$

where probability q_i that block $B_{i,k}$ is executed by the processing unit is given by the same formula used in evaluating the waiting time, but it has a different value since interarrival times are different.

From the classic results of queueing theory shown in figures 3 and 4, we obtain the average value $\bar{\tau}_R$ of the total transit time of a block on the processing unit and its variance by considering also the waiting time of receive operations. Finally, the cycle time τ_i of the process P_i may be obtained as:

$$\tau_i = \sum_{M_{i,n} \in P_i} \bar{\tau}_R N_{i,n} p_{M_{i,n}}$$

where $\bar{\tau}_R N_{i,n}$ is the actual average value of mesh execution time. Since we consider probabilistic distribution of computation and cycle time, the system which is generated by using this evaluation method is not completely guaranteed to meet very hard real time constraints; however, probabilistic approach is generally satisfactory.

4.2 Cost function

In each considered state, we evaluate the cost function $c(S)$. As already said, the cost function takes into account the hardware cost and the timing performances of the processes both in the basic and in redundant allocations:

$$c(S) = c_H(S) + \gamma c_T(S)$$

Our goal is to minimize the first component to achieve the cheapest distributed network while timing performances (the second component) are satisfied. The first term $c_H(S)$ of the cost function $c(S)$ gives the cost of the hardware used by allocation S , i.e. the costs of processors, storage and communication links:

$$c_H(S) = c_P(S) + c_S(S) + c_L(S)$$

The storage cost depends on the required amount of storage in a non-linear way: the amount of storage MEM_{req} required for a processing unit is

$$MEM_{req} = MEM_p + MEM_m$$

where MEM_p and MEM_m represent the storage required by processes and mailboxes allocated onto each processing unit. It has though to be considered that storage may not be supplied in a continuous way on memory boards, but only as memory banks of discrete and standard dimensions. It

follows that the amount of storage that we have to insert into a processing unit can be computed from the storage requirements through a non-linear function. The cost of storage $c_S(S)$ is:

$$c_S(S) = \sum_j C(MEM_{req_j})$$

where $C(MEM_{req_j})$ is the storage cost for processing unit j in the network.

The last hardware cost to be considered concerns hardware support to communications: two situations may be envisioned. If a process exchanges messages with a mailbox allocated on the same processing unit, storage is the only requirement for that link: no additional cost is then foreseen since storage requirement has already been accounted for in c_S . Conversely, if the mailbox and the process are on different processing units, some hardware connections are required.

Therefore, the function $c_L(S)$ is:

$$c_L(S) = \sum_{h,k} c_{LNK_{h,k}}$$

where $c_{LNK_{h,k}}$ is the cost of the hardware support to communication between process h and mailbox k .

The second term $c_T(S)$ of cost function $c(S)$ considers the timing constraints on the cycle time of processes both in the fault-free and in the faulty case; therefore it is

$$c_T(S) = c_W(S) + c_F(S)$$

It has been already said that when no processing unit of the distributed system is faulty, the cycle time τ_i of each active process P_i must satisfy the constraint $\tau_i \leq \tau_i^*$ imposed by the user. This kind of constraint is represented in the cost function $c_T(S)$ by the term $c_W(S)$. It is a binary function which is 0 iff constraint $\tau_i \leq \tau_i^*$ is satisfied for each process P_i in the basic allocation of solution S ; it is infinite otherwise since the allocation can not be accepted if timing performances are not fulfilled.

If a processing unit fails, the system should be able to survive by activating the redundant copies of the interrupted processes which were allocated onto the faulty processing unit: after the transition time required by such operation, the system is allowed to work with more relaxed timing constraints: all processes (basic and redundant) running on processing units involved in the fault tolerant action will then have to fulfill a new constraint $\tau_i \leq \tau_i^+$ being τ_i^+ not smaller than τ_i^* .

The term $c_F(S)$ of the cost function $c_T(S)$ takes into account all relaxed constraints for any faulty processing unit; it is

$$c_F(S) = \sum_{j=1}^n c_{F_j}(S)$$

where $c_{F_j}(S)$ is a binary function which is 0 iff $\tau_i \leq \tau_i^+$ for each active process P_i when processor j is faulty, otherwise it is infinite because the allocation is not able to satisfy the

relaxed constraints. Note that timing constraints must be evaluated by assuming one processing unit at time as faulty since basic and redundant allocations must assure that the system survives after the failure of *any* among its processing units.

5. ALGORITHM IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section we present the experimental results achieved by using simulated annealing for hardware dimensioning and software allocation. Different alternatives have been implemented to improve the performance of the algorithm.

At first we have considered the theoretical approach:

- a state S is the mapping of processes and mailboxes upon processing units;
- neighboring states are generated by randomly reallocating one process at time with the mailboxes from which it receives messages;
- annealing schedule consists in reducing the temperature according with the rule

$$T = \frac{rL}{\ln(n + n_o + 1)}$$

- one iteration is performed at each temperature value.

At each iteration of the algorithm a neighboring state is generated by changing the allocation of a process or a mailbox in the current state. Usually, the generate function randomly selects the item which must be reallocated. If the hardware architecture has to be designed, the *generate* function tries also to reduce the number of required processing units by removing items allocated onto a processing unit randomly selected.

For each new state the cost function is evaluated by computing the hardware cost of the allocation and by estimating the cycle time of each process as shown in the previous section. The new state is accepted both if the cost function decreases with regard to the current state and if the cost function is greater than in the current state, provided the random acceptance rule is satisfied (ie the cost function is not *too* high for the current temperature). Note that our aim is minimization of hardware costs: timing performances are not optimized, they must only stay above a minimum threshold both in the fault free situation and when a faulty processor is present.

This instance of the annealing algorithm implies the exact evaluation of annealing parameters, such as the radius of the graph underlying the Markov chain and the Lipschitz constant of the cost function; this requires a detailed knowledge of the structure of the Markov chain and of the cost function: thus, this approach is not feasible since the whole solution space should be explored.

As an alternative, we approximate the value for annealing parameters while preserving the overall behavior of

the algorithm. For radius r , we consider its worse-case value r_{apx} : we suppose that there is a path connecting two states S_1 and S_2 through all the other states of the graph: in this case r is given by $|S| - 1$, where $|S|$ is the cardinality of the set of states. Existence of such path, in our case, is assured by the definition we gave of *state*. By that definition it follows immediately that the the graph underlying the Markov chain will be strongly connected.

The Lipschitz constant L is the maximum variation of cost function between two neighboring states: to estimate L we consider only the term c_H of the cost function, since the term c_T is only a binary function which translates in a mathematic expression the capability of satisfying performance constraints. The worst-case value L_{apx} should be estimated on the considered network architecture; but such operation implies a detailed knowledge of the whole solution space. Therefore, we approximate the worst-case by means of a value that can never be reached by any allocation on the given architecture. To such purpose we consider two imaginary architectures which stress the concepts of dispersion and aggregation costs: in the first case only one process is allocated on each processing unit in the first architecture, while the second one is composed by only one processing unit. L_{apx} is the difference between the value of the cost function c_H for these architectures.

Implementation and experiments have shown that a drawback of this approach consists in the possible high number of iterations required to obtain the final solution.

A faster convergence rate may be achieved by adopting the following alternative annealing schedule [15]:

$$T_k = \alpha_k T_{k-1}$$

where k is number of changes of temperature and α is a tuning factor affected by the difference between the cost function in the last two accepted states. Experiments show that α ranges between 0.8 and 0.95: high values of α will be adopted when the cost function varies rapidly, while low values are used when the cost function is quite steady.

Convergence to the global optimum is preserved if the system is allowed to reach *equilibrium* at each value of the parameter T , ie a sufficient number of iterations must be performed at each temperature T , so that the state probability distribution is close to the stationary probability distribution of the considered Markov chain. This technique is referenced in figure 1 as the *inner loop*. The number of iterations is chosen accordingly with the values of the cost function before the two last temperature reductions: small differences lead to a reduced number of iterations (about 5), while high changes may require some tens of iterations.

A comparison of the performance of this alternative scheduling with respect to the basic one is given in figures 5, 6 and 7 and in table 2: in such figures a typical behavior of cost function, the acceptance ratio (i.e. the probability of accepting moves) and the number of iteration versus the annealing temperature are shown. Examples of CPU time

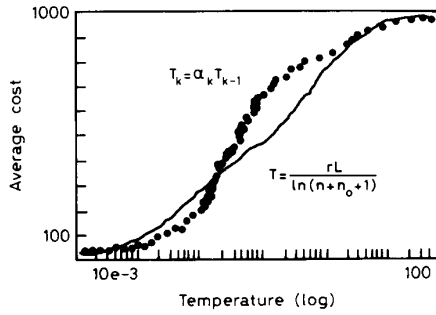


Fig. 5 Annealing Schedule

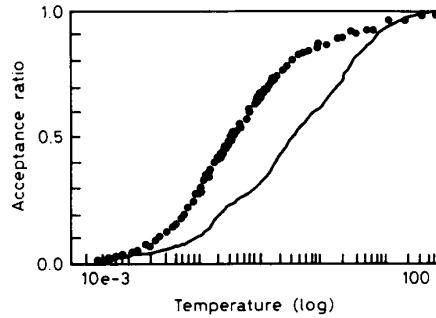


Fig. 6 Acceptance Ratio

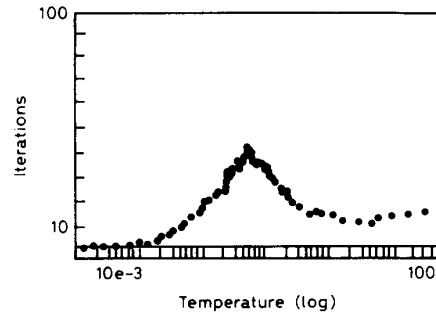


Fig. 7 Number of Iterations

obtained by running experimental implementations of the algorithms on VAX 11/780 are given in the table.

A global optimum may be more easily reached if the starting temperature T_0 for the annealing process has been accurately determined: if T_0 is too low, the optimum may not be reached since the system has frozen too early. To avoid this possibility, the *hot condition* has been introduced [16]: the system is said to be *hot* enough if

$$\sigma \ll T_0$$

where σ is the standard deviation of the probability distribution of the cost function. To determine the such value, a preliminary exploration of the solution space is performed: during such exploration all states are accepted, ie temperature is assumed as infinite. The cost function c_H is computed in each considered state to allow evaluation of standard deviation. The initial temperature is then given by

$$T_0 = k\sigma$$

where k is obtained assuming a normal cost distribution and selecting a temperature high enough to accept with probability P a configuration whose cost is 3σ worse than the present one, that is

$$k = - \frac{3}{\ln P}$$

Note that since only few moves are required to achieve equilibrium at high temperatures, the overhead in using a high value of k is insignificant. Annealing dependence on initial temperature is shown in figures 8 and 9 and in table 3: typical behaviors of annealing schedule, iterations number versus temperature and CPU time are given.

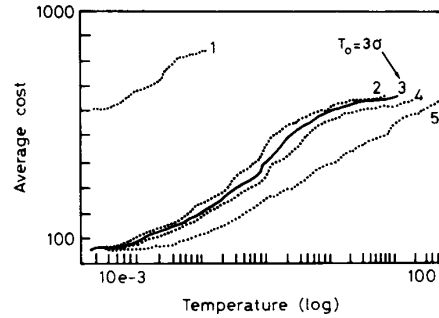


Fig. 8 Annealing Schedule

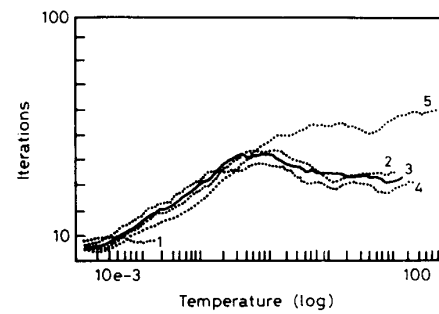


Fig. 9 Number of Iterations

Convergence of the algorithm is only asymptotical if approximate evaluation of r and L or the alternative scheduling is adopted: thus a suitable stopping criterion has to be defined. An exact stopping criterion should consider the difference between the next-to-least-cost solution and the least-cost solution. But their computation requires

TABLE 2
CPU time

Example →	#1	#2	#3	#4	#5	#6
n° of processes →	25	58	79	107	186	293
CPU time (sec.) if $T = \frac{rL}{\ln(n + n_o + 1)}$	199	638	1204	2471	6415	23014
CPU time (sec.) if $T_k = \alpha_k T_{k-1}$	168	494	785	1804	4583	15724

TABLE 3
CPU time

Example	#1	#2	#3	#4	#5
T_o	0.1	60	98	110	150
CPU time (sec.)	237	3625	3271	3598	4759

the complete knowledge of the space of solutions: therefore, a stopping criterion (validated by experiments) is to assume that the optimum has been reached when the last three accepted solutions at different temperatures are equal.

An alternative technique identifies first the basic allocation and the hardware architecture by considering all processing units as fault free; then, the second optimization step produces the redundant allocation by assuming one processing unit at time as faulty, but without modifying the basic allocation previously generated.

Obviously, complexity of the allocation problem decreases because in each of the two subsequent steps a smaller number of variables is involved: in fact, experiments have shown that the computation time is reduced about of 15%. On the other hand, the allocations produced by this approach are 10-20% worse; such behavior is due to the fact that, generally, a global optimum solution is not equivalent to the result of two subsequent local optimizations.

We observed that the global-optimization algorithm is able to reduce the hardware architecture (in particular storage), though, sometimes, the cycle time of some processes belonging to the basic allocation may grow. Anyway, this fact does not constitute a drawback until all constraints in the fault-free architecture and all relaxed constraints are satisfied; in fact, cycle time are not to be minimized, but only fulfilled by the allocation.

Therefore, since the design of the distributed system and the allocation of processes are performed only once and off-line with respect to the execution of the processes themselves, we think that this alternative approach has not to be preferred to the algorithm extensively discussed in this paper: a reasonable greater design time is acceptable to achieve a considerably better solution.

6. CONCLUDING REMARKS

In this paper we discuss a new approach to dimensioning of a distributed system and to allocation of processes: fault-tolerance capabilities are also considered to assure a fast restart after an hardware failure, even if with degraded performances. To achieve better results we reverse the classical approach to resource allocation in a distributed environment. Given a software application, we adopt an allocation algorithm which does not necessarily take into account existing hardware architectural constraints [18]: the algorithm supposes to allocate the software application onto an ideal hardware network where the number of processing units and the number of communication links is not a priori defined. Thus, only strictly software-related characteristics are taken into account during this *virtual* allocation: in this way it is possible, once optimum allocation has been achieved, to assemble the hardware architecture following the guidelines obtained by the allocation, thus satisfying all possible software requirements.

To optimize software allocation we adopt the Simulated Annealing heuristic algorithm, particularly suited for finding global optimum in large NP-complete combinatorial problems having also local optima. This algorithm, compared with other heuristic algorithms (such as, e.g. branch-and-bound, or greedy algorithms) gives notable advantages when storage requirements are concerned, thus allowing to cope with problems having quite large set of states since no track of previous steps is required. Efficiency may be the critical point of this algorithm: tuning of its parameters (such as annealing schedule, initial temperature, inner loop criterion) may heavily affect its performance. Great efforts should thus be put in this tuning phase.

If (as often happens due to budget considerations and when non mission-critical systems are designed) cost/performance ratio limits hardware availabilities, our approach may be easily modified so to take into account this further constraints. If the number of hardware devices is upper-bounded, each component of $c_H(S)$ will be described by a non-linear function. Suppose, for example:

- N to be the maximum number of available processors
- n to be the number of processors required by a given allocation.

$c_p(S)$ will then be equal to:

$$\begin{cases} n \times \text{cost of each processor,} & \text{for } n \leq N; \\ \infty, & \text{for } n > N. \end{cases}$$

The same obviously holds also for the other components of the hardware cost function.

Another case in which hardware characteristics affect allocation is when processors constituting the network are inhomogeneous (which possibly leads to different clock rates thus, ultimately, to different instruction execution time). In this case process cycle time will have to be *completely* re-evaluated at each step of the algorithm (ie at each proposed allocation) depending on which processing unit of the hardware network the process is allocated.

If the hardware architecture is already defined, our technique is an alternative to the other methods of resources allocation. In this case, the cost function c_H is fixed; the optimization algorithm must therefore consider the following cost function since software application must be mapped onto the fixed hardware architecture:

$$c(S) = \gamma c_r(S) + \epsilon c_o(S)$$

where

- γ and ϵ are the scale factors which adjust dimensions of these two terms.
- $c_o(S)$ is a two-valued function (with value in $\{0, \infty\}$), which accounts for the possibility of storing the processes allocated in each processing unit.

Another possible change in the basic algorithm here presented consists in allowing the algorithm to minimize the cycle time: in many critical applications the cycle time must be not greater than a predefined upper bound, but it must be often preferred that it is the lowest as possible below such bound. This additional requirement can be taken into account by adding another term c_r to the cost function:

$$c_r(S) = \sum_{S_h} \tau_h$$

where S_h is the subset of processes whose cycle time must be minimized.

From these examples we note that the hill-climbing approach presented in this paper is more than an optimization algorithm: it is a quite simple and general framework to design effective optimization techniques for hardware dimensioning and software allocation for distributed systems.

REFERENCES

- [1] H. S. Stone, S. H. Bokhari, "Control of distributed processes", *IEEE Computer*, 1978 July
- [2] P. Heidelberger, K. S. Trivedi, "Queueing network models for parallel processing with asynchronous tasks", *IEEE Trans. Computers*, vol. C-31, 1982 Nov

- [3] O. I. El-Dessoukian, W. H. Huan, "Distributed enumeration on network computers", *IEEE Trans. Computers*, vol C-29, 1980 Sept
- [4] W. W. Chu, "Optimal file allocation in a multiple computing system", *IEEE Trans. Computers*, vol C-18, 1969 Oct.
- [5] P. R. Ma, "A model to solve time-critical application problems in distributed computer systems", *IEEE Computer*, 1984 Jan.
- [6] S. Kirkpatrick, C. Gelatt, M. Vecchi, "Optimization by simulated annealing", Int. Rep. IBM Computer Sc./Eng. Tech. Watson Res. Center, Yorktown Heights, NY, 1984
- [7] D. S. Johnson, "Simulated annealing performance studies", *Simulated Annealing Workshop*, Yorktown Heights, 1984 April
- [8] F. Romeo, A. Sangiovanni-Vincentelli, "Probabilistic hill-climbing algorithms: properties and applications, UCB/ERL M84/34, University of California, Berkeley, 1984.
- [9] P. Brich Hansen, "Concurrent programming concepts", *ACM Computer Survey*, 1973 Dec.
- [10] G. R. Andrews, F. B. Schneider, "Concepts and notations for concurrent programming", *ACM Computer Surv.*, 1983 Mar.
- [11] M. R. Garey, D. S. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, Freeman & Co., 1979
- [12] A. Thomasian, P. F. Bay, "Analytic queueing networks models for parallel processing of task system", *IEEE Trans. Computers*, vol C-35, 1986 Dec.
- [13] M. Bedina, F. Distante, "SW & HW Resources allocation in a multiprocessor system: an architectural problem", *Proc. EUROMICRO'84*, Copenhagen, 1984 Aug.
- [14] D. Mitra, F. Romeo, A. Sangiovanni-Vincentelli, "Convergence and finite time behavior of simulated annealing", Tech. Rep. UCB/ERL M85/23, University of California, Berkeley, 1985
- [15] C. Sechen, A. Sangiovanni-Vincentelli, "The Timberwolf placement and routing package", *Proc. Custom Integrated Circuits Conf.*, 1984 May
- [16] S. White, "Concepts of scale in simulated annealing", *Proc. ICCD'84*, 1984
- [17] L. Kleinrock, *Queueing Systems*, Wiley & Sons Ed., 1975
- [18] F. Distante, V. Piuri, "Distributed architecture design to match optimum process allocation: a simulated annealing approach", *Proc. RTSS-8*, San Jose, 1987

AUTHORS

Fausto Distante; Department of Electronics; Politecnico di Milano; piazza L. da Vinci 32; I20133 Milano, ITALY

Fausto Distante obtained the Dr. Ing. degree in Electronic Engineering from Politecnico di Milano in 1982. Since 1983 he is a researcher of the C. N. R. (Italian National Science Foundation) at Department of Electronics of the Politecnico di Milano. His research interests include VLSI multiprocessor systems, fault tolerance, testing and real time systems; computer music is his main hobby. He is a council member of EUROMICRO (The European Association for Microprocessing and Microprogramming).

Vincenzo Piuri; Department of Electronics; Politecnico di Milano; piazza L. da Vinci 32; I20133 Milano, ITALY

Vincenzo Piuri obtained the Dr. Ing. degree with honor in Electronic Engineering from Politecnico di Milano in 1984. He was engaged in research activities at Department of Electronics of Politecnico di Milano with a grant of ITALTEL S. I. T. Spa; he cooperated at the "CAD VLSI for Telecommunications" Project, sponsored by EEC, and to the National Microelectronics Project, sponsored by the C. N. R. His research interests include distributed systems, fault tolerance, CAD for VLSI design and testing.

Manuscript TR88-354 received 1988 March 11; revised 1988 June 27.

IEEE Log Number 25468

◀ TR ▶