On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques

MARISA LÓPEZ-VALLEJO
Universidad Politécnica de Madrid
and
JUAN CARLOS LÓPEZ
Universidad Castilla-La de Mancha

This paper presents an in-depth study of several system partitioning procedures. It is based on the appropriate formulation of a general system model, being therefore independent of either the particular co-design problem or the specific partitioning procedure. The techniques under study are a knowledge-based system and three classical circuit partitioning algorithms (Simulated Annealing, Kernighan&Lin and Hierarchical Clustering). The former has been entirely proposed by the authors in previous works while the later have been properly extended to deal with system level issues. We will show how the way the problem is solved biases the results obtained, regarding both quality and convergence rate. Consequently it is extremely important to choose the most suitable technique for the particular co-design problem that is being confronted.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**]—*computer-aided design*

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: Hardware-software co-design, hardware-software partitioning, system modeling, general optimization procedures, clustering, cost functions, expert systems, fuzzy logic

1. INTRODUCTION

Hardware-software partitioning deals with the assignment of parts of a system description to heterogeneous implementation units: ASICs (hardware), standard or embedded microprocessors (software), memories, and so forth. This

This work has been partially supported by the Spanish Ministry of Science and Technology under project CORE, TIC2000-0583-CO2.
Authors' addresses: M. López-Vallejo, Dpto. Ingenieria Electronica, ETSI Telecomunicacion, Ciudad Universitaria s/n, 28040 Madrid, Spain; email: marisa@die.upm.es; J. C. López, Escuela Superior de Inforrmática Paseo de la Universidad, 4, 13071 Ciudad Real, Spain; email: juancarlos.lopez@uclm.es.

is a key task in system level design, because the decisions made during this step directly impact the performance and cost of the final implementation. Hardware-software codesign addresses the development of these complex heterogeneous systems looking for the best trade-offs among the different solutions. Several problems can be considered under the codesign paradigm [Micheli 1994]:

—ASIPs synthesis *(Application Specific Instruction-set Processor)*.
—Execution acceleration by means of *custom-computing machines*.
—Design of *System-on-a-chip* (ASICs with embedded processor cores).
—*Embedded-systems* design.

The aim of the partitioning task is to find a design implementation that fulfills all the specification requirements (functionality, goals and constraints) at a minimum cost. In the traditional design strategies, the system designer decided which blocks of the system could be implemented in hardware and which could be realized as software running on a standard processor, taking into account his/her own knowledge as an expert in the field. To automate such a difficult labor, several algorithms and techniques have been developed in different codesign environments [Ernst et al. 1993; Gupta and Micheli 1993; Kalavade and Lee 1997; Vahid 1997; Eles et al. 1997; López Vallejo et al. 1999]. All these approaches can work perfectly within their own codesign environments, but due to the enormous differences among them, it is not possible to compare the results obtained. In order to qualitatively and quantitatively analyze all these techniques we have formulated a common model for a general codesign environment, and we have implemented some of these techniques over this model.

In this paper, we present an in-depth analysis of system partitioning techniques. The implementation of the different methods is strongly based on the construction of a model whose generality will allow us to deal with different codesign problems and to find the best implementation for the particular problem being tackled. As a result of this study a better knowledge of the problem has been obtained, as well as a better understanding of the methods and their ranges of application.

The paper is structured as follows. First, other approaches to the problem will be reviewed. The next section presents the problem formulation used as a foundation for the different techniques. After that, several partitioning techniques will be introduced, outlining their major features. The paper ends with a summary of the results and some conclusions.

2. RELATED WORK

Important work has been done in hardware-software partitioning in recent years. Nevertheless, a comparison among the different solutions is almost impossible, because of the large differences in the co-design environments and the lack of benchmarks. We can enumerate many basic aspects that characterize a system partitioning environment, apart from the particular technique or algorithm chosen. For instance, some of them are the initial specification, which fixes the abstraction level, the kind of application the environment

Table I. Technical Characteristics of the Partitioning Task in Several Codesign Environments

Group	Specification	Application	Grain	Algorithm	Model	Architecture
Vulcan [Gupta 1993]	HardwareC	Data oriented	Fine (instructions)	Group Migration	DFG Set	Standard
Cosyma [Ernst 1993]	C	Data oriented	Fine (basic blocks)	Simulated Annealing	Syntactic tree	Standard
Ptolemy [Kalavade 1997]	Silage	Real Time	Coarse (tasks)	GCLP (constructive heuristic)	CDFG	DSP based embedded system
Vahid [Vahid 1997]	SpechCharts	Control	Coarse (processes)	Clustering, *min-cut*, S. Annealing	SLIF (access graph)	Standard
Lycos [Madsen 1997]	C or VHDL		Fine (basic blocks)	Dynamic Programming		
Cadlab [Eles 1997]	VHDL	-	Coarse (loops, processes)	S. Annealing Tabu Search	Process Graph	Standard
DDEL [Srinivasan 1998]	C and VHDL	Data oriented	Coarse (tasks)	Genetic	Task Graph	Standard
Wolf [Wolf 1997]	Object Oriented	Real Time	Coarse (tasks)	Heuristic	Task Graph	Parametric

focuses on (mainly data-intensive or control-oriented approaches), the model of computation that has been chosen as representation and the selected granularity. In this section we will review some of the previous approaches to hardware-software partitioning that are important for their early appearance or their novelty. Table I summarizes some of these groups and their major technical characteristics. Those works more related to the partitioning procedures described here will be analyzed in detail in the corresponding sections.

The two first algorithms that solved hardware-software partitioning were presented by Cosyma and Vulcan. In Cosyma [Ernst et al. 1993], a simulated annealing algorithm with fine granularity is applied. The initial specification is an extension of the C programming language translated into a syntax graph. The cost function minimizes the system execution time, extracting partitioning objects (basic blocks) from software to hardware. Vulcan [Gupta and Micheli 1993] also uses fine granularity, language-level operations, obtained from the initial specification in HardwareC. The partitioning algorithm is based on iterative improvement, and extracts software blocks from an initial all-hardware solution considering the timing constraints.

A coarse grain constructive algorithm was developed in Ptolemy [Kalavade and Lee 1997] extending list-based scheduling. Two important points characterize this algorithm: (1) it is able to adapt the objective function to global or critical measures, and (2) different hardware implementations are considered. This environment concentrates on real time applications implemented by means of DSP-based architectures.

An important work in functional partitioning has been done by Vahid et al. [Vahid 1997; Vahid and Gajski 1995a], applying classical partitioning

algorithms to hardware-software architectures (including the K&L heuristic, simulated annealing, clustering, etc.). As will be noted in the paper, this work is strongly based on its intermediate format (SLIF).

The application of artificial intelligence techniques has proven its usefulness when dealing with the system partitioning problem. The DDEL group [Srinivasan et al. 1998] performs system partitioning using a genetic algorithm that includes hardware space exploration, following a coarse grain approach. Another recent approach based on genetic algorithms is proposed by Dick et al. [Dick and Jha 1998], which considers power and real time in the optimization process. The expert system described in López Vallejo et al. [1999] will be explained in depth in Section 5.

Many other important contributions could be commented on in this section. For example, Cool [Niemann and Marwedel 1996] performs a hardware-extraction approach by means of integer linear programming, the same procedure used by Madsen within the LYCOS co-synthesis system [Madsen et al. 1997]. Wolf [1997] postulates a coarse grain architectural algorithm for the cosynthesis of distributed embedded computing systems. Eles et al. [1997] describe a comparison between simulated annealing and tabu search that will be commented on in Section 4.1.

Finally, it is worth mentioning here the recent and interesting work on flexible granularity proposed by Henkel et al. in [Henkel and Ernst 2001]. In this article an in-depth study on hardware-software partitioning is presented, paying special attention to the use of different granularities, the detailed description of estimation methodologies and the formulation of a multidimensional objective function. The algorithm used in this approach is simulated annealing and will also be commented upon in depth in Section 4.1.

3. SYSTEM MODEL FOR PARTITIONING

The resolution of a problem requires the definition of a model representing all the important issues related to the specific problem. In this section, the system partitioning problem will be characterized and its corresponding model [López Vallejo 1999] will be defined.

The global information flow of the partitioning procedure presented here is depicted in Figure 1. The input to the partitioning process is an execution flow graph that comes from the initial system specification, described using high-level specification languages. This is a directed and acyclic graph where vertices stand for basic computation units and edges represent data and control dependencies. Thus, vertices can be large pieces of information (tasks, processes, etc.) or small ones (instructions, operations), following respectively a coarse or fine granularity approach.

Every graph vertex is labeled with several attributes obtained after applying estimation procedures [Carreras et al. 1996]. In detail, these pieces of information for a node v_i are: hardware area (ha_i), hardware execution time (ht_i), software memory size (ss_i), software execution time (st_i) and the average number of times the task is executed (n_i). Edges have also associated a

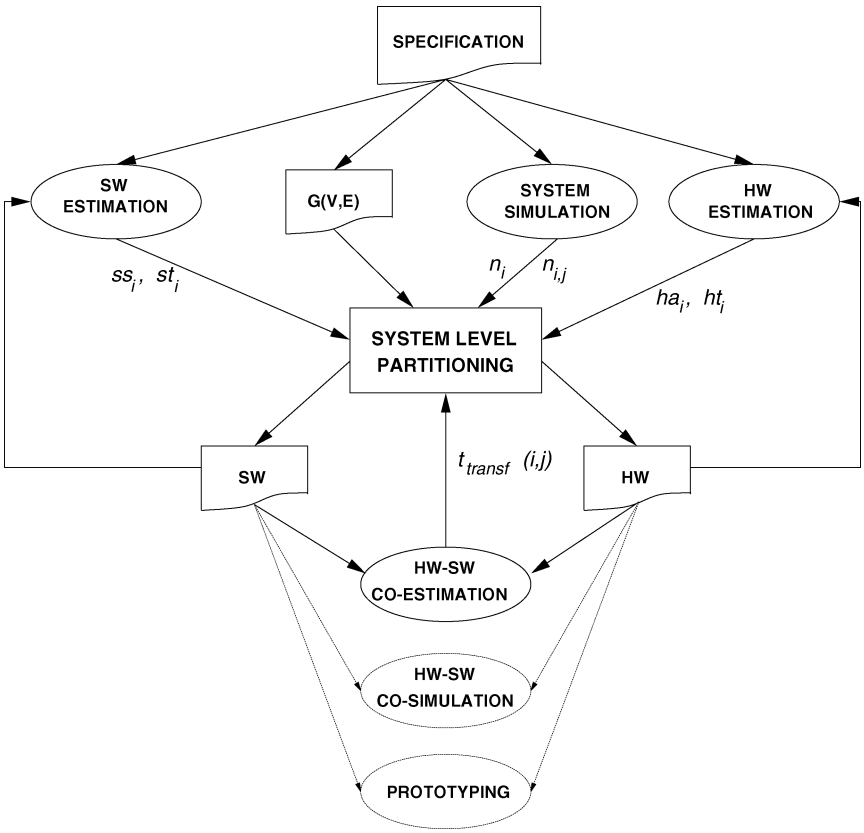

Fig. 1. Flow of information within the partitioning model.

communication value ($t_{comm}(i, j)$) obtained from three components: the transfer time ($t_{transf}(i, j)$), the synchronization time ($t_{synch}(i, j)$) and the average number of times the communication takes place n_{ij}.

As is well known, system partitioning is clearly influenced by the target architecture onto which the hardware and the software will be mapped. Here the target architecture considered consists of one processor running the software, several dedicated co-processors (based on ASIC or FPGA) and shared memory accessed through a common bus. Interface modules are used to connect the processor and the ASIC to the bus. This is a model in which we have used just one hardware co-processor during our experiments for the sake of simplicity. Hardware and software processes can be executed concurrently in the standard processor and the application-specific co-processor.

In order to concentrate on the partitioning techniques, we have considered this architecture model that is very restrictive and simplifies the problem in the context of current architectures of embedded systems. Nevertheless, the model we have used can be extended to consider current system on chip architectures, as well as distributed embedded systems, supporting several programmable processors and complex communication structures.

The outcome of the partitioning tool is not only an assignment of blocks to hardware or software implementations, but also their scheduling (starting and finishing time) and some information about the communication produced in the interface.

The validity of the solution is measured by means of some *design-quality attributes* which must perfectly describe the solution. These attributes are normally design costs and performance parameters. In particular, we have used as quality attributes the required hardware area for the co-processor, A_p, the design latency, T_p, calculated by scheduling the system graph, and the required memory space, M_p.[1] A design constraint is associated with each attribute. In our case, we will have the maximum available area, A, the maximum allowed execution latency T, and the maximum memory space, M.

Scheduling is implemented by means of a list-based scheduling algorithm. The scheduler takes into account the timing estimates of every vertex in the graph and the dependencies among them. As output, it gives the design latency, T_p, and the communication cost produced in the hardware-software interface.

Here we must define other important parameters, including the *extreme values*. These parameters are obtained through the extreme implementations, the *all-hardware* and the *all-software* solutions. These parameters will be used as a reference in the different techniques, bounding the constraint values. From the *all-hardware* solution we obtain, $MinT$, the minimum design latency and $MaxA$, the maximum hardware area. From the *all-software* solution we obtain two more parameters: $MaxT$, the maximum design latency and $MaxM$, the maximum memory space. To ensure that we do not look for an impossible solution, the system constraints must always verify: $0 \leq A \leq MaxA$, $0 \leq M \leq MaxM$ and $MinT \leq T \leq MaxT$.

3.1 Problem Formulation

More formally, the hardware-software partitioning problem can be framed as follows. Given a system description in the form of a task graph, directed and acyclic, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges, with a timing goal (for instance a latency T), a target architecture $\mathcal{D} = (\Pi, \Gamma_1, \Gamma_2, \ldots, \Upsilon)$ (where Π stands for the standard processor, Γ_i represents the i-th hardware coprocessor and Υ is the interface model) accompanied by a set of architectural constraints (A, maximum hardware area, M, maximum memory size, B, bus transfer rate and W, bus width), and a cost function that evaluates the quality of a given solution $\mathcal{F} : \mathcal{G} \Rightarrow [0, \infty)$; the hardware-software partition, $\mathcal{P}$, is a function that assigns every vertex of $\mathcal{G}$ to a processing unit of $\mathcal{D}$ with a starting time $t \in [0, \infty)$ while minimizing the cost

[1]Many other interesting quality attributes can be considered here. For instance, power consumption is a key issue to look at in the case of portable embedded systems. This new attribute could be included here once the corresponding estimation tool is implemented and the model can be extended with a new parameter for the graph nodes: their estimate on power consumption.

function $\mathcal{F}$. Formally:

$$\mathcal{P} : \mathcal{V} \Rightarrow \{\Pi, \Gamma_1, \Gamma_2, \ldots \Gamma_n\} \times [0, \infty) /$$
$$\begin{cases} \forall v \in \mathcal{V} \quad \mathcal{P}(v) = (i, t) \quad \text{with} \begin{cases} i \in \{\Pi, \Gamma_1, \ldots \Gamma_n\} \\ t \in [0, \infty) \end{cases} \\ \mathcal{F} : \mathcal{G} \Rightarrow [0, \infty) \quad \text{is minimized} \end{cases}$$

It must also be verified that:

(1) $\forall v_1, v_2 \in \mathcal{V} \quad \mathcal{P}(v_1) \neq \mathcal{P}(v_2)$: *condition of time-space exclusion*.

(2) $\forall v_1, v_2 \in \mathcal{V} / v_1 < v_2$, ($<$ expresses an order relation) $if \quad \mathcal{P}(v_1) = (i, t_1)$ and $\mathcal{P}(v_2) = (j, t_2) \Rightarrow t_1 + t(i, v_1) < t_2$, *condition of data dependencies among vertices*, where $t : \mathcal{D} \times \mathcal{V} \Rightarrow [0, \infty)$ is a function that provides for every vertex $v \in \mathcal{V}$, its execution time in the processing unit $i \in \{\Pi, \Gamma_1, \ldots \Gamma_n\}$.

(3) If $v_{end} \in \mathcal{V} / \forall v_j \in \mathcal{V}, v_j \neq v_{end}, v_j < v_{end}, \mathcal{P}(v_{end}) = (k, t_{fin}) \Rightarrow t_{fin} + t(k, v_{end}) \leq T$, *condition of adjustment of time goal*.

(4) Let Φ_{hw} be $\Phi_{hw} = \{v_j \in \mathcal{V}, \quad \mathcal{P}(v_j) = (k, t_j) / k \in \{\Gamma_1, \ldots \Gamma_n\}\}$ it must be verified that $\alpha(\Phi_{hw}) \leq A$, *condition of adjustment of the area constraint*, with α being the function that estimates the coprocessor hardware area with relation to Φ_{hw} in a given point of the design space.

(5) Let Φ_{sw} be $\Phi_{sw} = \{v_j \in \mathcal{V}, \quad \mathcal{P}(v_j) = (k, t_j) / k = \Pi\}$; it must be verified that $\mu(\Phi_{sw}) \leq M$, *condition of adjustment of the memory size*, with μ being the function that evaluates the memory space needed to run a software code with relation to Φ_{sw}.

4. CLASSICAL PARTITIONING METHODS

The introduction of system-level issues within classical circuit partitioning algorithms is complex due to the different nature (hardware and software) of the processing elements. Consequently, the application of these partitioning algorithms requires strong modifications. In particular, the following algorithms have been adapted to solve this problem: the simulated annealing stochastic algorithm, the Kernighan&Lin heuristic (K&L in the following) and hierarchical clustering techniques.

4.1 Simulated Annealing

Simulated annealing [Kirpatrick et al. 1983] is a well-known optimization procedure that emulates the physical annealing process. It can solve any combinatorial optimization problem if the quality of the proposed solution can be measured by means of a cost function. This is the case of hardware-software partitioning, which is why this method has been previously used to solve the problem [Ernst et al. 1993; Eles et al. 1997]. Its major advantage is its generality, while its main drawback is the long computation time required.

For simulated annealing to work, two main issues must be resolved: the cooling schedule and the cost function. We have used a dynamic cooling

schedule [Huang et al. 1986] for two main reasons:

—This schedule, based on the statistical definition of parameters, allows the algorithm to work on a range of problems and does not require the tuning of the multiple algorithm parameters (cooling speed, equilibrium criteria, etc.)
—The algorithm convergence rate is much faster.

The formulation of the cost function is fundamental to obtaining good results. The next section is devoted to the formulation of a general and efficient cost function [López Vallejo et al. 2000]. Other approaches based on simulated annealing deal with the cost-function straight-forwardly (just to minimize the execution time [Ernst et al. 1993]) or pay little attention to the cooling schedule, consequently requiring the tuning the algorithm parameters for each execution [Eles et al. 1997]. Recent work [Henkel and Ernst 2001] has improved these initial formulations and proposes the use of a dynamic cooling schedule and defines a multi-objective function that trades diverse goals and constraints. This is performed in a way similar to the one proposed here [López Vallejo et al. 2000], but only for area and performance metrics, while our formulation can deal with many diverse attributes in a unified way.

4.1.1 *Cost Function Formulation.* The main goal of a cost function is to measure the quality of a given solution and to guide the algorithm to the best solution. As stated before, the important information related to system partitioning is:

—The global cost associated with the solution, measured by the quality attributes.
—The design constraints and goals.

Design constraints must define the design space and cost issues must help measure the quality of the solution. The type of cost related to the quality attributes must also be taken into account: fixed costs must be considered differently than variable costs. Our formulation incorporates all these points.

The proposed cost function includes several correction terms, $\mathcal{F}_C()$, for each design goal affected by a specific constraint [López Vallejo et al. 2000]. It can be formulated as follows:

$$\mathcal{F}(\mathcal{P}) = \sum_i k_i \times \frac{C_i(\mathcal{P})}{C_i} + \sum_i k_{c_i} \mathcal{F}_C(C_i, C_i(\mathcal{P})) \tag{1}$$

where C_i, the design constraint applied to the *i-th* quality attribute, $C_i(\mathcal{P})$, of a given solution $\mathcal{P}$, has been used as a normalization parameter, and k_{c_i} is the weight factor for the correction terms.

Three different techniques can be used to correct the objective function:

—*Mean Square Error Minimization*, which helps the algorithm find a solution tuned to the constraints.
—*Barrier Techniques*, which forbid the exploration of solutions outside the allowed design space.

—*Penalty Methods*, which strongly punish the exploration of solutions that would produce medium or large constraints overhead, but allow the exploration of regions close to the boundaries defined by the constraints.

A typical way of tuning system constraints is minimizing the mean square error between quality attributes and their corresponding constraints. This kind of expression adjusts the attributes to the design goals and constraints. These functions can be applied to particular goals that should be completely fulfilled instead of minimized. For instance, in the system partitioning problem, if the target architecture includes an FPGA-based coprocessor, it is a good policy to fully exploit all the resources the FPGA provides. This is due to the fact that the FPGA has a fixed cost, and its maximum exploitation generally results in performance improvement. Nevertheless, it would not be suitable for those quality attributes that have variable cost, as they do in an ASIC area or timing measures.

The general expression for Mean Square Error based correction terms is:

$$\mathcal{F}_C(C_i, C_i(\mathcal{P})) = \frac{(C_i(\mathcal{P}) - C_i)^2}{C_i^2} \tag{2}$$

This correction term, applied to the general expression shown in Equation (1), only contributes to the cost of the final solution when the attribute adopts a value not tuned to the constraint. We should note here that this kind of correction term equally penalizes any deviation from the design objective.

Barrier functions [Luenberger 1984] provide a clear boundary around the design space, in such a way that the cost associated with a solution outside of the design space will be infinity. This is performed by placing asymptotes in the boundaries defined by the constraints. The analytical expression for these terms is:

$$\mathcal{F}_C(C_i, C_i(\mathcal{P})) = \frac{1}{b[C_i, C_i(\mathcal{P})]} \tag{3}$$

where $b[C_i, C_i(\mathcal{P})]$ is the barrier function. An example of a barrier function can be:

$$b[C_i, C_i(\mathcal{P})] = \max\{0, [C_i - C_i(\mathcal{P})]\} \tag{4}$$

This type of correction term can be used for hard design constraints, because it ensures that the constraint will never be violated. However, its application contributes to the final cost of valid design solutions, requiring a careful adjustment of the objective function weights and a modification of the final cost interpretation.

Finally, when the system constraints are not too hard, the use of penalty functions [Luenberger 1984] may be more suitable. Penalty functions do not contribute to the cost function when the solution is in the allowed search space. This kind of function is not so restrictive as the barrier functions, since solutions around the border of the allowed exploration region can be accepted if they are really close. At this point, the weight factor k_{c_i} is extremely important.

A general expression for penalty functions is the following:

$$\mathcal{F}_C(C_i, C_i(\mathcal{P})) = r^2[C_i, C_i(\mathcal{P})] \tag{5}$$

where the function $r[C_i, C_i(\mathcal{P})]$ corresponds to:

$$r(C_i, C_i(\mathcal{P})) = \max\left\{0, \frac{[C_i(\mathcal{P}) - C_i]}{C_i}\right\} \tag{6}$$

4.2 The Kernighan&Lin Heuristic

The original circuit partitioning heuristic was proposed by Kernighan and Lin in 1970 [Kernighan and Lin 1970]. It is based on iterative improvement, but allows the partitioning process to escape from some local minimum. The algorithm starts with an initial random partition and swaps nodes between both sides of the partition. All the nodes are interchanged following the order provided by the best gain (maximum decrease or minimum increment of nets in the cut set). The best solution found during the swap process is recorded and used as the new initial partition in the following iteration. The algorithm finishes when no further improvement is achieved.

Fiduccia and Mattheyses [1982] improved the algorithm performance by means of a sophisticated data structure, the *bucket array*. This structure guarantees linear access time when determining the best movement, taking advantage of the bounded integer gain produced when a node crosses the interface.

The work of Prof. Vahid [1997] in this kind of adaptation is very interesting, but his extension of the algorithm cannot be applied to any other hardware-software partitioning environment, due to the limitations of the model used in the formulation of the system partitioning. This model is defined over an access graph (the SLIF graph [Vahid and Gajski 1995b]), which eases the computation of magnitudes related to the current design (every design quality parameter is computed by adding the different values of the attributes associated with the nodes and edges), but introduces serious limitations:

(1) The design cannot be scheduled, which makes estimating time very rough.
(2) Since any estimation performed in the SLIF graph will be done by adding the node attributes, the trivial solution with everything implemented as hardware will appear in all the optimization procedures. This representation ignores the parallelism inherent to a multiprocessor architecture, whose consideration can provide better results when using a standard processor and a dedicated coprocessor.

The extension of the Kernighan and Lin method to solve the hardware-software partitioning problem must take into account several features:

— The main objective of a hardware-software partitioning tool is much more complex than the simple net cut. Different design aspects must be considered: essentially performance goals and design costs.
— The incremental evaluation of the cost function is not possible because complex measurements must be taken at every step of the algorithm.

The system model presented previously compensates for all these disadvantages allowing the extension of the algorithm to deal with the complex information related to system level design. First, the cost function introduced in Section 4.1.1 can properly handle the system design. Second, a different data structure can improve the algorithm convergence rate. The proposed cost function perfectly characterizes the current design quality, but it is done by means of real numbers. Thus, the integer gain-based bucket-array structure cannot be used for the system partitioning problem.

We have defined a data structure based on *maps* instead of arrays. A map is a sorted associative container that associates objects of type *Key* with objects of a given type. It is also a *unique associative container*, meaning that no two elements ever have the same key. We have used as a key the cost increment associated with a given movement, and as data the node that should be moved. Since more than one node could produce the same cost variation we have used a *multi-map* structure. A multi-map is a sorted associative container that associates objects of type *Key* with objects of a given type. It is also a *multiple associative container*, meaning that there is no limit on the number of elements with the same key.

The proposed data structure is bounded by the number of nodes contained by the system graph. The *standard template library* implementation of these structures [Stroustrup 1997] guarantees that given a *key* the associated data is found through a search with logarithmic complexity.

4.3 Hierarchical Clustering

Another classical heuristic used for circuit and architectural partitioning is hierarchical clustering. This is a constructive method that groups pairs of partitioning objects based on a proximity value between the objects. The algorithm is fully characterized after defining the following issues:

—The closeness function that provides the proximity values.
—The cut level in the cluster tree that is built upon the closeness values.

Both issues will be modified to distribute a given functionality among heterogeneous implementation units [López Vallejo and López 2001]. This is again due to the different nature of the target processing units. It is obvious that a closeness metric used to group objects that will run in a standard processor cannot be suitable for those objects implemented with dedicated hardware.

The proposed closeness function takes into account the system model information. In particular, it uses the estimates associated with the vertices and edges of the system graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in the following way: the function groups those partitioning objects with a clear tendency to be implemented as hardware. Actually, every algorithm iteration selects the two objects with the best time improvement when implemented as hardware (the highest latency decrease). Objects not grouped along the process will be assigned to the software processing unit. The cut level is dynamically computed whenever a cluster is grown taking into account the system constraints. It can be said that this is a software-oriented approach, since all objects are supposed to originally reside in

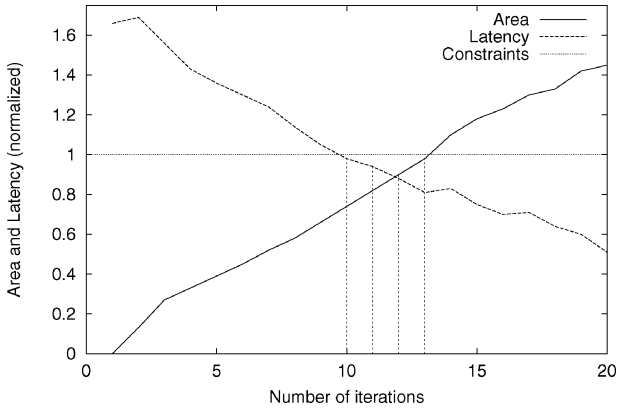

Fig. 2. Clustering process: evolution of the area and latency quality attributes versus their respective constraints.

software and during the process they are extracted to the hardware processing unit.

With this procedure, the whole cluster tree does not need to be built. Every time a cluster is grown, the area and latency constraints (A and T) are checked. The cluster tree construction stops whenever:

(1) The time constraint is met ($T_p < T$) and the hardware area value is below its corresponding constraint ($A_p < A$).
(2) The hardware area constraint is exceeded ($A_p > A$) while the latency constraint is not yet met ($T_p > T$).

In the first case, area and latency constraints are met and the algorithm is almost finished (only the memory constraint must be checked). In the second case, the area constraint is met without fulfilling the timing objectives, and therefore, a refinement phase is necessary. This can be accomplished by some local search procedure working with objects with smaller grain (thus, the last groups must be separated).

Our approach is based on the idea that to find a solution there must be several points in the design space satisfying their respective constraints, as can be seen in Figure 2. This plot shows the evolution of the area and latency quality attributes normalized with respect to the design constraints while building the whole cluster tree in a clustering process. Axis X shows the number of iterations in the clustering process (number of clusters grown). As the number of clusters added to the tree increases, the hardware area becomes larger and the design latency should decrease. When time and area constraints are satisfied, the memory constraint should be checked, although playing a secondary role. If the memory constraint is met, the algorithm finishes. Otherwise the cluster growth continues while other (primary) constraints are still under their limits. Figure 3 illustrates the algorithm control flow described above.

Now we will take a look to the closeness function formulation. As is well known, the exchange of information among the different processing units of the architecture penalizes the design latency. Closeness metrics attempt to

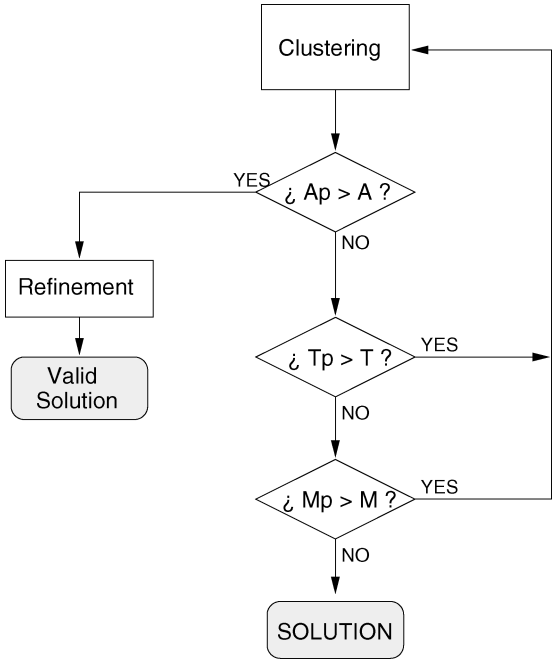

Fig. 3. Control scheme of the clustering process.

reduce this problem. In addition, since the algorithm follows a *"hardware extraction"* approach the time improvement obtained when extracting an object to hardware should also be considered. The expression we have formulated as a closeness function that takes into account both effects is the following:

$$C_{i,j} = q_T \times \left(n_i \frac{\Delta t_i}{st_i} + n_j \frac{\Delta t_j}{st_j} \right) + q_C \times \frac{t_{com}(v_i, v_j)}{t_{r\_com}(v_i) + t_{r\_com}(v_j)} \tag{7}$$

where $\Delta t_i = st_i - ht_i$ represents the time improvement obtained when the object i is moved from software to hardware. The function $t_{comm}(v_i, v_j)$ computes the communication between nodes v_i and v_j in the interface following a general model of the architecture [López Vallejo 1999]. The weight factors q_T and q_C help the designer emphasize which factor he/she wants to optimize.

As can be seen, every function term has been normalized against its software time, in such a way that the resulting closeness value is greater for those objects with bigger difference between hardware and software execution times. The communication term has been normalized with respect to the addition of communication values of the cluster object with other objects of the system graph, its expression being the following:

$$t_{r\_com}(v_i) = \sum_{v_k \in \mathcal{V}} t_{com}(v_i, v_k) / \exists\, (v_i, v_k) \in \mathcal{E} \tag{8}$$

We have used this kind of normalization to highlight the fact that the communication between two design modules must be considered only when there

are many transfers between these modules alone. This term will be therefore less important when communication with the rest of the design modules is considerable.

When checking the plausibility of this closeness function several deficiencies in the behavior of the algorithm were observed. In some cases the algorithm grouped objects with very high communication values and a good time improvement, but with a considerable size. This halted the algorithm early because of the hardware area constraint. For this reason the closeness function was modified to cluster objects that had the previous time and communication considerations but which required little hardware area. The resulting function expression is:

$$C_{i,j} = q_T \times \left(n_i \frac{\Delta t_i}{st_i} + n_j \frac{\Delta t_j}{st_j} \right) + q_C \times \frac{t_{com}(v_i, v_j)}{t_{r\_com}(v_i) + t_{r\_com}(v_j)} + q_A \frac{n_o \times \frac{MaxA}{|\mathcal{V}|}}{ha_i + ha_j} \quad (9)$$

where a hardware area term, controlled by its corresponding weight factor q_A, has been introduced. This term has a clear meaning: its value is greater when the area of the resulting cluster is smaller than the average system area. This average area is computed by dividing the area of the *all-hardware* solution, *MaxA*, by the total number of vertices of the system graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ given by the cardinal of $\mathcal{V}$, $|\mathcal{V}|$. The parameter n_o is the number of nodes integrating the cluster, $n_o = |i| + |j|$. As will be described, each clustering object (single or composed) will be characterized with the same attributes as the nodes of the system model, ha_i, ht_i, and so forth. This formulation allows us to handle objects with different size or various components in a uniform way.

In the same way we can introduce a term for considering the memory space. In this case, since we are extracting modules to hardware, the memory term must try to group objects with large-sized memory. This value is also computed using the parameter *MaxM* provided by the *all-software* solution. The final expression for the closeness function for two objects, i, j, is:

$$\begin{aligned} C_{i,j} \;=\; & q_T \times \left(n_i \frac{\Delta t_i}{st_i} + n_j \frac{\Delta t_j}{st_j} \right) + q_C \times \frac{t_{com}(v_i, v_j)}{t_{r\_com}(v_i) + t_{r\_com}(v_j)} \\ & + q_A \frac{n_o \times \frac{MaxA}{|\mathcal{V}|}}{ha_i + ha_j} + q_M \frac{ss_i + ss_j}{n_o \times \frac{MaxM}{|\mathcal{V}|}} \end{aligned} \quad (10)$$

It is important to remark that the resulting clusters must exhibit the same characteristics as the basic objects, in such a way that no cumulative error is introduced. This characterization allows us to use the same closeness function throughout the algorithm's execution. Consequently, we define the following approximation to characterize a cluster $k = i \cup j$:

—The resulting hardware area is computed by adding up of the hardware area of the nodes that integrate the cluster.

$$ha_k = \sum_{v_n \in p_i} ha_n + \sum_{v_m \in p_j} ha_m \quad (11)$$

This estimation is quite rough, because it does not take into account resource sharing. Nevertheless the approach is completely valid, and we leave for future study the improvement of the area-estimation procedures.

—The cluster memory space is computed as the sum of the memory sizes of its composing vertices.

$$ss_k = \sum_{v_n \in p_i} ss_n + \sum_{v_m \in p_j} ss_m \tag{12}$$

—The cluster execution time in a standard processor is the sum of the software execution times, of its composing nodes, due to the code serialization.[2]

$$st_k = \sum_{v_n \in p_i} st_n + \sum_{v_m \in p_j} st_m \tag{13}$$

—The resulting hardware execution time can be computed in two different ways:

(1) If there are no data dependencies among the cluster vertices, concurrency is possible, and the hardware execution time is given by:

$$ht_k = \max\{ht_n \text{ with } v_n \in p_i \cup p_j\} \tag{14}$$

(2) If there are data dependencies the cluster vertices must be scheduled. Since at every algorithm iteration a system scheduling is performed, we have the starting and finishing times of all the vertices, so the cluster execution time is therefore:

$$ht_k = \max\{t_{end}(v_n)\} - \min\{t_{end}(v_n)\} - t_{idle} \text{ with } v_n \in p_i \cup p_j \tag{15}$$

where t_{idle} is the time the hardware co-processor is idle between $\min\{t_{end}(v_n)\}$ and $\max\{t_{end}(v_n)\}$. This kind of timing evaluation has two clear advantages. First, there is no cumulative error, because ht_k is recalculated at every step of the algorithm. Second, the computational complexity is not greater, since we take advantage of the scheduling performed to evaluate design constraints.

Regarding the cluster edges, we will only consider the edges coming in and out of the cluster, canceling the edges within the cluster vertices. The external edges keep their original attributes, because the cluster will be considered exactly as a new system vertex.

Even though hierarchical clustering has been used for hardware-software partitioning, our approach is very different from these previous implementations. It is mainly due to the formulation of the closeness metric and the modification of the control scheme of the algorithm. In Vahid and Gajski [1995a] different and interesting closeness metrics are defined, but their application is not straightforward and their use is not clearly described. We believe that the

[2]The cluster execution time in the software processor only considers the time spent in this particular execution unit, which is subject to improvement if implemented as standard hardware. It may happen that due to communication delays the finishing time of the cluster is longer than the value estimated here. This is not important in our model, because the communication cost is already considered when the cluster is grown and scheduling is performed. st_k only characterizes an intrinsic parameter of the cluster.

closeness metrics must be different for the hardware and the software partitions. Thus, the clustering procedure needs to be modified and specific metrics for the different implementation units must be defined. Additionally, we have defined a problem model which helped us introducing important modifications to the original algorithm. In Barros et al. [1993], other closeness metrics are introduced, mainly based on the specification language UNITY. Nevertheless, cost issues, as we propose in this work, are not considered.

5. KNOWLEDGE-BASED PARTITIONING

Knowledge-based techniques can be used to solve many optimization problems by working at a higher abstraction level [Newell 1982], because the knowledge acquired by the designers can be modeled and conserved, even if the technology used to develop the partitioning tool becomes obsolete. We have developed a fuzzy logic based expert system, SHAPES (Software-Hardware Partitioning Expert System [López Vallejo et al. 1998]), following the *CommonKADS* methodology [Breuker and Van de Velde 1993]. This tool takes advantage of two important contributions of artificial intelligence: the use of an expert's knowledge in the decision making process and the possibility of dealing with imprecise and usually uncertain values by the definition of fuzzy magnitudes.

The main benefits of a knowledge-based approach are:

— The designer's knowledge is acquired for automatic handling, while the results obtained by traditional procedures must be reviewed by a designer.
— The whole process can be traced.
— Knowledge of the system can be increased by applying it to different cases.

There is also a solid rationale to use fuzzy logic:

— Since the designer's knowledge is imprecise in nature, fuzzy logic can be used to include the subjectivity implicit in the designer's reasoning.
— Most data are estimates, which can be easily modeled with fuzzy linguistic variables. For instance, the area estimate of a block is modeled as a linguistic variable with the terms {`small`, `medium` or `large`} (Figure 4). Since this value depends on the particular system under evaluation, the linguistic variable has been dynamically defined according to a granularity measure, σ:

$$\sigma = \frac{100}{i}$$

i being the number of vertices to consider.
— Some of the parameters defined along the process exhibit intrinsic looseness. For example, Figure 5 shows the implementation value of a task, which reflects the tendency of the task to be implemented as hardware or software. In this case, we can talk for instance about a *quite hardware* or a *very software* block.

The expert system construction has adopted a knowledge-modeling approach, following the knowledge level and knowledge separation principles. The expertise model is, then, the center of the knowledge-based system development.

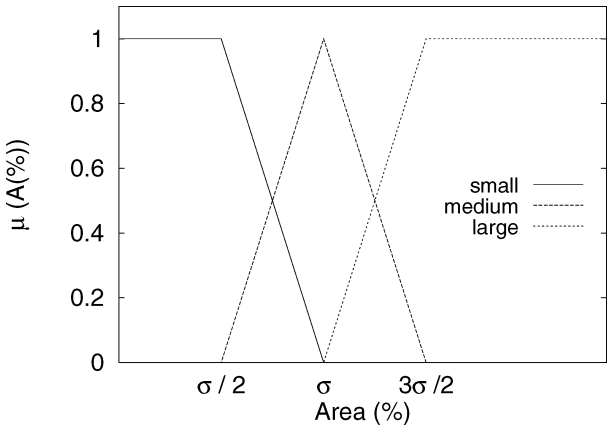

Fig. 4. Linguistic variable *Area*.

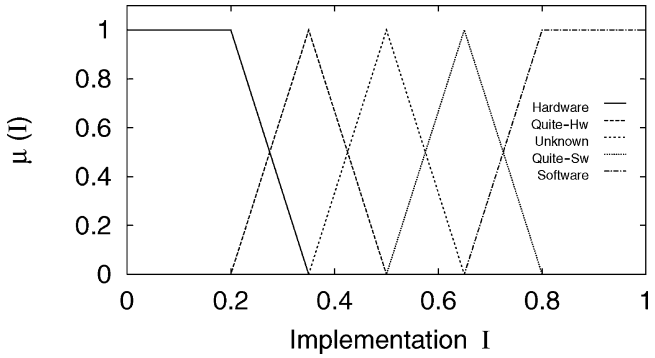

Fig. 5. Linguistic variable *Implementation*.

5.1 Expertise Model of Hardware-Software Partitioning

Three main issues must be addressed when an expertise model is constructed: an ontology of the domain knowledge, the task knowledge and the inference structure. The domain ontology provides the most important concepts and relations of a given domain. The task knowledge specifies how a task is related to the task objective. Actually it describes the way a goal can be achieved by means of some control mechanisms. The inference structure is a compound of predefined inference types (how the domain concepts can be used to make inferences, represented as ellipses) and domain roles (rectangles), as can be seen in Figure 6. For the purpose of this paper, we will only describe the inference structure. We refer readers interested in deeper explanations of the expertise model to [López Vallejo et al. 1999].

The inference structure shown in Figure 6 presents a general view of the problem solving method (PSM) chosen to implement system partitioning and the knowledge used by the inferences. This generic inference structure will be adapted to our problem.

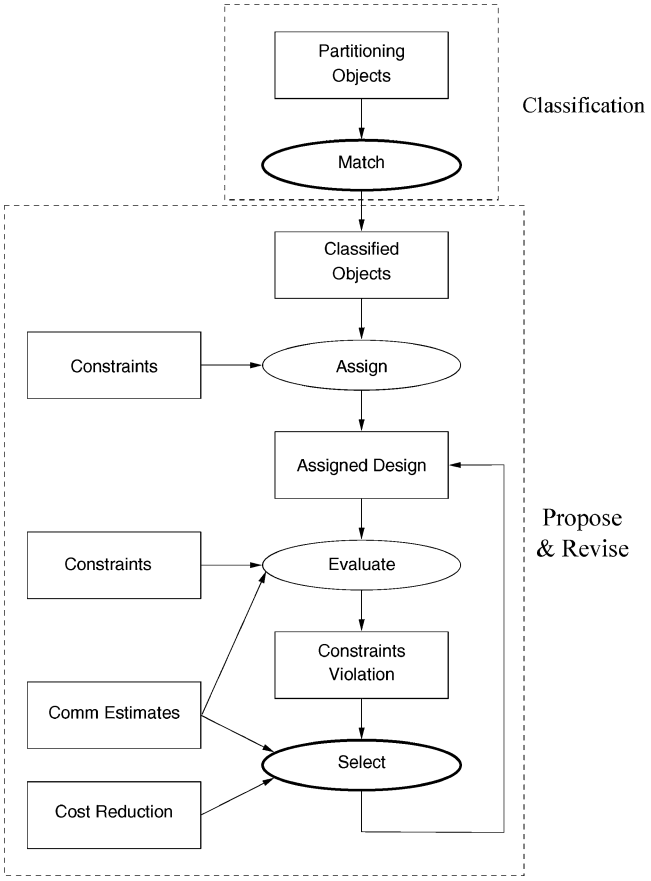

Fig. 6. Inference structure for hardware-software partitioning.

According to the CommonKADS library [Breuker and Van de Velde 1993], the partitioning task has been classified as a configuration task with antagonistic constraints. The *Propose and Revise* (P&R) problem solving method, whose inference structure is shown in Figure 6, has been followed. An initial *heuristic classification* is carried out before the standard P&R for providing additional information to the propose task.

Following the inference structure, four inferences can be distinguished: *match* (Section 5.2.1), *assign* (Section 5.2.2), *evaluate* (Section 5.2.3), and *select* (Section 5.2.4). This inference structure has guided the knowledge-acquisition process.

5.2 Partitioning Design Model

The design of the system has followed a structure-preserving approach, defining a module for each knowledge source of the defined inference structure (Figure 6).

5.2.1 *Match (Heuristic Classification).* The first step in SHAPES is to provide additional information for the P&R method. It is performed by means of a *heuristic classification module*. In this module, the observables are the estimates attached to every vertex properly converted into fuzzy variables. It is important to remark that all the fuzzy sets are dynamically defined, because their values are relative to the specific example under study (see Figure 4).

The heuristic rules stored in the knowledge base match the input variables and the output variable, called `implementation`. The terms this variable can adopt are {`Hardware`, `Quite-Hardware`, `Unknown`, `Quite-Software`, and `Software`}. This variable gives an idea of the intrinsic tendency of a process to be implemented in special purpose hardware or as software running on a standard processor. An example of the rules of this module follows:

```
if hw-area is small and time-improvement is high
   and number-executions are not few
then implementation is very hardware
```

5.2.2 *Assign.* The assign inference provides the first solution proposal by allocating part of the processes to hardware and the rest to software. In this module the observables are the system blocks with their related implementation values and the system constraints (maximum hardware area, A, maximum execution time, T, and maximum memory space, M). The module produces as output a *threshold* that determines the hardware-software boundary.

The *threshold* value is obtained after estimating the *"hardness"* of the specification requirements: how critical (or not) the constraints are, regarding the extreme values of the system performance (all hardware and all software solutions). Upon this parameter, a system configuration is composed.

Every process implementation variable is defuzzified to obtain its crisp value and consequently an initial partition. It is necessary to provide standard output results since they will be used in standard tools like the scheduler. This crisp classification output is the set of input blocks ordered by their implementation degree: value 0 stands for hardware and 1 for software. For instance, Figure 7 shows the results for an example with 23 processes after performing the assignment operation, defuzzifying the threshold and implementation values.

5.2.3 *Evaluate.* The evaluate module computes the different parameters that characterize the design obtained after assignment. These parameters will give an idea of the quality and acceptability of the proposal. This is the only module not based on knowledge. Here four parameters have been considered:

(1) The estimation of the area needed to implement the hardware part, A_p.
(2) The estimation of the memory space required by the software code and data, M_p.
(3) The scheduling of the assigned process graph, which gives the final execution time or latency, T_p.
(4) The communication costs in the hardware-software interface, which consider two different aspects: the total number of transfers that take place in the hardware-software interface, and the *communication penalty*, which

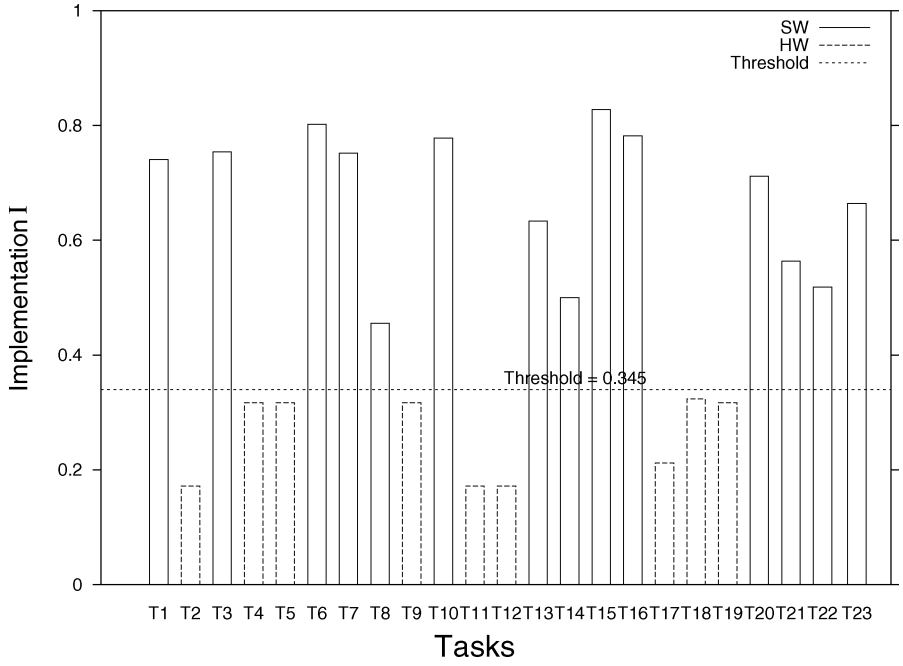

Fig. 7. Classification and assignment results for the case study.

computes the global delay introduced in the system execution time due to specific communication waiting.

Once these parameters have been calculated, the *select* inference will check the plausibility of the proposed partition. With this purpose, additional information is then generated to determine the state of the constraints. Specifically, the following variables are defined:

—The area, memory and time overhead : $\Delta A = A - A_p$, $\Delta M = M - M_p$, $\Delta T = T - T_p$.

—The communication penalty.

—The processor and ASIC throughputs, defined as:

$$\tau_{proc} = \frac{\sum_{i \in SW} st_i}{T_p} \qquad \tau_{asic} = \frac{\sum_{i \in HW} ht_i}{T_p}$$

5.2.4 *Select.* The *select* inference executes the most complex inference in SHAPES. This is a composite inference where two other inferences have been defined. Its purpose is to revise the proposed solution and to search for another proposal. Consequently, the select strategy has been divided into two stages:

(1) *Diagnosis*, which studies how closely the solution comes to the optimum and how many corrections are needed.

(2) *Operation*, which performs the selection of a new proposal based on the previous information and a knowledge base of previous experiences.

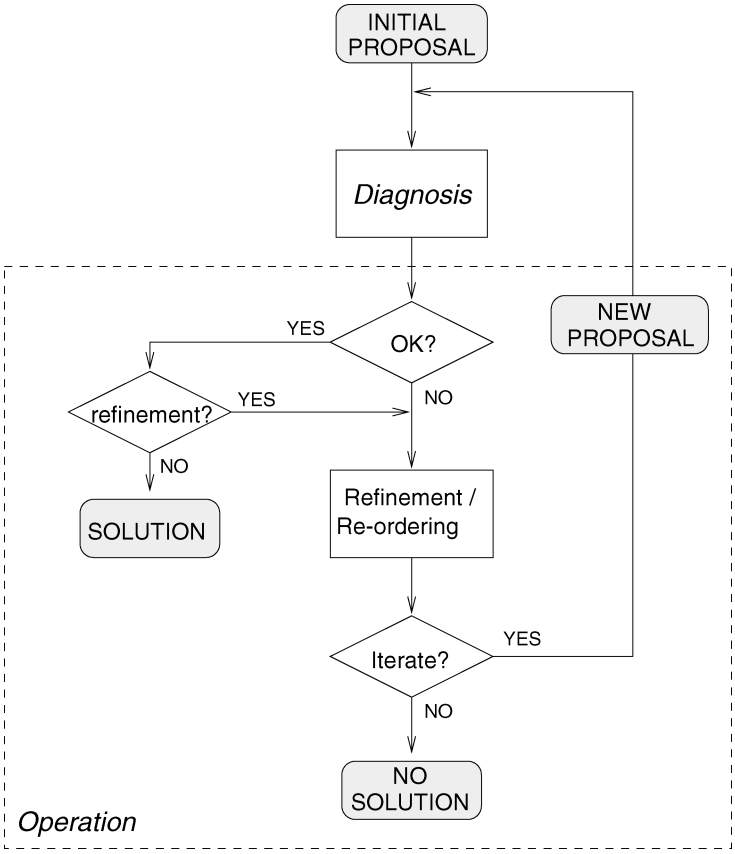

Fig. 8. Control flow within the composed *select* inference.

Figure 8 shows the dependencies among these sub-tasks, specifying the control knowledge within this inference. The diagnosis module performs a simple analysis, identifying the state of the design regarding initial goals and constraints and carrying out the revision procedure of the P&R PSM. In this simple *diagnosis* inference the following observable categories have been considered:

—The violation or satisfaction of the hardware area and global time constraints (upon their respective gaps).
—The balance of the proposal, regarding the processor and ASIC throughputs ($\tau_{proc}/\tau_{asic} \approx 1$).
—Which vertices are related to the communication penalty.

Taking these symptoms into account, two kinds of revisions can be performed. First, if the system constraints are not met, the method must refine the solution to find a feasible one. Second, if constraints are met, but the attribute values are very large, the solution can be refined to reduce costs.

Table II. Set of Examples Provided by Dr. Srinivasan Characterized by the Number of Vertices and Parameters of the Extreme Solutions

Example	Vertices	MaxA	MinT	MaxT	MaxM
lu	9	69315	74	454	19595
fft	15	106355	145	842	33619
dct	9	21952	2231	7312	329692
dct16	36	34944	6712	20680	1143982
laplace	9	73009	79	386	17811
mean	9	132626	99	607	27244
reg	8	3379	1890	6714	303288

The *operation* module performs the proposal of the new partition, if needed. There are two non-exclusive alternative strategies to choose:

—To migrate processes from hardware to software or vice versa according to the global constraints. This is carried out by tuning the threshold.
—To improve the partition while attempting to minimize the communication on the hardware-software boundary. This is performed by communication-based *reordering*, that modifies the classification results according to the communication penalty produced on the interface.

6. ANALYSIS OF RESULTS

All the techniques have been applied to a set of examples provided by Dr. Srinivasan [Srinivasan et al. 1998], whose characterization appears in Table II: the size of the system graph and the values of the *extreme implementations*, which bound the design space. The examples are described as directed and acyclic graphs of coarse-grain tasks. We have applied the four partitioning techniques to every example using different constraints (always within the bounds provided by the extreme solutions). The examples have been subjected to intensive experiments, the results of which are summarized in Table III. These results will be analyzed from both qualitative and quantitative perspectives.[3] The qualitative aspects will be mainly represented by the resulting cost of the solutions obtained from each method, under different constraints. The quantitative issues will be shown by means of the computation time resulting from each technique.

For simulated annealing and K&L the penalty-based cost function (Section 4.1.1) has been used. This formulation has been chosen after checking the behavior of all the proposed cost functions, and proving that this was the best for a general system partitioning problem. Therefore, the quality attributes are always optimized within their respective constraints. The resulting

[3]These results cannot be compared to the ones presented in [Srinivasan et al. 1998] because in this work the partitioning phase integrates hardware design exploration, while we have chosen a given hardware implementation for every task in our experiments.

Table III. Results Obtained with the Examples Provided by the University of Cincinnati

	Constraints			Simulated Annealing				Kernighan&Lin			
Ex.	A	T	M	A_p	T_p	M_p	Cs.	A_p	T_p	M_p	Cs.
LU	15000	400	17000	9110	404	14856	0.588	38501	176	6099	0.806
	30000	200	16000	27311	206	9594	1.177	25604	193	9757	0.656
	50000	150	12000	47611	126	1360	0.624	47611	126	1360	0.849
FFT	30000	400	25000	32464	424	19902	2.674	30530	470	22020	5.986
	60000	250	25000	58092	255	14747	0.824	61619	256	10928	1.035
	80000	200	15000	78242	199	2860	0.611	73337	195	8180	0.772
DCT	16000	4000	100000	13720	2822	73212	0.796	13720	2825	60076	0.529
	13000	5500	200000	5488	5372	164125	0.652	8232	4254	134701	0.489
	20000	3000	120000	13720	2822	73212	0.849	8232	4762	269616	0.538
DCT16	10000	20000	800000	8736	15510	796866	0.594	8736	14450	761912	0.724
	20000	15000	800000	—	—	—	—	17472	9280	364505	0.560
	30000	16000	750000	—	—	—	—	17472	9280	415821	0.404
Laplace	20000	250	16000	16059	251	11232	0.615	20099	260	14894	1.550
	30000	200	15000	29401	179	7280	0.611	31668	229	10023	1.040
	50000	200	12000	38358	137	4200	0.471	14588	320	13345	0.488
Mean	60000	300	20000	55679	294	13938	0.792	55679	294	13938	0.942
	80000	500	10000	76252	313	10164	0.616	100354	239	5740	0.738
	120000	200	10000	116517	127	3124	0.513	116517	127	3124	0.513
Reg	3000	2500	40000	2407	2590	13180	0.779	2113	2380	29029	0.682
	2500	3000	35000	2407	2590	13180	0.585	2407	2590	13180	0.585
	2000	4000	50000	2208	2313	26918	2.181	1435	2030	44043	0.455
	Constraints			Clustering				Expert System			
Ex.	A	T	M	A_p	T_p	M_p	Cs.	A_p	T_p	M_p	Cs.
LU	15000	400	17000	9619	356	16463	0.556	14377	333	13112	0.63
	30000	200	16000	25604	193	9757	0.606	30362	170	6406	0.60
	50000	150	12000	34714	143	5018	0.536	34714	143	5018	0.53
FFT	30000	400	25000	28764	557	18633	23.89	27957	456	23576	1.27
	60000	250	25000	59054	265	7880	1.573	60723	244	12797	0.88
	80000	200	15000	78192	172	3526	0.575	74605	207	8054	0.68
DCT	16000	4000	100000	10976	3675	75013	0.556	13720	3390	38136	0.55
	13000	5500	200000	5488	5372	164125	0.502	10976	3408	85747	0.50
	20000	3000	120000	13720	2825	48779	0.529	13720	2825	48779	0.53
DCT16	10000	20000	800000	4368	16240	776552	0.472	7644	14500	535014	0.53
	20000	15000	800000	7644	14500	628607	0.483	17472	9287	258506	0.52
	30000	16000	750000	5460	15660	721591	0.444	17472	9287	220625	0.49
Laplace	20000	250	16000	17132	274	9013	2.024	16059	251	11232	0.72
	30000	200	15000	23921	213	7631	1.243	29401	179	7280	0.61
	50000	200	12000	37263	141	3679	0.536	43989	115	2814	0.47
Mean	60000	300	20000	43865	386	15438	13.01	55241	292	21751	0.74
	80000	500	10000	76137	260	9698	0.538	72316	259	9898	0.59
	120000	200	10000	103603	176	4676	0.570	105160	135	5022	0.50
Reg	3000	2500	40000	2407	2590	13180	0.779	2407	2590	13180	0.78
	2500	3000	35000	2407	2590	13180	0.585	2167	2863	11285	0.59
	2000	4000	50000	1435	2030	44043	0.455	1873	3354	47005	0.56

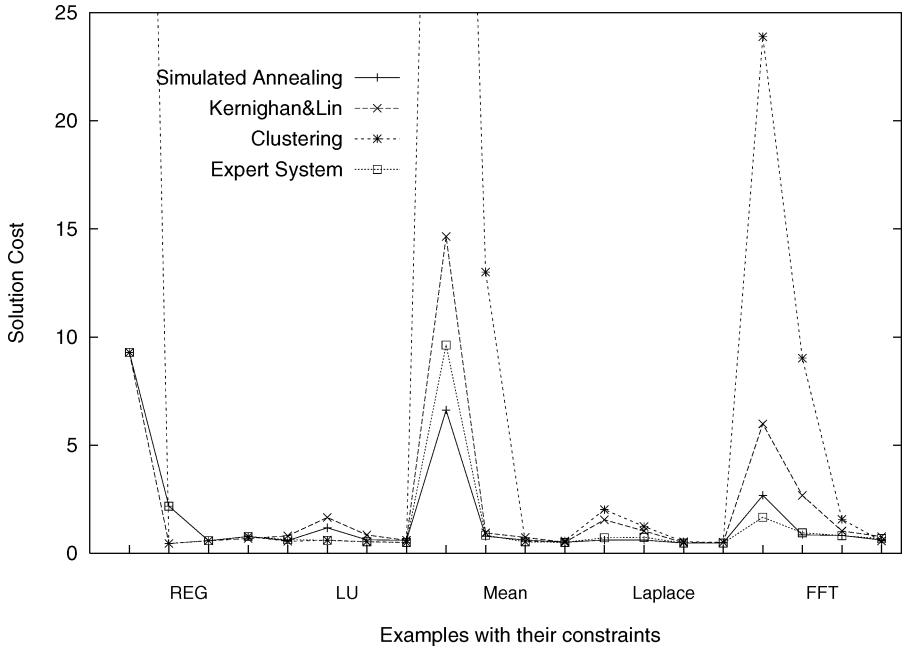

Fig. 9. Cost values obtained when running the Cincinnati University examples.

cost function is:

$$\mathcal{F}(\mathcal{P}) = k_A \times \frac{A_p}{A} + k_T \times \frac{T_p}{T} + k_M \times \frac{M_p}{M} + k_{c_A}\mathcal{F}_C(A, A_p) + k_{c_T}\mathcal{F}_C(T, T_P) + k_{c_M}\mathcal{F}_C(M, M_p) \quad (16)$$

The following weight factors have been applied in all tests: $k_a = 0.3$, $k_t = 0.4$, $k_m = 0.3$ and $k_{c_i} = 150$. These factors allow the designer to put emphasis on the desired design attribute. In these examples, the time goal is slightly emphasized. If these factors are appropriately chosen, they can also help to interpret the results. For instance, when $\sum_i k_i = 1$ (as in our case) then $\mathcal{F}(\mathcal{P}) = 1$ is a figure of merit, since (1) constraint overheads will produce cost values much greater than 1 (due to the penalty weight factor $k_{c_i} = 150$), (2) attribute values tuned to the constraints will produce costs close to the unity, and, (3) if the solution could be optimized (penalty terms are used) the cost value will be lower than 1.

For hierarchical clustering and the expert system, once we have the results provided by these methods (driven by the closeness function or the knowledge bases respectively), we have used the same cost function to characterize the quality of the solutions found and to compare results more easily.

6.1 Solution Quality

In Figure 9 we have represented the cost of the solutions computed by all the methods under evaluation. Each example has been checked with four sets of constraints (defined in Table III) represented in the figure as four points in the axis X. These four sets of constraints have been ordered from harder to softer.

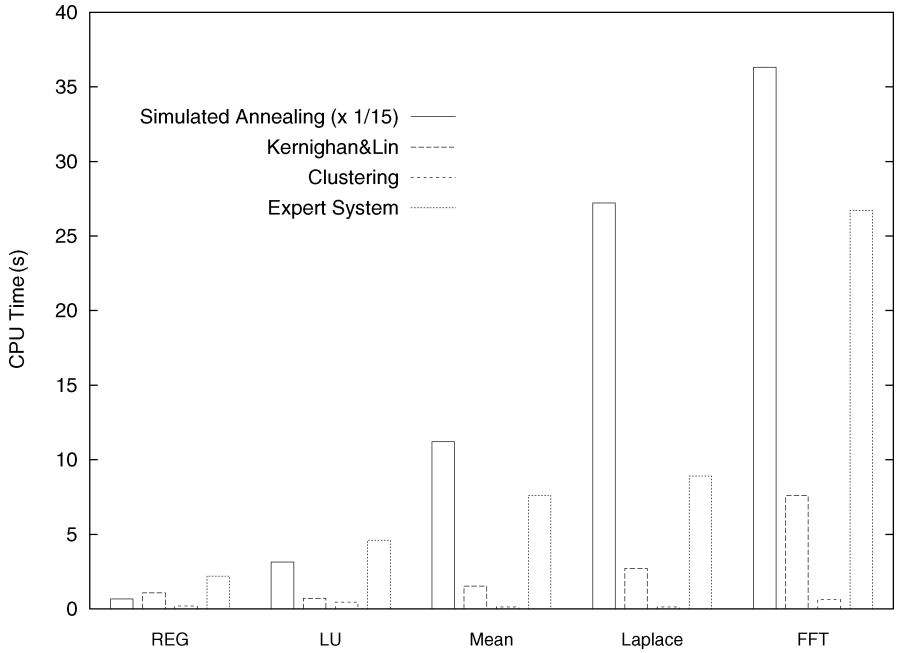

Fig. 10. CPU times (in seconds) for the different partitioning techniques with the Cincinnati University examples.

A first analysis of the results shows that for hard constraints the heuristic algorithms (clustering and K&L) cannot find a valid solution. Even worse, these algorithms provide solutions quite far from the valid region. On the other hand, simulated annealing and the expert system can find solutions for these hard constraints or at least give solutions not far from the allowed exploration region. We can conclude that these two methods provide the best results concerning solution quality. If the constraints are not too hard, all the methods can provide a workable solution.

6.2 Computation Time

The computation time of the different methods under study has been represented in the histogram of Figure 10. Axis Y represents the CPU time spent by the example resolution (simulated annealing time appears divided by 15 for the sake of clarity).

After executing all the examples and examining Figure 10, it is clear that the fastest technique is the clustering algorithm. The reason why this procedure is faster is that it performs the system scheduling very few times (only when a cluster is grown). The other classical algorithms must schedule the design at each trial. Consequently, simulation annealing requires the longest computation time, since the number of explored points of the design space is much more larger than in the other methods.

The expert system expends more time than the K&L and the clustering, but always within reasonable values and much lower than the simulated annealing.

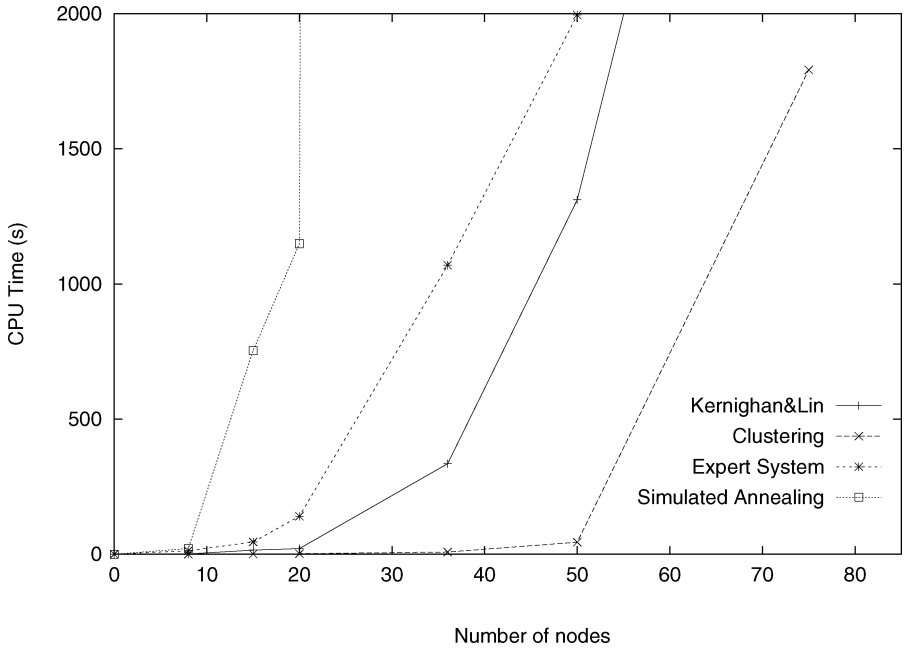

Fig. 11. CPU times (in seconds) for the different partitioning methods vs. the example size.

This longer execution time is partly due to the use of an interpreted inference engine [Lyndon B. Johnson Space Center 1993]. A future study could seek to obtain a faster version of the expert system by compiling its knowledge bases. This has not been done because of the continuous refinement of the system rules.

Finally, to test the behavior of the partitioning techniques when working with bigger examples we have generated several system graphs with a greater number of vertices (from 50 to 500 nodes). Figure 11 represents the execution times obtained after running these examples, ordered by size. These experiments confirmed the results obtained previously.

6.3 Lessons Learned

The experiments we have performed with the various partitioning implementations have shown that this problem can be solved by very different techniques. We can conclude, as a first lesson, that it is very important to build the different implementations over a complete and robust model, which holds all vital information and allows us to deal with different problems.

Simulated annealing has provided the best results from a qualitative point of view, but its computations take a very long time. This occurs even after applying a dynamic cooling schedule, which facilitates the algorithm convergence. On the other hand, the experiments performed with K&L and clustering have shown that these algorithms produce a good trade-off between quality of results and computation time, even though they show an irregular behavior. The K&L algorithm depends too much on the initial solution (here calculated randomly). This problem can be easily solved by starting from a better solution

Table IV. General Characteristics of the Different Techniques

Method	Limit	Granularity	Advantages	Drawbacks	Application
Simulated Annealing	50	coarse	Quality Regularity	Slow	Reference
Expert System	75	coarse	Quality Regularity Information	Interpreted	System Partitioning Design help
Kernighan&Lin	100	do not care	fast	Init. solution dependent	Refinement
Clustering	500	fine	fast	Constraint dependent	Pre-processing

or by running the program several times and taking the best solution, since its computation time is not high.

Hierarchical clustering can be used to perform a fast design-space exploration, especially if a refinement stage is performed later. This method can work with very large system graphs due to its short computation time. Consequently, we recommend its use for fine grain descriptions as a pre-partitioning stage. After this step the design space is reduced and other techniques can work with objects of different granularity. We have obtained excellent results with the clustering technique followed by K&L.

Finally, the tests of the expert system have shown that this is the best system partitioning procedure, both qualitatively and quantitatively. This method has added value since it provides important information to the designer and can be run interactively. This interaction is very easy because of the use of linguistic variables to model the information. For this procedure only coarse-grain graphs can be used, because that is the granularity that specialists employ to model the knowledge.

Table IV summarizes the previous conclusions in a schematic way. This table also shows the maximum number of nodes every technique can deal with and the recommended granularity and applications.

7. CONCLUSIONS

The evaluation of different system implementation possibilities is a key issue in complex system design. We have shown how this problem can be solved by means of very different partitioning techniques. The problem resolution has been based on the definition of a common system model that allows the comparison of different procedures.

The evaluated techniques have been three classical partitioning algorithms and a knowledge based system. The classical circuit partitioning algorithms have been subject to important extensions, since the inclusion of system level issues requires strong modifications in these procedures. These extensions have improved previous implementations, because they include some issues previously not considered. In simulated annealing and K&L system constraints have been integrated into the cost function in a general and efficient way. In the clustering implementation the control scheme of the algorithm has been

modified and new closeness metrics have been defined. Artificial intelligence techniques are very promising to model and work with system level issues as a consequence of the quality of the results and the extra information provided to the user.

The use of these techniques has proven to be effective, as some experiments have shown. Table IV summarizes the conclusions we have drawn after executing and comparing the different system partitioning methods.

A future study could extend the system model to encompass other quality attributes, like power consumption or the degree of parallelism. Also, the final refinement of the knowledge bases of the expert system and their compilation are currently under study.

ACKNOWLEDGMENTS

We would like to thank Carlos Angel Iglesias for his insights into artificial intelligence; Antonio Garcia Quintas for his contribution with the development of the scheduler; and Jesús Grajal for his help in the formulation of the cost function.

REFERENCES

BARROS, E., ROSENSTIEL, W., AND XIONG, X. 1993. HW/SW Partitioning with UNITY. In *Handouts of the 2nd International Workshop on HW-SW Codesign*.

BREUKER, J. A. AND VAN DE VELDE, W., Eds. 1993. *The CommonKADS Library*. Netherlands Energy Research Foundation ECN, Swedish Institute of Computer Science, Siemens, Univ. of Amsterdam and Free University of Brussels. Tech. rep., ESPRIT Project P5248.

CARRERAS, C., LÓPEZ, J. C., LÓPEZ-VALLEJO, M. L., DELGADO-KLOOS, C., MARTÍNEZ, N., AND SÁNCHEZ, L. 1996. A Co-Design Methodology Based on Formal Specification and High-Level Estimation. In *Proceedings of the Workshop on HW/SW Co-Design*.

DICK, R. AND JHA, N. 1998. Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Trans. CAD Int. Circ. Syst. 17,* 10 (October), 920–935.

ELES, P., PENG, Z., KUCHCINSKI, K., AND DOBOLI, A. 1997. System Level Hardware/Software Partitioning based on Simulated Annealing and Tabu Search. *Design Automation for Embedded Systems 2*, 1 (January), 5–32.

ERNST, R., HENKEL, J., AND BENNER, T. 1993. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Des. Test Comput.* 64–75.

FIDUCCIA, C. AND MATTHEYSES, R. 1982. A Linear-time Heuristic for Improving Network Partitions. In *Proceedings of the Design Automation Conference*. IEEE.

GUPTA, R. K. AND MICHELI, G. D. 1993. HW-SW Cosynthesis for Digital Systems. *IEEE Des. Test Comput.* 29–41.

HENKEL, J. AND ERNST, R. 2001. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *Trans. VLSI Syst.* 273–289.

HUANG, M. D., ROMEO, F., AND SANGIOVANI-VINCENTELLI, A. 1986. An Efficient General Cooling Schedule for Simulated Annealing. In *Proceedings of the Design Automation Conference*. 381–384.

KALAVADE, A. AND LEE, E. A. 1997. The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling and Implementation-bin Selection. *J. Design Automat. Embedded Syst. 2*, 2 (March), 125–164.

KERNIGHAN, B. W. AND LIN, S. 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*. 291–307.

KIRPATRICK, S., GELATT, C., AND VECCHI, M. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

LÓPEZ VALLEJO, M., GRAJAL, J., AND LÓPEZ, J. C. 2000. Constraint-driven System Partitioning. In *Proceedings of DATE'00*. 411–416.

LÓPEZ VALLEJO, M., IGLESIAS, C., AND LÓPEZ, J. C. 1998. A Knowledge based System for Hardware-Software Partitioning. In *Proceedings of DATE'98*. 914–915.

LÓPEZ VALLEJO, M. AND LÓPEZ, J. C. 2001. Multi-way Clustering Techniques for System Level Partitioning. In *Proceedings of the 14th IEEE ASIC/SOC Conference*. 242–247.

LÓPEZ VALLEJO, M., LÓPEZ, J. C., AND IGLESIAS, C. 1999. Hardware-Software Partitioning at the Knowledge Level. *J. Applied Intell*. 173–184.

LÓPEZ VALLEJO, M. L. 1999. Hardware-software partitioning methods for the design of heterogeneous systems. Ph.D. thesis, Universidad Politécnica de Madrid.

LUENBERGER, D. G. 1984. *Linnear and non-Linear Programming*. Addison-Wesley.

LYNDON B. JOHNSON SPACE CENTER. 1993. *Clips's Reference Manual. Volume II, Advanced Programming Guide. CLIPS Version 6.0*. Lyndon B. Johnson Space Center, Software Tecnology Branch.

MADSEN, J., GRODE, J., AND KNUDSEN, P. 1997. Hardware/software partitioning using the lycos system. *Hardware/Software Codesign: Principles and Practices* (chapter 9). Kluwer Academic Publishers.

MICHELI, G. D. 1994. Guest editor's introduction: Hardware-Software Codesign. *IEEE Micro*. 8–9.

NEWELL, A. 1982. The knowledge level. *Artificial Intelligence*. 87–127.

NIEMANN, R. AND MARWEDEL, P. 1996. Hardware/software partitioning using integer programming. In *Proceedings of the European Design & Test Conference, 1996*.

SRINIVASAN, V., RADHAKRISHNAN, S., AND VEMURI, R. 1998. Hardware Software Partitioning with Integrated Hardware Design Space Exploration. In *Proceedings of DATE'98*. Paris, France, 28–35.

STROUSTRUP, B. 1997. *The C++ Programming Language. Third edition*. Addison-Wesley.

VAHID, F. 1997. Modifying Min-Cat for Hardware and Software Functional Partitioning. In *Proceedings of the Workshop on HW/SW Co-Design CODES/CASHE'97*. Braunschweig, Germany.

VAHID, F. AND GAJSKI, D. D. 1995a. Clustering for Improved System-level Functional Partitioning. In *Proceedings of ISSS'95*. 28–33.

VAHID, F. AND GAJSKI, D. D. 1995b. SLIF: A Specification-Level Intermediate Format for System Design. In *Proceedings EDAC'95*.

WOLF, W. H. 1997. An Architectural Co-synthesis Algorithm for Distributed Embedded Computing Systems. *IEEE Trans. VLSI Syst. 5*, 2 (June), 218–229.

Received October 2000; revised January 2003; accepted January 2003