Hardware/Software Partitioning with Iterative Improvement Heuristics

Petru Eles[1,2], Zebo Peng[1], Krzysztof Kuchcinski[1], and Alexa Doboli[2]

[1] Dept. of Computer and Information Science
Linköping University
Sweden

[2] Computer Science and Engineering Department
Technical University of Timisoara
Romania

Abstract

The paper presents two heuristics for hardware/software partitioning of system level specifications. The main objective is to achieve performance optimization with a limited hardware and software cost. We consider minimization of communication cost and improvement of the overall parallelism as essential criteria. One of the heuristics is based on simulated annealing and the other on tabu search. Experiments show the superiority of the tabu search based algorithm.

1. Introduction

Satisfaction of performance requirements for embedded systems can frequently be achieved only by hardware implementation of some system components. Selection of the appropriate part of the system for hardware and software implementation respectively has a crucial impact both on the cost and the overall performance of the final product.

Several approaches have been presented in the literature for the partitioning of hardware/software systems. In [6, 8, 13, 16] automatic partitioning is performed, while the approach presented in [1] is based on manual partitioning. Partitioning at a fine grained level is performed in [6, 8]. In [9, 16] partitioning is performed at a coarser granularity.

Iterative improvement algorithms based on neighborhood search are widely used for hardware/software partitioning. In order to avoid being trapped in a local minimum heuristics are implemented which very often are based on simulated annealing [6, 14]. This is mainly due to the fact that simulated annealing algorithms can be quickly implemented and are widely applicable to many different problems.

In [16] a hardware/software partitioning algorithm is proposed which combines a hill climbing heuristic with binary search algorithm. It minimizes hardware cost while satisfying certain performance constraints. This differs from our approach which tries to maximize performance under given cost constraints. The partitioning strategy presented in [9] combines a greedy algorithm with an outer loop algorithm which takes into account global measures. This approach is based on knowledge of execution times for each task and of communication times. This imposes hard restrictions on the features of the system specifications accepted as input. In our approach we do not necessarily impose such limitations, considering more general applications which have to be accelerated by hardware implementation of some components.

Our design environment accepts as input a system level, implementation independent specification of an application. The synthesized system has to produce maximal performance, using a given amount of hardware and software resources. Automatic partitioning at a coarse grain level (process, loop, subprogram, block) is based on metric values derived from profiling, static analysis of the specification, and cost estimations. We consider that minimization of communication cost between the software and the hardware partition and improvement of the overall parallelism are of outstanding importance.

We have implemented first a simulated annealing based algorithm for hardware/software partitioning. We then implemented our partitioning algorithm using the tabu search method. Based on extensive experiments we show that tabu search clearly outperforms simulated annealing.

The paper is divided into 4 sections. Section 2 introduces the partitioning steps, the metric values, and the proposed cost function. In section 3 we discuss our simulated annealing and tabu search based partitioning heuristics, and evaluate their performance. Finally, section 4 presents the conclusions.

2. Partitioning Steps and the Cost Function

The input specification accepted by our co-synthesis environment describes system functionality without prescribing the hardware/software boundary or implementation details. The basic assumption is that this specification is formulated as a set of processes interacting via messages transmitted through communication channels. We also assume that the specification is executable and that profiling information can be generated. The current implementation accepts input designs specified in VHDL [3].

When the final partitioning is done, the hardware implementation is synthesized by the CAMAD high-level synthesis system [15] while the software is generated by a compiler. We have made the following assumptions concerning the target architecture:

1. There is a single microprocessor executing the software part;

2. The microprocessor and the hardware coprocessor are working in parallel;
3. Reducing the amount of communication between the microprocessor and the hardware coprocessor improves the overall performance of the system.

The partitioning algorithm generates as output a model consisting of two sets of processes which are the candidates for hardware and software implementation respectively. The main goal of partitioning is to maximize performance in terms of execution speed. In order to achieve this we try to distribute functionality between the software and the hardware partitions taking also into account communication cost and overall parallelism of the synthesized system. Thus, the following three objectives are considered:

1. To identify *basic regions* (processes, subprograms, loops, and blocks of statements) responsible for most of the execution time, in order to be assigned to hardware;
2. To minimize communication between partitions;
3. To increase parallelism within the resulted system at the following three levels:
   - internal parallelism of each process assigned to hardware;
   - parallelism between processes assigned to hardware;
   - parallelism between the hardware and the microprocessor.

The partitioning algorithm takes into account simulation statistics, information from static analysis of the source specification, and cost estimations. Two types of simulation statistics are used for partitioning:

1. *Computation load* (*CL*) of a basic region is a quantitative measure of the total computation executed by that region, considering all its activations during the simulation process. It is expressed as the total number of operations (at the level of internal representation) executed inside that region, where each operation is weighted with a coefficient depending on its relative complexity [5]. The *relative computation load* (*RCL*) of a block of statements, loop, or a subprogram is the computation load of the respective basic region divided by the computation load of the process the region belongs to. The RCL of a process is the computation load of that process divided by the total computation load of the system.
2. *Communication intensity* (*CI*) on a channel connecting two processes is expressed as the total number of send operations executed on the respective channel.

2. 1. The Partitioning Steps

Hardware/software partitioning is performed in four steps:

1. *Extraction of basic regions*: During the first partitioning step processes are examined individually to identify regions that are responsible for most of the execution time spent inside a process. Candidate regions are typically loops and subprograms, but can also be blocks of statements with a high CL. When a region has been identified for extraction, a new process is built to have the functionality of the original block, loop, or subprogram and communication channels are established to the *parent* process. In [4] we show how extraction of critical regions and process generation is solved in our current implementation.
2. *Process graph generation*.
3. *Partitioning of the process graph*.
4. *Process merging*: During the first step one or several *child* processes are possibly extracted from a *parent* process. If, as result of step 3, some of the child processes are assigned to the same partition with their parent process, they are, optionally, merged back together.

2. 2. The Process Graph

The data structure on which hardware/software partitioning is performed is the *process graph*. Each node in this graph corresponds to a process and an edge connects two nodes if and only if there exists at least one direct communication channel between the corresponding processes.

The graph partitioning algorithm takes into account weights associated to each node and edge. Node weights reflect the degree of suitability for hardware implementation of the corresponding process. Edge weights measure communication and mutual synchronization between processes. The weights capture simulation statistics and information extracted from static analysis of the system specification or of the internal representation resulted after its compilation. The following data extracted from static analysis are captured:

Nr_op_i: total number of operations in process i;

$Nr\_kind\_op_i$: number of different operations in process i;

L_path_i: length of the critical path (in terms of data dependency) through process i.

The weight assigned to process node i, has two components. The first one, $W1_i^N$, is equal to the CL of the respective process. The second one is calculated by the following formula: $W2_i^N = M^{CL} \times K_i^{CL} + M^U \times K_i^U + M^P \times K_i^P - M^{SO} \times K_i^{SO}$; where:

K_i^{CL} is equal to the RCL of process i, and thus is a measure of the computation load;

$K_i^U = \dfrac{Nr\_op_i}{Nr\_kind\_op_i}$; K_i^U is a measure of the uniformity of operations in process i;

$K_i^P = \dfrac{Nr\_op_i}{L\_path_i}$; K_i^P is a measure of the potential parallelism inside process i;

$K_i^{SO} = \dfrac{\sum_{op_j \in SP_i} w_{op_j}}{Nr\_op_i}$; K_i^{SO} captures the suitability of operations of process i for software implementation. SP_i is the set of such operations in process i and w_{op_j} is a weight associated to operation op_j, measuring the degree to which the operation has to be implemented in software.

The relation between the above-named coefficients K^{CL}, K^U, K^P, K^{SO} is regulated by four different weight-multipliers M^{CL}, M^U, M^P, and M^{SO}, controlled by the designer.

Both components of the weight assigned to an edge connecting nodes i and j depend on the amount of communication between processes i and j. The first one is a measure of the total data quantity transferred between the two processes. The second one does not consider the number of bits transferred but only the degree of synchronization between the processes, expressed in the total number of mutual interactions they are involved in:

$$W1_{ij}^{E} = \sum_{c_k \in Ch_{ij}} wd_{c_k} \times CI_{c_k} \; ; \; W2_{ij}^{E} = \sum_{c_k \in Ch_{ij}} CI_{c_k} \; ;$$

where Ch_{ij} is the set of channels used for communication between processes i and j; wd_{c_k} is the width of channel c_k in bits; CI_{c_k} is the communication intensity on channel c_k.

2. 3. Cost Function and Constraints

After generation of the process graph hardware/software partitioning can be performed as a graph partitioning task. The partitioning information, captured as weights associated to the nodes and edges, have to be combined into a cost function which guides the partitioning algorithm towards the desired objective.

Our hardware/software partitioning heuristics are guided by the following cost function which is to be minimized:

$$C(Hw,Sw) = Q1 \times \sum_{(ij) \in cut} W1_{ij}^{E} + Q2 \times \frac{\displaystyle\sum_{(i) \in Hw} \frac{\sum_{\exists (ij)} W2_{ij}^{E}}{W1_{i}^{N}}}{N_H}$$

$$-Q3 \times \left(\frac{\sum_{i \in Hw} W2_{i}^{N}}{N_H} - \frac{\sum_{i \in Sw} W2_{i}^{N}}{N_S} \right) \; ; \; \text{where:}$$

Hw and Sw are sets representing the hardware and the software partition respectively; N_H and N_S are the cardinality of the two sets; *cut* is the set of edges connecting the two partitions; *(ij)* is the edge connecting nodes i and j; *(i)* represents node i.

The partitioning objectives stated at the beginning of section 2 are captured by the three terms of the cost function:

- The *first term* captures the amount of communication between hardware and software partition. Decreasing this component reduces communication cost and also improves parallelism between processes in the hardware partition and those implemented in software.

- The *second term* stimulates placement into hardware of processes which have a reduced amount of interaction with the rest of the system relative to their computation load and, thus, are active most of the time. This strategy improves parallelism between processes inside the hardware partition where physical resources are allocated for real parallel execution. For a given process i, $\left(\sum_{\exists (ij)} W2_{ij}^{E} \right) / W1_{i}^{N}$ is the total amount of interaction the process is involved in, relative to its computation load. The whole term represents the average of this value over the nodes in the hardware partition.

- The *third term* in the cost function pushes processes with a high node weight into the hardware partition and those with a low node weight into the software one, by increasing the difference between the average weight of nodes in the two partitions. This is a basic objective of partitioning as it places time critical regions into hardware.

The criteria combined in the cost function are not orthogonal, and sometimes compete with each other. This competition between partitioning objectives is controlled by the designer through the cost multipliers *Q1*, *Q2*, and *Q3*.

Minimization of the cost function has to be performed in the context of certain constraints. Thus, our heuristics have to produce a partitioning with a minimum for *C(Hw, Sw)* so that the total hardware and software cost is within some specified limits:

$$\sum_{(i) \in Hw} H\_cost_i \le Max^{H} \quad ; \quad \sum_{(i) \in Sw} S\_cost_i \le Max^{S} \quad .$$

Cost estimation has to be performed before graph partitioning. In the current implementation of our environment, the CAMAD high level synthesis system [15] produces hardware cost estimations in terms of design area. Software cost, in terms of memory size, is estimated for each process through compilation by our VHDL to C compiler.

3. Process Graph Partitioning

Hardware/software partitioning, formulated as a graph partitioning problem, is NP-complete. In order to efficiently explore the solution space, heuristics have to be developed which hopefully converge towards an optimal or near-optimal solution. We have implemented two such algorithms, one based on simulated annealing (SA) and the other on tabu search (TS).

For evaluation of the partitioning algorithms we used random and geometric graphs [17] generated for experimental purpose, and graphs resulted from compilation of real-life examples. We generated for experiments 32 graphs altogether, 16 random and 16 geometric. 8 graphs (4 random, 4 geometric) have been generated for each dimension of 20, 40, 100, and 400 nodes. The generation of these graphs and their characteristics are presented in [5]. Experiments have been carried out in order to tune the algorithms for each graph dimension so that partitioning converges with a high probability towards an optimum *for all test graphs of the given dimension* and the run time is minimized.

It still has to be clarified what we call an *optimum* in this context. For the 20 node graphs it was possible to run exhaustive search to get the *real* optimum which we later used as a reference value. For each of the other graphs we performed, in preparation of the experiments, very long and expensive runs using both SA and TS. We used aggressively very long cooling schedules, for SA, and a high number of restarting tours, for TS (see sections 3.1 and 3.2). These runs have been performed starting with different initial configurations and finally the *best ever* solution produced for each graph has been considered as the *optimum* for the further experiments.

During experiments with SA an additional difficulty originates from the random nature of this algorithm. The same implementation with unchanged parameters can pro-

Step 1. Construct initial configuration x^{now}:=(Hw_0, Sw_0)
Step 2. Initialize Temperature T:=TI
Step 3. 3.1. *for* i:=1 *to TL do*
Generate randomly a neighboring solution $x' \in N(x^{now})$
Compute change of cost function $\Delta C := C(x') - C(x^{now})$
if $\Delta C \leq 0$ *then* $x^{now} := x'$
else
Generate q:=*random*(0,1)
if $q < e^{-\Delta C/T}$ *then* $x^{now} := x'$
3.2. Set new temperature $T := \alpha * T$
Step 4. *if* stopping criterium not met *then goto* Step 3
Step 5. *return* solution corresponding to the minimum cost function

Fig. 1. Simulated annealing algorithm

duce different results, for the same graph, in different runs. We considered that a certain configuration of parameters produces an optimum for a graph if for 100 consecutive runs of the SA algorithm we got each time the optimal partitioning.

All experiments presented were run on SPARCstation 10.

3. 1. Partitioning with Simulated Annealing

Simulated annealing selects a neighboring solution randomly and always accepts an improved solution. It also accepts worse solutions with a certain probability that depends on the deterioration of the cost function and on a control parameter called temperature [11]. In Fig. 1 we give a short description of the algorithm. With x we denote one solution consisting of the two sets Hw and Sw. x^{now} represents the current solution and $N(x^{now})$ denotes the neighborhood of x^{now} in the solution space.

For implementation of this algorithm the parameters TI (initial temperature), TL (temperature length), α (cooling ratio), and the stopping criterium have to be determined. They define the so called cooling schedule and have a decisive impact on the quality of partitioning and the CPU time consumed. As result of our experiments we determined for each graph dimension values for TI, TL, and α so that an optimal partitioning for each graph with the respective number of nodes is produced [5]. The algorithm terminates when for three consecutive temperatures no new solution has been accepted.

For the generation of a new solution x', starting from the current one x^{now}, we implemented two strategies: the *simple move* (SM) and the *improved move* (IM).

For the *SM* a node is randomly selected and moved to the other partition. The configuration resulted after this move becomes the candidate solution x'. Random node selection is repeated if transfer of the selected node violates some design constraints.

The *IM* accelerates convergence by moving, together with the randomly selected node, also some of its direct neighbors (nodes which are in the same partition with the selected one and are directly connected to it). A direct neighbor is moved together with the selected node if this movement improves the cost function and does not violate any constraint. This strategy stimulates transfer of connected node groups instead of individual nodes. Experiments revealed a negative side effect of this strategy: the repeated move of the same or similar node groups from

TABLE 1: Partitioning time with SA

nr. of nodes	CPU time (s) SM	CPU time (s) IM	speedup
20	0.28	0.23	22%
40	1.57	1.27	24%
100	7.88	2.33	238%
400	4036	769	425%

TABLE 2: Parameters and CPU time with TS

nr. of nodes	τ	Nr_f_b	Nr_r	CPU time (s)
20	7	30	0	0.008
40	7	50	0	0.04
100	7	50	0	0.19
400	18	850	2	30.5

one partition to the other, which resulted in a reduction of the spectrum of visited solutions. To produce an optimal exploration of the solution space we combined movement of node groups with that of individual nodes: nodes are moved in groups with a certain probability p. After analysis of experimental results the value for p was fixed at 0.75.

Partitioning times and the speedup produced by the improved strategy are presented in Table 1. The times shown are the average CPU time needed for optimal partitioning for all graphs of the given dimension.

3. 2. Partitioning with Tabu Search

By contrast to simulated annealing, tabu search controls uphill moves not purely randomly but in an intelligent way [7]. Two key elements of the TS algorithm are the data structures called short and long term memory. Short term memory stores information relative to the most recent history of the search. It is used in order to avoid cycling that could occur if a certain move returns to a recently visited solution. Long term memory, on the other side, stores information on the global evolution of the algorithm. These are typically frequency measures relative to the occurrence of a certain event. They can be applied to perform *diversification* which is used to improve exploration of the solution space by broadening the spectrum of visited solutions.

In Fig. 2 we give a brief description of our implementation of the TS algorithm. In the first attempt an improving move is tried. If no such move exists (or it is tabu and not aspirated) frequency based penalties are applied to the cost function and the best possible non tabu move is performed; this move can be an uphill step. Finally, in a last attempt, the move which is closest to leave the tabu state is executed.

We consider as a candidate solution x_k the configuration obtained from x^{now} by moving node k from its current partition to the other one, if this movement does not violate any constraints. In the *tabu list* we store the list of the reverse moves of the last τ moves performed, which are considered as being forbidden (tabu). The size τ of this list (the *tabu tenure*) is an essential parameter of the algorithm. In Table 2 we present the optimal values for τ as resulted from our experiments.

Under certain circumstances it can be useful to ignore the tabu character of a move (the tabu is *aspirated*). We ignore the tabu status of a move if the solution produced is better than the best obtained so far.

Step 1. Construct initial configuration x^{now}:=(Hw_0, Sw_0)
Step 2. *for* each solution $x_k \in N(x^{now})$ *do*
Compute change of cost function $\Delta C_k := C(x_k) - C(x^{now})$
Step 3. 3.1. *for* each $\Delta C_k < 0$, in increasing order of ΔC_k *do*
if not *tabu*(x_k) or *tabu_aspirated*(x_k) *then*
$x^{now} := x_k$
goto Step 4
3.2. *for* each solution $x_k \in N(x^{now})$ *do* Compute $\Delta C'_k := \Delta C_k + penalty(x_k)$
3.3. *for* each $\Delta C'_k$ in increasing order of $\Delta C'_k$ *do*
if not *tabu*(x_k) *then*
$x^{now} := x_k$
goto Step 4
3.4. Generate x^{now} by performing the least tabu move
Step 4. 4.1. *if* iterations since previous best solution < *Nr_f_b then goto* Step 2
4.2. *if* restarts < *Nr_r then*
Generate initial configuration x^{now} considering frequencies
goto Step 2
Step 5. *return* solution corresponding to the minimum cost function

Fig. 2. Tabu search algorithm

Three means of improving the search strategy by diversification have been implemented:

1. For the second attempt to generate a new configuration, moves are ordered according to a penalized cost function which favors the transfer of nodes that have spent a long time in their current partition:

$$\Delta C'_k = \Delta C_k + \frac{\sum_i |\Delta C_i|}{Nr\_of\_nodes} \times pen(k) \quad ; \quad \text{where}$$

$$pen(k) = \begin{cases} -C_H \times \dfrac{Node\_in\_Hw_k}{N_{iter}} & \textit{if node } k \in Hw \\ -C_S \times \left(1 - \dfrac{Node\_in\_Hw_k}{N_{iter}}\right) & \textit{if node } k \in Sw \end{cases}$$

Node_in_Hw$_k$ is the number of iterations node k spent in the hardware partition; N_{iter} is the total number of iterations; *Nr_of_nodes* is the total number of nodes; Coefficients have been experimentally set to C_H=0.4 and C_S=0.15.

2. We consider a move as forbidden (tabu) if the frequency of the node in its current partition is smaller than a certain threshold; thus, a move of node k can be accepted if:

$$\frac{Node\_in\_Hw_k}{N_{iter}} > T_H \qquad \textit{if node } k \in Hw$$

$$\left(1 - \frac{Node\_in\_Hw_k}{N_{iter}}\right) > T_S \qquad \textit{if node } k \in Sw$$

Thresholds have been experimentally set to T_H=0.2, T_S=0.4.

3. If the system is frozen (more than *Nr_f_b* iterations have passed since the current best solution was found) a new search can be started from an initial configuration which is different from those encountered previously.

The number of iterations performed for partitioning is influenced by parameters *Nr_f_b* (number of iterations without improvement of the solution after which the system is considered frozen) and *Nr_r* (number of restarts with a new initial configuration). The minimal values needed for an optimal partitioning of all graphs of the respective dimension and the resulted CPU times are presented in Table 2. The times have been computed as the average of the partitioning time for all graphs of the given dimension.

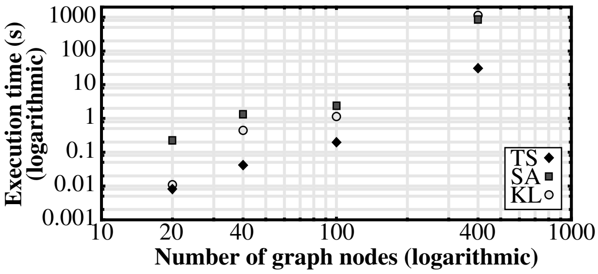

Fig. 3. Partitioning times with SA, TS, and KL

3. 3. Evaluation of the SA and TS approaches

The experiments presented in the previous sections lead to the following main conclusions:

1. Near-optimal partitioning can be produced both by the SA and TS based algorithm.
2. SA is based on a random exploration of the neighborhood while TS is completely deterministic. The deterministic nature of TS makes experimental tuning of the algorithm and setting of the parameters less laborious than for SA. At the same time adaptation of the SA strategy for a particular problem is relatively easy and can be performed without a deep study of domain specific aspects. Although, problem specific improvements can result, as we have shown, in large gains of performance. On the contrary, development of a TS algorithm is more complex and has to consider particular aspects of the given problem.
3. Performances obtained with TS are definitely superior in comparison to those given by SA, as shown in Fig. 3 (for SA the execution time with IM is represented). This conclusion is very important especially in the context that, to our knowledge, no TS based hardware/software partitioning approach has yet been reported, while SA continues to be one of the most popular approaches for automatic partitioning.

Finally, we compared our SA and TS-based heuristics with a classical iterative-improvement approach, the Kernighan-Lin (KL) algorithm [10]. Given the relatively limited capacity of the KL-based algorithm to escape from local minima and its sensitivity to the initial configuration, we had to perform several runs for each graph, with randomly determined starting configurations. The number of necessary restarting tours has been fixed so that all graphs of a given dimension are optimally partitioned with a sufficiently high probability (for 100 consecutive runs we got each time the optimal[1] partitioning). As shown in Fig. 3, partitioning times with KL are slightly better than those with SA for small and medium graphs. For the 400 nodes graphs SA already outperforms the KL-based algorithm. TS is on average 10 times faster than KL for 40 and 100 nodes graphs, and 30 times faster for graphs with 400 nodes.

In order to validate our system level partitioning approach we performed two further experiments on real-life

1. We use "optimal" in the sense introduced at the beginning of section 3.

TABLE 3: Partitioning of the VHDL models

model	nr. of processes		part. with SA	part. with TS	t_{TS}/t_{SA}
	model	after extr.	t_{SA} (sec)	t_{TS} (sec)	
Eth. cop.	10	20	0.08	0.006	0.075
OAM bl.	19	27	0.10	0.007	0.07

models: the *Ethernet network coprocessor* and the *OAM block of an ATM switch*. Both models were specified at system level in VHDL. Partitioning was performed using both the SA based and the TS algorithm, with the cost function presented in section 2.3 and a constraint on the hardware cost representing 30% of the cost of a pure hardware implementation.

The *Ethernet network coprocessor* is given in [12] as an example for system specification in SpecCharts and has been used, in a HardwareC version, in [8]. We have rewritten it in VHDL, as a model consisting of 10 cooperating processes (730 lines of code). After the first partitioning step, extraction of performance critical loops and subprograms, we got a VHDL specification consisting of 20 processes. Process graph generation and partitioning produced a hardware partition with 14 processes and a software partition with 6 processes. The most time critical part of those processes that are handling transmission and reception of data on the ethernet line as well as processes which are strongly connected to them have been assigned to hardware and the rest belong to the software partition.

Our second example implements the *operation and maintenance (OAM) functions corresponding to the F4 level of the ATM protocol layer* [2]. We specified functionality as a VHDL model consisting of 19 interacting processes (1321 lines of code). The model resulted after extraction of basic regions has 27 processes. The resulted process graph has been partitioned into 14 processes assigned to hardware and 13 to software. Processes performing the filtering of input cells and those handling user cells (which constitute the majority of received cells) were assigned to hardware. Processes handling exclusively OAM cells (which are arriving at a very low rate) were assigned to software.

In Table 3 we show the partitioning times using SA and TS for both examples which confirm the conclusions drawn from experiments with geometric and random graphs.

4. Conclusion

We have presented an approach to automatic hardware/software partitioning of system level specifications. Partitioning is performed at the granularity level of blocks, loops, subprograms, and processes and produces an implementation with maximal performance using a limited amount of hardware and software resources. Partitioning is based on metric values derived from simulation, static analysis of the specification, and cost estimations. A cost function that combines these metrics and guides partitioning towards the desired objective has been developed.

We formulated hardware/software partitioning as a graph partitioning problem and solved it by implementing iterative improvement heuristics based on simulated annealing and tabu search respectively. We have demonstrated that both algorithms can produce high quality solutions. We have also shown that performances obtained with TS are superior in comparison to those given by even improved implementations of SA, or by classical algorithms like KL.

The algorithms we presented can be used also for partitioning purposes other than system level hardware/software partitioning. They can be, for instance, equally useful, and can be easily extended, for partitioning at finer levels of granularity.

References

[1] P.H. Chou, R.B. Ortega, G. Boriello, *The Chinook Hardware/Software Co-Synthesis System*, Proc. Int. Symp. on Syst. Synth., September '95, 22-27.

[2] M. De Prycker, *Asynchronous Transfer Mode: Solution for Broadband ISDN*, Ellis Horwood, New York, 1993.

[3] P. Eles, K. Kuchcinski, Z. Peng, M. Minea, *Synthesis of VHDL Concurrent Processes*, Proc. of EURO-DAC/VHDL'94, 1994, 540-545.

[4] P. Eles, Z. Peng, A. Doboli, *VHDL System-Level Specification and Partitioning in a Hardware/software Co-Synthesis Environment*, Proc. of Third International Workshop on Hardware/Software Codesign, 1994, 49-55.

[5] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, *Performance Guided System Level Hardware/Software Partitioning with Iterative Improvement Heuristics*, Res. Rep. LiTH-IDA-R-95-26, Dep. of Comp. and Inf. Science, Linköping University, 1995.

[6] R. Ernst, J. Henkel, T. Benner, *Hardware-Software Co-Synthesis for Microcontrollers*, IEEE Design & Test of Computers, September 1993, 64-75.

[7] Glover, E. Taillard, D. de Werra, *A User's Guide to Tabu Search*, Annals of Operations Research, vol. 41, 1993, 3-28.

[8] R.K. Gupta, G. De Micheli, *Hardware-Software Cosynthesis for Digital Systems*, IEEE Design & Test of Computers, September 1993, 29-41.

[9] A. Kalavade, E.A. Lee, *A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem*, Proc. of Third International Workshop on Hardware/Software Codesign, 1994, 42-48.

[10] B.W. Kernighan, S. Lin, *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Systems Tech. J. vol. 49, no. 2, 1970, 291-307.

[11] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, *Optimization by simulated annealing*, Science, vol. 220, no. 4598, 1983, 671-680.

[12] S. Narayan, F. Vahid, D.D. Gajski, *Modeling with SpecCharts*, Technical Report #90-20, Dept. of Inf. and Comp. Science, Univ. of California, Irvine, 1990/1992.

[13] R. Niemann, P. Marwedel, *Hardware/Software Partitioning using Integer Programming*, Proc. of ED&TC'96, 1996, 473-479.

[14] Z. Peng, K. Kuchcinski, A*n Algorithm for Partitioning of Application Specific Systems*, Proc. EDAC'93, 1993, 316-321.

[15] Z. Peng, K. Kuchcinski, *Automated Transformation of Algorithms into Register-Transfer Level Implementation*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 2, February 1994, 150-166.

[16] F. Vahid, J. Gong, D. Gajski, *A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware/software Partitioning*, Proc. of EURO-DAC/VHDL'94, 1994, 214-219.

[17] C.W. Yeh, C.K. Cheng, T.T.Y. Lin, *Optimization by Iterative Improvement: An Experimental Evaluation on Two-Way Partitioning*, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 2, February 1995, 145-153.