Hardware-software partitioning in embedded system design

Péter Arató, Sándor Juhász, Zoltán Ádám Mann, András Orbán, Dávid Papp
Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
arato@iit.bme.hu, jsanyi@mail.matav.hu, {zoltan.mann, andras.orban, pappd}@cs.bme.hu

***Abstract* – One of the most crucial steps in the design of embedded systems is hardware-software partitioning, i.e. deciding which components of the system should be implemented in hardware and which ones in software. In this paper, different versions of the partitioning problem are defined, corresponding to real-time systems, and cost-constrained systems, respectively. The authors provide a formal mathematic analysis of the complexity of the problems: it is proven that they are $\mathcal{NP}$-hard in the general case, and some efficiently solvable special cases are also presented. An ILP (integer linear programming) based approach is presented that can solve the problem optimally even for quite big systems, and a genetic algorithm (GA) that finds near-optimal solutions for even larger systems. A specialty of the GA is that non-valid individuals are also allowed, but punished by the fitness function.**

***Keywords* – genetic algorithm, graph partitioning, hardware/software codesign, hardware/software partitioning, integer linear programming**

I. INTRODUCTION

Today's computer systems typically consist of both hardware and software components. For instance in an embedded signal processing application it is common to use both application-specific hardware accelerator circuits and general-purpose, programmable units with the appropriate software [1], [2].

This is beneficial since application-specific hardware is usually much faster than software, but it is also significantly more expensive. Software on the other hand is cheaper to create and to maintain, but slow. Hence, performance-critical components of the system should be realized in hardware, and non-critical components in software. This way, a good trade-off between cost and performance can be achieved.

However, this kind of system design is not without challenges. Usual hardware and software design methodologies are in many aspects inadequate for such design tasks. The composition of hardware and software elements also creates new problems, *e.g.* related to the communication of hardware and software components, as well as system architecture issues. In order to address these problems, *hardware-software co-design* (HSCD) methods have to be used [3].

One of the most crucial design steps in HSCD is partitioning, *i.e.* deciding which components of the system should be realized in hardware and which ones in software. Clearly, this is the step in which the above-mentioned optimal trade-off between cost and performance is to be found. Therefore, partitioning has dramatic impact on the cost and performance of the whole system [4].

Traditionally, partitioning was carried out manually. However, as the systems to design have become more and more complex, this method has become infeasible, and many research efforts have been undertaken to automate partitioning as much as possible.

II. PREVIOUS WORK

One of the most relevant works is presented in [5], [6]: a very sophisticated integer linear programming model for the joint partitioning and scheduling problem for a wide range of target architectures. This integer program is part of a 2-phase heuristic optimization scheme which aims at gaining better and better timing estimates using repeated scheduling phases, and using the estimates in the partitioning phases.

[7] presents a method for allocation of hardware resources for optimal partitioning. During the allocation algorithm, an estimated hardware/software partition is also built. The algorithm for this is basically a greedy algorithm: it takes the components one by one, and allocates the most critical building block of the current component to hardware. If the finite automaton realizing the control of the component also fits into hardware, then that component is moved to hardware.

An even more heuristic approach is described in [8]. This paper deals with a partitioning problem, in which even the cost function to be optimized is a very complicated, heuristically weighted sum, which tries to take into account several optimization criteria. The paper also describes two

This work was supported by OTKA T043329.

heuristic algorithms: one based on simulated annealing and one based on tabu search.

[9] shows an algorithm to solve the joint problem of partitioning and scheduling. It consists of basically two local search heuristics: one for partitioning and one for scheduling. The two algorithms operate on the same graph, at the same time.

[10] considers partitioning in the design of ASIPs (application-specific integrated processors). It presents a formal framework and proposes a partitioning algorithm based on branch and bound.

[11] presents an approach that is largely orthogonal to other partitioning methods: it deals with the problem of hierarchically matching tasks to resources. It also shows a method for weighting partially defined user preferences, which can be very useful for multiple-objective optimization problems.

III. PROBLEM DEFINITION

We observed that previous works tried to capture too many details of the partitioning problem and the target architecture. Therefore, the solutions proposed in the literature

- do not scale well for large inputs but they fall victim to combinatorial explosion; and/or
- are heuristic in nature, and drift away too much from optimal solutions.

Moreover, the complexity of the problem is hard to determine.

Therefore, our main goal was to introduce a simplified model for partitioning, using which we can

- define different versions of the partitioning problem formally;
- analyze the complexity of the problems formally;
- give algorithms that can be used for the design of large systems as well.

Note that algorithms published previously in the literature have been tested on systems with some dozens of components. Our aim was to make our algorithms scalable for systems with hundreds or even thousands of components, so that they can indeed be used in practical, large-scale projects. For these reasons we had to keep the model as simple as possible, only taking into account the most important properties of the partitioning problem.

A. Informal model

The characteristics of our model are the following:

- We consider only one software context (*i.e.* one general-purpose processor) and one hardware context (*e.g.* one FPGA). The components of the system have to be mapped to either one of these two contexts.
- Software implementation of a component is associated with a software cost, which is the running time of the component if implemented in software.
- Hardware implementation of a component is associated with a hardware cost, which can be for instance area, heat dissipation, energy consumption etc.
- Since hardware is significantly faster than software, the running time of components implemented in hardware is taken to be zero.
- If two communicating components are mapped to different contexts, this is associated with a (time dimensional) communication overhead. If two components are mapped to the same context, then the communication between them is neglected.

One of the most important advantages of this simplified model is that scheduling does not have to be addressed explicitly. Hardware components do not have to be scheduled, because their running time is assumed to be zero. Software components do not have to be scheduled because there is only one processor, so that the overall running time will be the sum of the running times of the software components, regardless of their schedule. Therefore, we can completely decouple the partitioning problem from the scheduling problem, and focus solely on the partitioning problem.

B. Formal problem definition

An undirected simple graph $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $s, h : V \to \mathbb{R}^+$ and $c : E \to \mathbb{R}^+$ are given. G is the so-called task graph of the system, its nodes are the components of the system that have to be partitioned, and the edges represent communication between the components. $s(v_i)$ (or s_i) and $h(v_i)$ (or h_i) denote the software and hardware cost of node v_i, respectively, while $c(v_i, v_j)$ denotes the communication cost between v_i and v_j if they are in different contexts (HW or SW).

P is called a hardware-software (HW-SW) partition if it is a bipartition of V: $P = (V_H, V_S)$, where $V_H \cup V_S = V$ and $V_H \cap V_S = \emptyset$. The crossing edges are: $E_P = \{(v_i, v_j) : v_i \in V_S, v_j \in V_H \text{ or } v_i \in V_H, v_j \in V_S\}$. The hardware cost of P is: $H_P = \sum_{v_i \in V_H} h_i$; the software cost of P is: $S_P = \sum_{v_i \in V_S} s_i + \sum_{(v_i, v_j) \in E_P} c(v_i, v_j)$, *i.e.* the software cost of the nodes and the communication cost; since both are time-dimensional, it makes sense to add them, and together they make up the running time of the system. The following optimization and decision problems can be defined (G, h, s, c are given in all problems):

- PART1: H_0, $S_0 \in \mathbb{R}^+$ are given. Is there a P HW-SW partition so that $H_P \leq H_0$ and $S_P \leq S_0$?
- PART2: $H_0 \in \mathbb{R}^+$ is given. Find a P HW-SW partition so that $H_P \leq H_0$ and S_P is minimal. (Cost-constrained systems)
- PART3: $S_0 \in \mathbb{R}^+$ is given. Find a P HW-SW partition so that $S_P \leq S_0$ and H_P is minimal. (Systems with hard real-time constraints)

C. Complexity

Theorem 1. PART1 *is $\mathcal{NP}$-complete even if only graphs with no edges are considered.*

Proof. PART1$\in \mathcal{NP}$, since if partitioning is possible with the given limits, then the partition itself is a good proof for this.

To prove the $\mathcal{NP}$-hardness, we reduce the KNAPSACK problem [12] to PART1. Let an instance of the KNAPSACK problem be given. (There are n objects, the weights of the objects are denoted by w_i, the price of the objects by p_i, the weight limit by W and the price limit by K. The task is to decide whether there is a subset X of objects, so that $\sum_{i\in X} w_i \leq W$ and $\sum_{i\in X} p_i \geq K$.) Based on this, we define an instance of PART1 as follows: $V = \{v_1, \ldots, v_n\}$, $E = \emptyset$. Let $h_i = p_i$, $s_i = w_i$. (Since E is empty, there is no need to define c.) Introducing $A = \sum_{v_i \in V} p_i$, let $S_0 = W$, $H_0 = A - K$.

We state that this instance of PART1 is solvable iff the original KNAPSACK problem has a solution.

Assuming that PART1 has a solution: $P = (V_H, V_S)$, where $V_H \cup V_S = V$ and $V_H \cap V_S = \emptyset$. This means that

$$S_P = \sum_{v_i \in V_S} w_i \leq W \tag{1}$$

and

$$H_P = \sum_{v_i \in V_H} p_i \leq A - K = \sum_{v_i \in V} p_i - K$$

the last one can also be formulated as:

$$K \leq \sum_{v_i \in V} p_i - \sum_{v_i \in V_H} p_i = \sum_{v_i \in V_S} p_i \tag{2}$$

(1) and (2) proves that $X = V_S$ is a solution of the original KNAPSACK problem.

Let now assume that X solves the KNAPSACK problem. Therefore:

$$\sum_{v_i \in X} s_i = \sum_{v_i \in X} w_i \leq W = S_0 \tag{3}$$

and

$$\sum_{v_i \in X} p_i \geq K = A - H_0 = \sum_{v_i \in V} p_i - H_0$$

that is

$$H_0 \geq \sum_{v_i \in V} p_i - \sum_{v_i \in X} p_i = \sum_{v_i \in V \setminus X} p_i = \sum_{v_i \in V \setminus X} h_i \tag{4}$$

(3) and (4) verifies that $P := (V \setminus X, X)$ solves PART1. □

Remark 1. *The above proof shows that the special case of the* PART1 *problem in which the graph has no edges is equivalent with the* KNAPSACK *problem.*

Theorem 2. PART2 *is* $\mathcal{NP}$*-hard.*

Proof. PART1 can be reduced to PART2: PART2 provides a solution where $H_P \leq H_0$ and S_P is minimal; let this value be S_P^*. Clearly PART1 is solvable iff $S_P^* \leq S_0$. □

It can be proven in the same way that

Theorem 3. PART3 *is* $\mathcal{NP}$*-hard.* □

Although the general partitioning problem is too hard for large inputs, some special cases are easier. If communication is cheap, *i.e.* $c(v_i, v_j) \equiv 0$, then the partitioning problem reduces according to Remark 1 to the well-known KNAPSACK problem, for which efficient pseudo-polynomial algorithms [12] and approximation algorithms [13] are known.

On the other hand, if communication is the only significant factor, *i.e.* $s_i \equiv 0, h_i \equiv 0$, then the trivial optimal solution is to put every node to software. However, if there are some predefined constraints considering the context of some nodes (*i.e.* the nodes in $\emptyset \neq \overline{V_S} \subseteq V$ are prescribed to be in software and the ones in $\overline{V_H} \subseteq V$, to be in hardware, where $\overline{V_H} \cap \overline{V_S} = \emptyset$) the problem reduces to finding the minimal weighted s-h-cut in a graph, where s and h represent the $\overline{V_S}$ and $\overline{V_H}$ sets, respectively. (If $\overline{V_H} = \emptyset$, then it reduces to finding a minimal weighted cut.) This can be solved in polynomial time [14].

IV. ILP-BASED ALGORITHM

The following ILP (integer linear programming) solution is appropriate for the PART3 problem, but it is straightforward to adopt it to the other versions of the partitioning problem.

$h, s \in (\mathbb{R}^+)^n, c \in (\mathbb{R}^+)^e$ are the vectors representing the cost functions (n is the number of nodes, e is the number of edges). $E \in \{-1, 0, 1\}^{e \times n}$ is the transposed incidence matrix of G, that is

$$E[i,j] := \begin{cases} -1 & \text{if edge } i \text{ starts in node } j \\ 1 & \text{if edge } i \text{ ends in node } j \\ 0 & \text{if edge } i \text{ is not incident to node } j \end{cases}$$

Let $x \in \{0,1\}^n$ be a binary vector indicating the partition, *i.e.*

$$x[i] := \begin{cases} 1 & \text{if node } i \text{ is realized in hardware} \\ 0 & \text{if node } i \text{ is realized in software} \end{cases}$$

It can be seen that the components of the vector $|Ex|$ indicate which edges cross the boundary between the two contexts. (The absolute value is taken component-wise.) So the problem can be formulated as follows:

$$\min hx \tag{5a}$$

$$s(\mathbf{1} - x) + c|Ex| \leq S_0 \tag{5b}$$

$$x \in \{0,1\}^n \tag{5c}$$

In Equation (5b) $\mathbf{1}$ means the n-dimensional $(1, \ldots, 1)$ vector. The (5a)-(5c) problem can be transformed to an ILP equivalent by introducing the variables $y \in \mathbb{R}^e$ to eliminate the $|\cdot|$:

$$\min hx \tag{6a}$$

$$s(\mathbf{1}-x)+cy \le S_0 \quad (6b)$$
$$Ex \le y \quad (6c)$$
$$-Ex \le y \quad (6d)$$
$$x \in \{0,1\}^n \quad (6e)$$

The last two programs are equivalent. If x solves (5b)-(5c), then $(x, |Ex|)$ solves (6b)-(6e). On the other hand, if (x, y) solves (6b)-(6e), then x will solve (5b)-(5c) too, since $y \ge |Ex|$ and $c \ge 0$.

Hence (6a)-(6e) is the ILP formulation of the PART3 problem. We solve this integer program using LP-relaxation and branch-and-bound [15].

V. GENETIC ALGORITHM

Although the ILP-based solution is efficient for graphs with up to some hundreds of nodes, and it produces the *exact optimum*, it cannot be used to partition even bigger graphs. For this purpose, we also developed a genetic algorithm (GA). For GA in general, see [16], [17] and references therein.

Individuals. The partitioning problem is fortunate from the point of view of a genetic algorithm. The applicability of genetic algorithms requires that the solutions of the optimization problem can be represented by means of a vector with meaningful components: this is the condition for recombination to work on the actual features of a solution.

Fortunately, there is an obvious vector representation in the case of the partitioning problem: each partition is a bit vector just like in the ILP program.

Population. The population is a set of individuals. The question is whether non-valid individuals, *i.e.* those violating for instance the software limit in case of PART3, should also be allowed in the population. Since non-valid individuals violate some important design constraint, it seems to be logical at first glance to work with valid partitions only. However, this approach would have several drawbacks: first, invalid individuals may contain valuable patterns that should be propagated, and second, it is hard to guarantee that genetic operations do not generate non-valid individuals even from valid ones. This holds for both mutation and recombination. For these reasons we decided to allow non-valid individuals as well in the population. Of course the GA must produce a valid partition at the end, so we must make sure to insert some valid individuals into the initial population, and choose the fitness function in such a way that it punishes invalidity.

Our tests showed that the population size should be around 300.

Initial population. In order to guarantee diversity in the population, the initial population usually consists of randomly chosen individuals. However, this method does not guarantee that there will be valid individuals in the initial population—in fact, if the problem space is big and constraints are tight, then the chances are very low for this.

Generating random valid individuals with approximately uniform distribution would be a promising alternative, but it is by no means obvious how one could implement such a scheme.

Therefore we chose to fill the initial population partly with randomly selected, not necessarily valid individuals, and partly with valid individuals generated by a fast greedy algorithm. This way, there are valid individuals, but also a wide variety of other individuals in the initial population. Clearly, the ratio between the two kinds of individuals in the initial population is a crucial parameter of the GA. The tests have shown that about one third of the individuals in the initial population should be chosen randomly.

Fitness function. Since we mainly focused on the PART3 problem, the objective is to minimize hardware cost. However, since invalid individuals should be punished, the fitness has a second component: the measure of invalidity, $inv(P)$, defined as the amount by which software cost (including communication cost) exceeds the software limit (and 0 for a valid partition).

We tried several versions for the fitness function, which fall basically into two categories: those ranking every valid individuals in front of invalid ones, and the less rigorous ones, that allow invalid individuals to beat valid ones. Later in our tests a rigorous version proved to be best:

$$f(P) = \begin{cases} H_P & \text{if } P \text{ is valid} \\ H_P + c * inv(P) + M & \text{if } P \text{ is invalid} \end{cases}$$

where H_P is the hardware cost, c is an appropriate constant determined by the tests, and M is a sufficiently large constant that makes each invalid individual have a higher fitness than any valid individual. (Note that the fitness has to be *minimized*.)

Genetic operations. Mutation, recombination and selection are used.

Mutation is done in the new population; each gene of each individual is altered with the same probability.

In the case of recombination we tested both one-point-crossover and two-point-crossover. Moreover, we tested two schemes for choosing the individuals for crossover: in the first one, all individuals are chosen with the same probability, in the second one, the probability of choosing a particular individual depends on its fitness, so that better individuals are chosen with a higher probability. So we tested four different versions of the recombination operator. According to the test results, the best strategy is to choose individuals with probabilities determined by their fitness, and to use two-point-crossover (although one-point-crossover is not much worse).

Selection is usually realized as filling some part of the new population with the best individuals of the old population. However, since some versions of the fitness functions rank all invalid individuals behind the valid ones, this would mean that invalid individuals are completely omitted from selection. Therefore, a given ratio of the selected individuals is taken from the invalid individuals.

Stopping criteria. The GA takes a given minimum number of steps (generations). After that, it stops after x steps if the best found partition does not improve in the last px steps. $0 < p < 1$ is also a parameter.

VI. EMPIRICAL RESULTS

The aim of our tests was threefold:

1. the limit of the applicability of the ILP algorithm had to be identified;
2. the parameters of the genetic algorithm had to be tuned;
3. the quality of the solutions found by GA had to be determined.

For the latter purpose the optimal solution of the ILP algorithm was used as reference in those problem instances that both algorithms were able to solve in acceptable time. We have tested the ILP-based and the genetic partitioning algorithm on both industry benchmarks and random problem instances.

First, we tuned the parameters of the genetic algorithm using random problem instances of varying size (400–1000 nodes). We varied the type of recombination, the fitness function, the constants in the fitness function, the size of the population, the ratio of random individuals in the initial population, the mutation rate, selection rate, percentage of invalid individuals from the selected ones, and the constants in the stopping criteria. Because of this huge number of parameters, we could not test all configurations. Rather, we first tuned the less sensitive parameters, and fixed them to the values that seemed best. We tested each configuration on 27 problem instances: 3 different problem sizes, 9 runs each, and we used the average of the 9 measurements for comparison.

After having tuned the parameters of the GA, we fixed them to the best found configuration, and moved on to compare it with the ILP algorithm on industry benchmarks.

The benchmarks we used are cryptographic algorithms (IDEA, RC6, MARS) of different complexity. Concerning the communication costs we tested two scenarios in every benchmark: *communication dominated*, in which the communication cost values relative to the software costs are very high, and *processing dominated*, in which the communication cost is in the same order of magnitude as the software costs. In both scenarios and in each benchmarks we tried seven different software limits. Altogether there were 42 test cases.

An overview of the results can be seen in Figures 1 and 2. Figure 1 shows the cumulative running time of the algorithms on each benchmark. The values on the y-axis mean the sum of the execution times of all tests in seconds. According to our expectations the running time of the ILP-solution grows exponentially with the problem size, in contrast with the slowly growing execution time of the GA. It can be seen that the ILP can solve problems with up to some hundreds of nodes in acceptable time, but becomes impractical on larger instances. We tested the GA on big random problems (random instances have the advantage that they can be generated and scaled easily) and found that it can solve problems of several

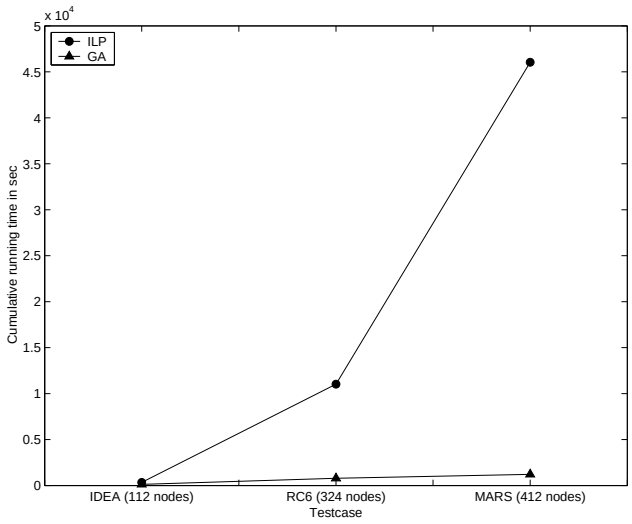

Fig. 1. The cumulative running time of the algorithms on benchmark problems

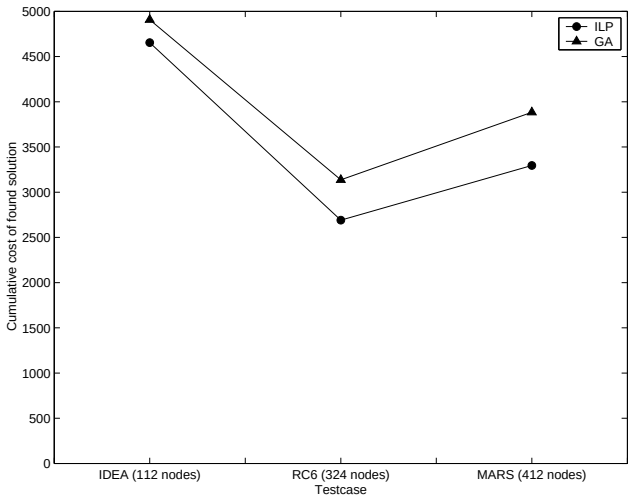

Fig. 2. The cumulative cost of the solutions found by the algorithms on benchmark problems

thousands of nodes in acceptable time. On a Pentium II PC, the GA can solve a problem with 2000 nodes within an hour, roughly the same running time that the ILP produces for a graph with about 300 nodes.

The quality of the found solution can be seen in Figure 2. This is the cumulative cost of the found solutions in each benchmark. Since the ILP always finds the optimum, we were interested in the deterioration of the result of GA. The figure shows that, on average, GA could find a solution close to the optimum. The exact deviances are: 5.4%, 16.5% and 17.8%. On larger tests we could not compare the result of GA with the result of ILP, since the latter was unable to terminate in acceptable time, but we expect the GA to finish not very

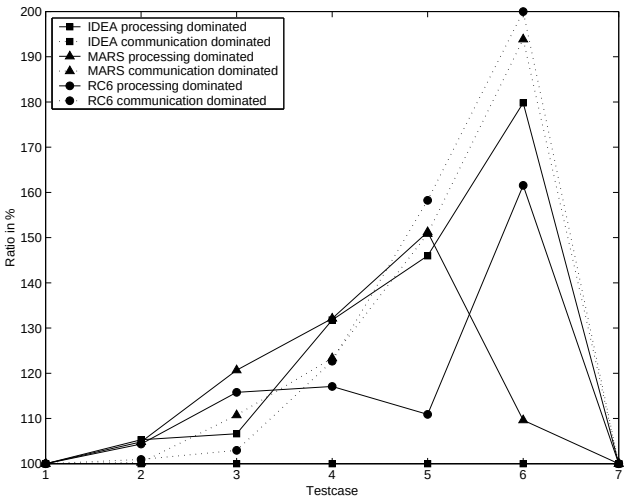

Fig. 3. The deviation of the results of GA from the optimum

far from the optimum. However, the gap increases with the problem size.

Figure 3 shows the deviation of the result of GA from the optimum in detail. On the x-axis the different tests according to the software limit, on the y-axis the deviation in % can be seen. In test-case 1 and 7 the software limit was chosen in such a way that the optimum is to put every component in software or hardware, respectively. It can be seen that GA finds these extremes in every case. Another consequence of the figure is that the difficulty of the problem depends strongly on the software limit. Tests 5 and 6 seemed to be the hardest problems, where in the worst case only an approximation ratio of 2 could be achieved. The best results were found in the 'IDEA, communication-dominated' scenario, in which all the seven test cases resulted in the optimal solution.

The easy-hard-easy pattern that can be recognized in Figure 3 is in accordance with previous findings in the literature [18]. We could expect a similar pattern in the running time of the ILP algorithm; however, this is not true. In fact, the running time of the ILP algorithm oscillates wildly. (In contrast, the running time of the GA is quite consistent.) This might be caused by the previously observed heavy-tailed runtime distribution of search algorithms [19]. Actually, it is possible that the ILP algorithm exhibits a similarly clear easy-hard-easy pattern *on average*, but this phenomenon is hidden because of the large deviation caused by the heavy-tailed property. In order to decide this, we are planning to implement a randomized ILP algorithm, and test its average behaviour.

VII. CONCLUSION

In this paper, we have introduced a new, simplified model for the hardware-software partitioning problem. This model has made it possible to investigate the complexity of the problem formally. In particular, we have proven that the problem is $\mathcal{NP}$-hard in the general case, but we could also identify some efficiently solvable meaningful special cases. Moreover, we presented two new partitioning algorithms: one based on ILP, and a genetic algorithm. It turned out in our empirical tests that the ILP-based solution works efficiently for graphs with several hundreds of nodes and yields optimal solutions, whereas the genetic algorithm gives near-optimal solutions on average, and works efficiently with graphs of even thousands of nodes. Moreover, we observed an easy-hard-easy pattern in the performance of GA, and wild oscillations in the running time of the ILP algorithm.

Our future plans include randomization of our ILP-based algorithm, developing better bounds for the branch-and-bound scheme used in the ILP algorithm, as well as the inclusion of some simple scheduling methods into our partitioning model.

REFERENCES

[1] P. D. Randall, "Hardware/software co-design and digital signal processing," MSc thesis, University of Oxford, May 1996.

[2] P. Arató, Z. Á. Mann, and A. Orbán, "Hardware-software co-design for Kohonen's self-organizing map," in *Proceedings of the IEEE 7th International Conference on Intelligent Engineering Systems*, 2003.

[3] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, PhD thesis, Stanford University, December 1993.

[4] Z. Á. Mann and A. Orbán, "Optimization problems in system-level synthesis," in *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003.

[5] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," *Design Automation for Embedded Systems, special issue: Partitioning Methods for Embedded Systems*, vol. 2, pp. 165–193, March 1997.

[6] R. Niemann, *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*, Kluwer Academic Publishers, 1998.

[7] J. Grode, P. V. Knudsen, and J. Madsen, "Hardware resource allocation for hardware/software partitioning in the LYCOS system," in *Proceedings of Design Automation and Test in Europe (DATE '98)*, 1998.

[8] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "Hardware/software partitioning of VHDL system specifications," in *Proceedings of EURO-DAC '96*, 1996.

[9] K. S. Chatha and R. Vemuri, "MAGELLAN: Multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs," in *Proceedings of CODES 01*, 2001.

[10] N. N. Binh, M. Imai, A. Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts," in *Proceedings of the 33rd Design Automation Conference*, 1996.

[11] G. Quan, X. Hu, and G. Greenwood, "Preference-driven hierarchical hardware/software partitioning," in *Proceedings of the IEEE/ACM International Conference on Computer Design*, 1999.

[12] C. H. Papadimitriou, *Computational complexity*, Addison Wesley, 1994.

[13] D. S. Hochbaum, Ed., *Approximation Algorithms for NP-Hard Problems*, PWS Publishing, Boston, MA, 1997.

[14] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.

[15] A. Schrijver, *Theory of linear and integer programming*, Wiley, 1998.

[16] L. Davis, *Handbook of genetic algorithms*, Van Nostran Reinhold, 1991.

[17] W. Kinnebrock, *Optimierung mit genetischen und selektiven Algorithmen*, Oldenburg, 1994.

[18] D. Achlioptas, C. P. Gomes, H. Kautz, and B. Selman, "Generating satisfiable instances," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, 2000.

[19] C. P. Gomes, B. Selman, N. Crato, and H. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," *Journal of Automated Reasoning*, vol. 24, no. 1-2, pp. 67–100, 2000.