An Automatic Hardware-Software Partitioner Based on the Possibilistic Programming.

I. Karkowski and R.H.J.M. Otten
Delft University of Technology
Faculty of Electrical Engineering

Abstract

The problem of hardware-software partitioning in the design of embedded systems is addressed. Uncertainties about the performance of the options for realization are expressed in triangular possibilistic numbers. To handle such numbers an integer programming formulation of the partitioning problem is derived. This formulation can be converted into a possibilistic program without changing the asymptotic computational complexity. The approach is illustrated with results obtained with the receiver part of a transceiver of a wireless indoor spread spectrum system. This example and several other experiments have shown that these optimizations can reach solutions within seconds for designs of that complexity and above.

1 Introduction

In this context *embedded systems* are combinations of hardware with general purpose computational capabilities and more dedicated modules. Together they perform a function carefully partitioned in software and hardware to obtain the optimum trade-offs between the various performance metrics. These types of implementations have become increasingly popular as advances in ic-technology and processor architectures allow for flexible computational parts and high-performance modules integrated on a single carrier. What is needed is an interactive environment that supports the designer in transforming an algorithmic specification into a suitable implementation [TSV94]. A central problem in automated co-synthesis of hardware and software in embedded systems is the hardware/software partitioning. Two different approaches exist. A *software-oriented* approach has been implemented in the **COSYMA** system [EHB93]. It starts from an all software solution and iteratively selects operations to be moved to hardware until all timing constraints are met. Another co-synthesis approach [GCM92], *hardware-oriented*, starts from a complete hardware description and iteratively selects components, which can be implemented in software without violating the performance constraints. Our partitioning approach is non-iterative and has the following features:

- The program accepts as input a specification in a high level language (at this moment a superset of ANSI C).
- The model is a quite straightforward mapping of the input language statements into a partitioning graph.
- It supports parallel flows of execution within the program (by adding extra constructs), loops and conditional statements.
- All information about functions and variables can be imprecise, by giving the coefficients a possibility distributions [Zad78]. This is essential! Otherwise we could be optimizing a model which is quite far from reality, or with a high probability of not meeting the constraints. We choose for possibility distributions because they model our knowledge about the design parameters better than for example probability distributions. Another consequence of this choice is that the computational complexity of the used partitioning algorithms remains in the same class. Also generating meaningful possibility distributions is much easier than deriving probability distributions.
- The partitioning program runs fully automatically, but support for user interaction and iterative improvement is provided. This allows the user to experiment with several implementation alternatives.
- In addition to partitioning, the program also automatically selects a main processor from the list of CPUs available.
- The approach allows for a large degree of flexibility - many kinds of constraints may be specified by the user. Also several optimization criteria can be investigated. It is easy to take into account many important design aspects and user preferences.
- Functional units can execute simultaneously with the main CPU. This extends the approach presented in [EHB93], where the processor activates a coprocessor and then waits idling until it delivers a result. Without this extension it would be impossible to take advantage of the intrinsic parallelism usually present in the system specification.

We would like to emphasize that our approach differs substantially from previous results reported in the area

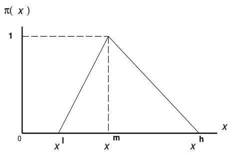

Figure 1: The triangular possibility distribution of fuzzy number x.

of synthesis of application-specific multiprocessor systems [PP92]. Here follow main differences:

- Our program does not perform scheduling simultaneously with partitioning (in section 3 is explained why). Thanks to that, the program is smaller and contains only binary/integer variables. This results in very short solving times. For instance it takes only 4 seconds on HP735 computer to solve an example with 38 nodes and 66 edges. The SOS system [PP92] needed 6416 minutes of CPU for an example with only 9 nodes, due to the mixed-integer nature of their formulation.
- The program partitions not only functions to be implemented, but also variables used in the program. Thanks to that, a better modelling of data transfers is obtained. For every function, variable or data transfer several implementation options are available.

2 Possibilistic programming

2.1 Possibility distributions

To define fuzzy/imprecise numbers, we use possibility distributions[1], as introduced by Zadeh [Zad78], see e.g. figure 1. In this study, the possibility measure of an event might be interpreted as the possibility degree of its occurrence under the possibility distribution, $\pi(.)$, (analogous to a probability distribution).

Among the various types of distributions, triangular and trapezoid are the most common in solving possibilistic mathematical programming problems. We will concentrate only on triangular fuzzy numbers.

The triangular fuzzy number is denoted by $X = (x^m, x^l, x^h)$, where x^m is the *most possible* value, x^l and x^h are lower and upper bound values of the acceptable events, respectively. These bound values can be interpreted as the most pessimistic and the most optimistic values. Which one is pessimistic and which optimistic depends on the context.

2.2 Linear programming with imprecise objective coefficients

A possibilistic linear program (**PLP**) has the following form:

[1] in fuzzy mathematical programming we use membership functions instead.

$$\min \quad \tilde{c}x \tag{1}$$
$$s.t. \quad \tilde{A}x \leq \tilde{b}, and \quad x \geq 0 \tag{2}$$

where $\tilde{c}$, $\tilde{A}$, and $\tilde{b}$ may consist of imprecise numbers with possibility distributions. Suppose only c is imprecise. In [LH92] it is shown that this is not an essential limitation. For given x, the value of the fuzzy objective function (eq. 1) is a fuzzy number defined by three corner points ($c^m x$,1),($c^l x$,0) and ($c^h x$,0). Thus, reducing the fuzzy objective can be achieved by pushing these three critical points to the left without loosing the triangular shape (normal and convex) of the possibility distribution function. It is attractive to consider the following three objectives simultaneously: a small $c^m x$, a large $[c^m x - c^l x]$ and a small $[c^h x - c^m x]$, see figure 2. This is written as:

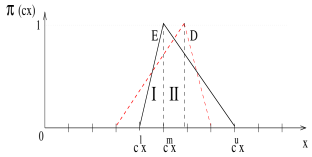

Figure 2: The strategy to solve " $\min \tilde{c}x$". We prefer the possibility distribution of D to that of E.

$$\begin{aligned} \max \quad & z_1 = (c^m - c^l)x \\ \min \quad & z_2 = c^m x \\ \min \quad & z_3 = (c^h - c^m)x \end{aligned}$$

$$s.t.\ x \in X = \{x : Ax \leq b\ ,\ x \geq 0\} \tag{3}$$

This Multiple Objective Linear Program (**MOLP**) aims at reducing the *most possible* value of the imprecise cost (at the point of possibility degree = 1), but at the same time, we "minimize the risk of paying higher cost" (see region II in figure 2), and "maximize the possibility of the lower cost" (region I). Any **MOLP** technique can be used (e.g. utility theory, goal programming, fuzzy programming or interactive programming) [LH92][KO95].

3 Embedded system modelling

Our model is based on the following assumptions:

- An embedded system consumes some data, performs some operations on them and generates some data. The software component of the embedded system is thought to consist of a set of variables and exclusively/concurrently executing routines, called *basic blocks*. A set of single/multiple activations of basic block(s) necessary to process a given amount of input data is a *run* of the system.
- The total time (in clock cycles) spent by the processor in one *run* of the system is called the *latency* of the *run*.

- The set of basic blocks is a result of the clustering operation on all operations. This clustering can be *locked* by the user or done automatically. In the first case the highest level of the calling hierarchy in the input program is chosen as the partitioning boundary. Hardware software partitioning on too low a level does not seem to be promising because of the interfacing overheads.
- The design contains at least one processing unit, selected from the list of available processor templates. (In our formulation we assume exactly one processor.)
- One of the basic blocks, called the $Supervisor$ represents the main scheduling engine of the system. It accommodates all operating system and program overheads. It must be implemented in software.
- Every basic block can have several implementation options, some of them in hardware. They may have the form of multi-dimensional *trade-off curves*.
- Functional units/co-processors can perform their function independently of and simultaneously with the main processor.
- The embedded system is *software dominated*. This means that majority of the basic blocks will remain in software and that the critical path will be located there. The consequence of this assumption is that we do not have to perform scheduling simultaneously with the partitioning.

The basic blocks are vertices of a directed graph $G =< V, E, C >$ that represents the functionality of the system to be designed. The edges $e \in E$ represent data transfers between basic blocks ($e : u \stackrel{e}{\rightarrow} v$ means that some data is transferred from the basic block u to the basic block v). The edges $e \in C$ represent control flow. Both, vertices and edges, have a number of performance and cost parameters associated with the various realization options. The values of these parameters may be real or possibilistic numbers. They can be used as coefficients for **IP** formulations of our problem. The coefficients for vertices and edges are presented in table 1. Further there is also some general design information, for example as in the third part of the table.

Example 1 *In figure 3 the source code of a part of the WISSCE receiver [Gla94] is presented. This standard ANSI C code is augmented with < and > brackets to denote that all statements within them can be executed in parallel. The system implementing this algorithm will continuously repeat operations specified by this diagram. The set of statements and variables from this figure can be directly mapped into the set of basic blocks. Basic blocks from the two compound-statements can execute concurrently since they are placed within triangular braces. The latency of a single* run *may be imprecise, because the exact time that some operations take is unknown or may vary. For example* $readSample$ *usually takes 1 cycle, but it can take as much as 3 cycles. Also the latency of other operations may be data dependent (for example the* **limit** *function).*

Coef.	Description
F_i	execution rate of the basic block i (how many times it executes in one run)
$T_{i,j}$	latency of the j-th hardware implementation of the basic block i, (real time units)
$L_{i,p}$	latency of the software implementation of basic block i on processor p, (clock cycles).
$C_{e,j}$	size of the code for an interface on the edge e if it is implemented in the j-th way
$L_{e,j}$	latency of the j-th implementation of the interface at the edge e
F_e	execution rate of the interface on edge e (the same as at vertices)
D_e	amount of data to transfer through the edge e during one transaction
P_j	the cycle time of the processor j, (real time units)
A_u	size of the functional unit of type u
A	area available, for example on a master image
C	upper bound on the code size, for example in a fixed rom

Table 1: The vertex, edge and general information coefficients

Example 2 *In figure 4 the graph model of the receiver from the figure 3 is presented. The oval vertices represent routines and the square ones the variables. The continuous arrows are used for the control flow, while the ones with dotted lines for the data flow. Only the coefficients assigned to one vertex called "bb5" are depicted. As can be seen some of them are crisp values representing the well-known values for the coefficients, some are triangular fuzzy numbers capturing the uncertainty.*

4 Problem formulation

In case the data about the design is precise we formulate our problem as an integer program (**IP**). Integer programming **IP** has a number of salient features. When it finishes it is guaranteed to have reached a globally optimal solution. It offers quite some flexibility in formulating combinatorial solutions. One may even conjecture that any combinatorial problem can be formulated as an **IP** although it is not always easy. There is also a quite large degree of flexibility in cost functions. **IP**'s can be adapted to possibilistic theory in a straightforward manner, without increasing the number of variables. However, integer programming in general is $\mathcal{NP} - hard$. Many years of experience in the Operations Research have revealed that applying general **IP** methods is

```
bit acq=0;
kasgen gen;  /* kasami-code generator */
register IQ sample;
              /* read input from ad-converter */
sample = readSample();
shiftPNgen(&gen, trackCntr, acq);
<  /* beginning of the parallel block */
 {       /* beginning of the first branch */
  register byte promptCode;
  register IQ promptSample;
  register short pPointer;
  pPointer = i;
  promptSample = sample;
  promptCode = getPrompt(&gen);
  promptSample = IQmpy(promptSample, promptCode);
  promptSample = filter(promptSample);
  promptBuf1[pPointer] = limit(promptSample);
 }
 {            /* second branch */
  register byte trackCode;
  register IQ trackSample;
  register short tPointer;
  tPointer = i;
  trackSample = sample;
  trackCode = getEarly(&gen) - getLate(&gen);
  trackSample = IQmpy(trackSample, trackCode);
  trackBuf1[tPointer] = limit(trackSample);
 }
>             /* end of the parallel block */
```

Figure 3: The C specification of a part of the WISSCE receiver

only recommended when either the problem has some kind of special structure, or the size of the problem is moderate. In our case the first is true, because we formulate the program in similar manner as for the *scheduling* problem from [GE90]. Since the partitioning is performed on the result of the clustering of all operations the second condition is also true. The details of the **IP** formulation of the problem and estimates of its size are presented in appendix A.

5 Handling imprecision

Assume now that some parameters in the **IP** program are imprecise. For example execution rate F_i of a basic block and of an interface F_e may be data dependent. The amount of data to transfer through an edge D_e may differ when for example we transfer a list. Also the latency of a hardware implementation $T_{i,j}$ and the exact size of a functional unit A_u may be not exactly known when the unit has not been synthesized yet. The situation is the same with software components. For instance, the size of the code $C_{e,j}$ depends on heuristic optimizations performed by the compiler and is therefore imprecise.

By substituting imprecise numbers into the integer program (appendix A) we obtain a possibilistic integer program (**PIP**). The method from section 2.2 can be directly applied to solve it.

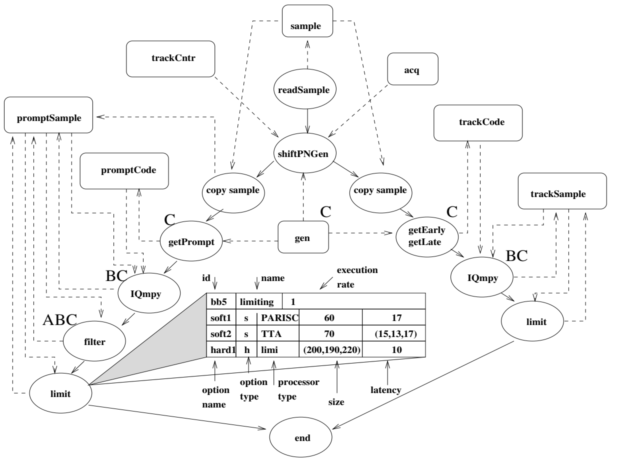

Figure 4: The graph model of the WISSCE receiver.

6 Experimental results

The partitioner program is called HSPART and has been implemented in C++ on HP9k/735 Unix computer. We built it on the top of our general possibilistic solver called FSOLVE. It forms the central part of our hardware/software co-design environment presented in figure 5. Let us shortly describe the function of all its components. The algorithm

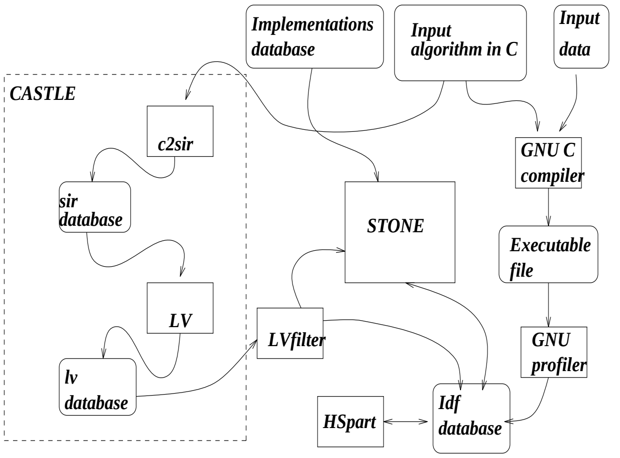

Figure 5: The hardware software co-synthesis environment

specification in C (augmented to allow parallel flows of execution) is parsed by the **c2sir** tool (part of the CASTLE system [TSV94]) and written into the SIR database. Then, it is converted using the **lv** program to another (more convenient for us) format, called *LV*. Data and control flow graphs from that file are parsed by a filter program called **sirFilter**. The program removes all dummy or unnecessary (not partitioned) nodes and edges. We use the call hierarchy of the program to determine the partitioning boundaries. The partitioning stops at the first level. This requires the program to be written with this strategy kept in mind.

If the algorithm contains parallel flows of execution, appropriate paths in the control flow graph are split and put in parallel. The result is read by the STONE program, which serves as a graph viewer and general user interface. The program uses the so called *Idf* data format to store all information. Simultaneously the input code is compiled with the GNU C compiler and profiled using provided input data. The result is written into the *Idf* database. Now there are two possibilities:

1. the user defines all possible implementations for all basic blocks and data transfers, using dialog boxes within STONE,
2. the information about the implementations is imported from other projects (implementations database in figure 5).

After that, the user can define design constraints, types of available processors etc. Also locking of some implementations is allowed. All the information is directly written into the *Idf* database. Now the partitioner (HSPART) is called taking input from the *Idf* file. The resulting partition is written back to the database and is displayed in STONE. The user can now evaluate the obtained solution, modify it and eventually run the partitioning again, until satisfactory solution is reached.

To test the program we run it on several examples, also on the WISSCE receiver. Figure 3 presents the code used as the system's input. We run the program 780 times (3 input sequences in WISSCE). This gives 780 executions of every node and edge. Data about hardware implementations was provided by experts. The advantages of using possibility distribution instead of just a single most expected value became obvious almost immediately. It was difficult to predict exactly even the number of clock cycles that certain function will take on the CPU. Predicting the total number of transistors eventually used by the hardware is known to be a difficult problem. Allowing for fuzzy numbers has been welcomed with enthusiasm. At the same time another advantage of our approach has been confirmed - there was simply no way of generating good probability distributions other than generating every potential candidate vertex several times - a designer's nightmare.

We use the HP PARISC1.1 general purpose processor as the CPU of the receiver. To obtain data about the latencies (number of clock cycles) of the software versions of the basic blocks we compiled the code with the GNU C compiler with debug and optimization options on. Next we investigated the generated code with the gdb debugger. The number of cycles that the argument's passing takes was used as the data transfer latency. The call instruction, saving/restoring registers, allocating/deallocating local frame and the body of the callee accounted for the latency of the basic blocks. Latency of all other instructions (initialization of global variables and loops' control) constituted the latency of the *Supervisor* vertex. Figure 6 shows latencies of all non-variable basic blocks in the program.

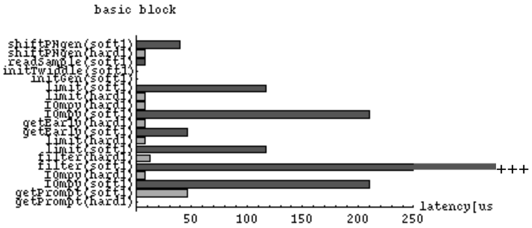

Figure 6: Total latencies of all basic blocks

The running times were rather short. For instance for the *receiver* example they were well below 1 second. We optimized for the minimum area of the extra hardware. Three timing constrains were used:

1. *latency*: this constraint limits the latency of one *run of the system* (total time spent by the CPU in one *run of the system*).
2. *path1*: limits the total time that basic blocks **readSample, shiftPNgen, getPrompt, IQmpy, filter, limit** and associated with them data transfers take.
3. *path2*: limits the total time that basic blocks **readSample, shiftPNgen, getEarly, IQmpy, limit** and associated with them data transfers take.

We experimented with different time values in these constraints. The partitioning results can be seen in figure 4. Let us shortly try to analyze the obtained results (in figure 4 vertices moved to hardware are annotated with capital letters):

1. We set the timing constraints to be $1ms$. As can be seen in figure 6, the software version of the **filter** operation takes almost 2/3 of the total time. To satisfy the timing constraints it has been moved to hardware (letter A). All other operations could remain in software.
2. The timing constraints are set to be $600\mu s$. Now neither path satisfies the constraints and therefore something else has to be moved to hardware (vertices with letter B). **IQmpy** and **limit** are better candidates than **getPrompt** and **getEarly** because their basic blocks (belonging to both paths) can use the same unit. The gain in performance obtained by moving **IQmpy** or **limit** is sufficient to satisfy the constraints. The sizes of hardware for the operations are $(50, 45, 60)$ for **IQmpy** and $(50, 46, 65)$ for **limit**. As can be seen the most possible values are almost the same, but the hardware implementation of **limit** is slightly more "risky". Therefore the solution with **IQmpy** in hardware has been chosen. The salient feature of the possibilistic approach is clear. If there exist solutions with a similar *most possible* value of the cost function, the solution with lower "risk" of obtaining a "worse" solution and higher chance of getting a "better" solution is selected.

3. The timing constraints are $600\mu s$. In addition we lock the implementation of the **gen** basic block to be in hardware. The Kasami-code generator (represented by this variable) is large. Only transferring it, without actually doing any other computation could violate the constraints. There are 3 data transfer edges from **gen**. Two of them require access to whole **gen** object, one uses only a small part of it (basic block **shiftPNgen**). Therefore to avoid spending too much time in data transfers the partitioner selected hardware implementations also for basic blocks **getPrompt** and **getEarly** (vertices with letter C).

As can be seen the partitioner generates very logical results. Also the imprecise nature of the system parameters is taken into account.

7 Conclusions

In this paper we presented an automatic partitioner for hardware-software co-synthesis. Our approach differs from the others by the fact that it is non iterative, flexible and taking into account the imprecise nature of the available information. These goals are obtained by using possibilistic programming. The program was successfully applied on the partitioning of a part of the **WISSCE** spread spectrum receiver.

A Appendix: IP formulation details

In this appendix we will present details the **IP** formulation of the problem and estimate its size.

Symbol	Description
P	set of available processors
H	set of hardware implementations
S	set of software implementations
U	set of available hardware units
I	set of possible ways of implementing interfaces(between software, between hardware and between basic blocks of mixed types)
I^p	a subset of I for processor p

Table 2: Symbols used in the **IP** formulation

The general symbols used are presented in table 2, while all decision and temporary variables are explained in table 3.

A.1 Constraints & cost functions

The program contains at least the following constraints:

1. Implementation selection constraints. Every basic block i has to be implemented either in software or in hardware:

$$\forall_{i \in V} \quad \left[\sum_{p \in P} s_{i,p} + \sum_{j \in H_i} h_{ij} = 1\right], \qquad (4)$$

$$\forall_{e \in E} \qquad \left[\sum_{j \in I} m_{e,j} = 1\right] \qquad (5)$$

Var.	Description
$h_{i,j}$	bin. var. assuming value 1 if bb i is implemented in HW, using j-th impl. option
$s_{i,j}$	assumes value 1 if bb i is implemented in sw, on a processor of j-th type
p_i	assumes value 1 if we use processor i
s_e	temp. bin. var., assumes value 1 if it is necessary to provide interface in SW at e
h_e	temp. bin. var., assumes value 1 if it is necessary to provide interface in HW at e
$m_{e,j}$	bin. var. which assumes value when 1 the j-th interfacing scheme on edge e is used ($j \in I$)
n_u	number of FU's of type u used
L	total time spent by the processor
C	SW size (in bytes of code)
A	the area of the extra HW

Table 3: Decision and temporary variables used in the **IP**, var - variable, bb - basic block, HW - hardware, SW - software

2. "Interface type selection" constraints. There are three kinds of interfaces: software, hardware and mixed. A software interface for edge $e : u \stackrel{e}{\to} v$ is necessary if basic blocks u and v are implemented in software. This can be expressed as:

$$\forall_{e \in E} \left[s_e = \sum_{j \in P} s_{u,j} \circ \sum_{j \in P} s_{v,j}\right]. \qquad (6)$$

Two basic blocks in hardware generate a hardware interface:

$$\forall_{e \in E} \left[h_e = \sum_{j \in P_u} h_{u,j} \circ \sum_{j \in P_v} h_{v,j}\right]. \qquad (7)$$

$\circ$ is the boolean *and*. This kind of constraint can be replaced by the following two arithmetic constraints:

$$\left\{\prod_{j \in N_k} x_j = y_k\right\} \Rightarrow \left\{\begin{array}{ll} \sum_{j \in N_k} x_j - y_k & \leq |N_k| - 1, \\ \frac{1}{|N_k|} \sum_{j \in N_k} x_j - y_k \geq 0. & \end{array}\right\}$$

where $\prod$ is a boolean product, N_k is the index set of all variables appearing in the product.

3. Interface assignment constraints. If an interface on edge e is necessary then select exactly one implementation scheme for it:

$$\forall_{e \in E} \quad \left[\sum_{s \in S} m_{e,s} = s_e\right] \qquad (8)$$

$$\forall_{e \in E} \quad \left[\sum_{h \in H} m_{e,h} = h_e\right] \qquad (9)$$

Note that if $s_e = 0$ and $h_e = 0$ then because of the constraint (eq. 5) one of the mixed options (interface between hardware and software) has to be selected.

4. Processor selection constraints. They guarantee that only one processor is selected:

$$\sum_{j \in P} \quad p_j = 1, \qquad (10)$$

$$\forall_{i \in V} \forall_{j \in P} \quad [p_j \geq s_{i,j}] \qquad (11)$$

$$\forall_{j \in P} \quad \left[p_j \leq \sum_{i \in V} s_{i,j}\right] \qquad (12)$$

$$\forall_{p \in P} \forall_{e \in E} \quad \left[p_j \geq \sum_{j \in I^p} m_{e,j}\right] \qquad (13)$$

The second and third group of constraints ensures that p_j is equal 1 only if at least one of basic blocks is implemented in software on processor j. The fourth group guarantees that p_j is equal 1 only if at least one of the interfaces is implemented on processor j.

5. *Latency* of a *run* constraint (total time spent by the processor):

$$\sum_{i \in V} \sum_{s \in S} F_i L_{i,s} s_{i,s} + \sum_{e \in E} \sum_{j \in I} F_e L_{e,j} m_{e,j} \leq L \quad (14)$$

The first term includes all software basic blocks. The second one represents time spent in the interfaces.

6. Software size constraints. We assume that the code for every interface on edge e must be generated separately:

$$\sum_{i \in V} \sum_{p \in S} C_{i,p} s_{i,p} + \sum_{e \in E} \sum_{j \in H \cup S} C_{e,j} m_{e,j} \leq C \quad (15)$$

7. The size of the extra hardware. We assume that several basic blocks can be mapped to the same hardware unit.

$$\sum_{u \in U} \quad A_u n_u \leq A, \quad (16)$$
$$\forall_{u \in U} \quad \forall_{i \in Op(u)} \quad [h_{i,u} \leq n_u]. \quad (17)$$

In the equations above binary variable n_u takes value 1 if at least one basic block has been mapped to it. $Op(u)$ represents the set of all basic blocks that can be assigned to unit type u.

8. Many other constraints taking into account power consumption, number of pins, size of the code, design and prototyping time can be easily formulated.

On the basis of these constraints many programs can be constructed and solved. For example we can minimize extra hardware area under total latency constraint or minimize total latency under constraint on the size of the ASIC part. Also cost functions combining several optimization criteria are possible.

A.2 Program size estimates

There will be $|H_i| * |V| \quad h_{i,j}$ variables, $|S_i| * |V| \quad s_{i,j}$ variables, $|E| \quad s_e, h_e$ variables, $|E| * |I_e| \quad m_{e,j}$ variables, $|U| \quad A_u$ and n_u variables and $|P| \quad p_i$ variables.
Because P, H_i, S_i and I will be usually small positive numbers ($\ll 10$) the size of the program will be proportional to the number of vertices and data transfer edges. Our experience learns that when the size of the program is even about 1000 binary variables the optimal solution can be reached within few seconds. Also larger examples were optimized in reasonable time.

References

[CPL93] CPLEX Optimization Inc. *The CPLEX Linear Optimizer and Mixed Integer Optimizer with Barrier. User manual*, second edition, 1993.

[EHB93] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrolers. In *IEEE Design & Test of Computers*, December 1993.

[GCM92] R.K. Gupta, C.N. Coelho, and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proc. of DAC'92*, pages 225–230, 1992.

[GE90] C.H. Gebotys and M.I. Elmasry. A global optimization approach to architectural synthesis. In *IEEE International Conference on Computer Aided Design*, pages 258–261, 1990.

[Gla94] J.P.F. Glas. On multiple access interference in a ds/ffh spread spectrum communication system. In *Proceedings of the Third IEEE International Symposium on Spread Spectrum Techniques and Applications*, Oulu, Finland, July 1994.

[KO95] I. Karkowski and R.H.J.M. Otten. Retiming synchronous circuitry with imprecise delays. In *Proceedings of 32nd DAC*, June 1995.

[LH92] Y.J. Lai and C.L. Hwang. A new approach to some possibilistic linear programming problem. *Fuzzy Sets and Systems*, 49:121–133, 1992.

[PP92] S. Prakash and A.C. Parker. Sos: Synthesis of application-specific heterogeneous multiprocessor systems. *IEEE Journal on Parallel and Distributed Computing*, page 338, 1992.

[TSV94] M. Theißinger, P. Stravers, and H. Veit. Castle: An interactive environment for hw-sw codesign. In *Proc. of the HW-SW Co-Design Conf.*, 1994.

[Zad78] L.A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, 1:3–28, 1978.