Hardware-Software Co-Design for Kohonen's Self-Organizing Map

Péter Arató, Zoltan Ádám Mann, András Orbán
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
H-1117 Budapest, Magyar Tudósok Körútja 2.
Hungary
{arato, mann}@iit.bme.hu, Andras.Orban@cs.bme.hu
This paper was published in the Proceedings of the 7th IEEE International Conference on Intelligent Engineering Systems, 2003.

Abstract— **Kohonen's self-organizing map (SOM) is a widely used technique to cluster unstructured data. It has applications in computer graphics, image processing, robotics, soft-computing and many more. The exact specification and time requirements may vary according to the concrete application, therefore a re-design of Kohonen's SOM offering several different performance/cost trade-offs by using special purpose hardware acceleration would be beneficial. This paper introduces the main principles of system-level synthesis (SLS) which aims at constructing optimal hardware-software systems fulfilling a high-level specification and applies its methodologies to construct different hardware-software SOM realizations corresponding to the concrete expectations.**

I. Introduction

The convergence of hardware and software development may lead to the disappearance of separate software- and hardware vendors, but rather companies offering different price/performance solutions fro the same functionality, beginning from a pure software realization until the ASIC (Application-Specific Integrated Circuit), will emerge. They may start with a software version, and according to the different time-requirements some part of the complex system may be accelerated by special purpose hardware elements. Moreover, in the parts chosen for hardware implementation there is a further optimization possibility: one can either strive for the best possible parallelization, which is rather costly, or to settle for a cheaper but slower solution.

The implementation of Kohonen's SOM [1] has been chosen as benchmark because of its simplicity and significance in many areas, like robotics, computer graphics, face recognition, document clustering etc. The actual realizations of the SOM may differ in the various fields according to the particular time and/or performance requirements. For example, if a robot uses the SOM for recognizing the objects of the realworld to coordinate its movements, then the performance of the clustering algorithm plays a key role; if it is used to recognize the number plate of a car in a petrol station, the speed is not that important, rather the system should be possibly cheap.

Now, we give a short introduction to the concepts and methods of system-level synthesis (SLS, [2], [3]). The goal of SLS is to automatically design the optimal hardware and/or software structure from the high-level (yet formal) specification of a system. The optimality criteria may differ according to the particular application; in our model we will use the number of required processing units in the case of hardware and the execution time in the case of software as main cost factors.

The central data structure of SLS is the elementary operation graph (EOG), which is an attributed data-flow graph. Its nodes represent elementary operations (EOs). An EO might be *e.g.* a simple arithmetic operation but it might also be a complex function block. Each EO has a given duration $d_i \in \mathbb{N}$. The edges of the EOG represent data flow—and consequently precedences—between the operations.

One important problem of SLS is partitioning: deciding which parts of the problem should be realized in hardware and which ones in software, taking into account hardware, software and communication costs. One possible approach to partitioning is to start with a pure software solution, identify the slowest parts and construct a special purpose hardware for them. The parts that should be realized in hardware will be designed using the methodologies of high-level synthesis (HLS, [2], [4], [5]), which aims the automatic construction of hardware units based on a high-level behavioral description. SLS can be regarded as an extension of HLS, since several methodologies of HLS can be transferred to SLS.

The process of HLS starts by building the EOG from the specification, then some transformations are made on the graph in order to meet timing and consistency requirements. Finally, the elementary operations are scheduled—*i.e.* their starting times are determined—and allocated in real processing units. The actual optimization is carried out in these last two steps. Both are computationally hard [6].

The above problems are complicated by the fact that we consider pipeline hardware to achieve maximum throughput, since pipeline processing can achieve higher performance even in systems that are difficult to parallelize. A pipeline system is characterized by two numbers: latency (denoted by L) is the time needed to process one data item, while restart time (denoted by R) is the period of time before a new data item is introduced into the system. Generally $R \le L$. Thus, non-pipeline systems can be regarded as a marginal case of pipeline systems, with $R = L$.

At the Department of Control Engineering and Information Technology at Budapest University of Technology and Economics, a software called PIPE [2] has been developed, which realizes the HLS procedure. The unique feature of PIPE which distinguishes it from other HLS tools is that it takes pipeline processing into account from the beginning

of the design process: the desired R and L values are parts of the input of PIPE. If a large amount of data has to be processed, then minimizing R at the cost of a reasonable increase in L or hardware price is an important objective of HLS.

The paper is organized as follows: Section II gives a short overview about related work, Section III presents the software implementation of the Kohonen's SOM, which is the starting point of the partitioning. The redesign using SLS methods and the PIPE system is described in Section IV and finally Section V concludes the paper.

II. Existing hardware and hardware/software implementations of the SOM

There are already some existing hardware and hardware/software solutions for Kohonen's SOM algorithm. Both commercial tools and research projects are dealing with that problem. Since the notion *Kohonen's SOM* does not cover a single algorithm, but rather a family of methods, so to set them against each other is rather complicated and only partially meaningful.

A. Commercial tools

This subsection lists commercial neuro-chips, neural coprocessors and neural accelerators. They are usually not made specifically for the Kohonen algorithm but their vendor claims that they also support it, or they can be used to accelerate operations that are also critical in the Kohonen algorithm.

SAND (Simple Applicable Neural Device [7]) is a neural processor based upon the principle of a systolic processor array. Four parallel processor elements form the heart of this array. Each processor element has a multiplier and two adders, one of which serves as an accumulator. With a maximum clock frequency of 50 MHz, SAND achieves a performance of 200 MCPS (Mega Connections Per Second). Multiple SAND chips may be connected in parallel in order to attain a further increase in performance.

NeuroLution [8] is a tool for the development of artificial neural network applications and contains hardware and software components, which can be used as stand-alone tools or in an integrated form. The system is available for PCs under Windows 95 and Windows NT.

The CNAPS system [9] is a full NNW (Neural NetWork) development system based on the proprietary CNAPS-1064 Digital Parallel Processor chip that has 64 sub-processors operating in SIMD mode. Each sub-processor has its own 4KB local memory and a fixed-point arithmetic unit to perform 1-bit, 8-bit, or 16-bit integer arithmetic. Each subprocessor can emulate one or more neurons and multiple chips can be used together.

The processors must be programmed to execute a given NNW algorithm. The CNAPS tools include CNAP-C (a C-compiler and debugger), Quicklib (hand-coded standard functions callable form CNAPS-C), BuildNet (pre-coded neural network algorithms), and CodeNet (assembly language debugger.)

Further related tools are AAC Neural Network Processor (NNP) and Neural Network Tool (NNT) [9].

B. Research projects

According to Research group Rosenstiel at the University of Tübingen [10], the critical time consumed by SOM is the computation of the the distance for all the neurons. Therefore, a neural coprocessor is designed on a Weaver-prototyping-system which calculates the distances of all neurons and finds rapidly the position of the winning neuron.

The working group of Prof. Dr. H. G. Purwins at the Institute of Applied Physics of Westfälische Wilhelms-Universität [11] is concerned with the investigation of special physical phenomena, which can also be used as a coupling medium for implementing neural networks, and particularly the SOM.

III. Software realization of the selected Kohonen algorithm

First, the algorithm of Kohonen's Self-Organizing Map has been implemented in software, more specifically in the C programming language. This section discusses this software implementation, which is going to be the starting point for the system-level design of Section IV.

The input of the software are the training patterns given in a text file; after running the learning algorithm it stores the resulting weight vectors also in a text file.

A. Program structure

In our approach we only deal with the learning phase of the SOM (after learning the algorithm only becomes simpler since the weights need not be modified), therefore we only analyze the `Learning()` function. Initialization and output generation is not interesting either, because it is not performance critical.

The function `Learning()` iteratively calls the functions `WhoWins()` and `ModifyWeights()` for every training pattern. The former function determines the winning neuron by calculating the Euclidean distance between each cell of the SOM and the given training pattern (using the function `CompareDistance()`). The latter function updates the weight vector of the winning neuron as well as that of its neighboring cells.

A.1 Neighborhood definition

The main feature that differentiates Kohonen's approach from other clustering techniques is that it creates an *ordered* map. This is due to the placement of the neurons on a map and to the consideration of neighborhood around the winning neuron. This determines the weight vectors to be updated in each iteration. We declared the neurons lying in a rectangle around the wining neuron to be modified in each iteration. The size of the rectangle decreases with time.

A.2 Weight modification

Another question related to the neighborhood definition is the parameter to be used in the weight modification of the neurons. In our method the weight update strategy in the neighborhood of the winning neuron is:

```
w(t+1)=w(t)+Alpha(t,dx,dy)*(x-w(t))
```

where `t` has the value of the current iteration, `x` has the value of the current pattern and `dx` and `dy` are the coordinates of the difference vector between the winning neuron and the neuron to be modified. `Alpha()` decreases with time and with the distance on the map.

B. Performance analysis of the software

B.1 Methodology

Since hardware is usually much more costly than software, it is logical to move only those operations into hardware which are

- simple, *i.e.* consist of relatively few elementary operations; and
- performance critical, which usually means that they are invoked very often.

So the first task is to find the optimal candidates to be put into hardware according to these two criteria. For this purpose, we used the profiler `gprof` [12] to extract that information. The results can be seen in Figure 1. Blocks in this calling graph represent functions, the edges indicate function calls. The edges are marked with numbers indicating the (relative) amount of time spent in the called function (including the time spent in all of the functions called from this function). The root of the calling graph is the function `Learning`, its execution time is taken to be 100%. All other execution time values are relative to the execution time of `Learning`.

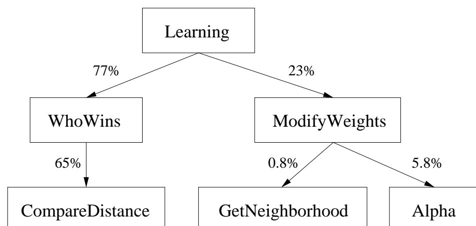

Fig. 1. Calling graph of the software

B.2 Quantitative assumptions

For the purposes of this empirical evaluation, a 'typical' situation has to be considered. That is, some assumptions have to be made concerning the parameters. Based on our former experience and the review of the literature, the following assumptions seem to be appropriate:

1. The size of the map is between 4x4 and 32x32; a typical value is 10x10.
2. The dimension of the vectors is between 4 and 64; a typical value is 9.
3. The number of iterations is between 10^3 and 10^5; a typical value is 1000.

In our empirical analysis, we made several measurements in the above range. We found that the proportion of time spent in the respective routines is quite independent of the particular parameters (at least in this range). So we could simply use the above typical values for profiling the software.

For the measurements, we used a PC with Intel Pentium II Celeron processor running at 433 MHz with 128 KB cache and 64 MB RAM (865 bogomips). The operating system was Linux 2.2.16 with `gcc` version 2.95.2. One clock cycle is about 2.3ns.

B.3 Results

According to Figure 1, the function `Learning` can be divided into two parts:

1. `WhoWins` (also including `CompareDistance`) is responsible for 77% of the execution time. One execution of this function takes about 40000 ns, which means approximately 17000 clock cycles. The function `WhoWins` calls the function `CompareDistance` `CompareDistance` is responsible for 65% of the execution time. On the other hand, one execution of this function takes only about 300 ns, which means approximately 130 clock cycles.
2. The function `ModifyWeights` modifies the weights of the winning neuron and of its neighbors. It takes 23% of the execution time (9000 ns or 4000 clock cycles each execution) and contains two subfunctions. `GetNeighborhood` determines the neurons to be modified around the winning neuron. Function `Alpha` calculates the modification factor for a given neuron.

As can be seen from the above data, the best candidate for hardware acceleration is `CompareDistance`, because it is a relatively simple function that nevertheless takes 65% of the execution time. Note though that if `CompareDistance` could be accelerated with an order of magnitude, this would not accelerate the whole process with an order of magnitude, but only to 35%+6.5%=41.5% of the original execution time. That is, if a higher factor of acceleration is to be achieved, other functions have to be transferred to hardware as well. Additional candidates for this are: `WhoWins`, `Alpha` and `ModifyWeights`.

IV. System-level design for the SOM

A. Introduction

As discussed in section I, one of the most crucial tasks of system-level design is the partitioning problem, *i.e.* deciding which system components should be realized in hardware and which ones in software.

A possible approach to this problem is the one followed in this project: we start from the cheapest, but slowest extreme, *i.e.* a software implementation. After analyzing the performance of the software (profiling, see section III), the component(s) with the highest performance impact is (are) transferred into hardware. This process is continued iteratively, until the system meets the pre-defined performance

requirements. This way, a solution with minimum cost can be found.

The components that are decided to be transferred to hardware undergo a hardware design and synthesis process, using high-level synthesis. In particular, we used the HLS tool PIPE (introduced in section I) for this purpose.

We will assume throughout the following sections these technology conditions:
- As elementary operations, simple arithmetic and logic operations can be used.
- As processing units, ALUs (Arithmetic & Logic Unit) are used. There is no difference between the ALUs; each ALU is capable of executing any arithmetic or logic elementary operation.
- The cost of a hardware solution is measured by the number of the necessary ALUs.

Somewhat less straight-forward is the determination of the execution time of the elementary operations, since it should be comparable with the times of the software realization. We assume for the ALUs the same performance as in the case of the PC. The resulting execution time values are presented in Table I. These are actually the restart time values of the corresponding pipeline in the processor of the PC [13]. If custom VLSI technology is assumed as target technology, than the same execution time can really be achieved. If other technology (*e.g.* FPGA) is used, then the exact execution time values are of course different; however, this is unimportant from the point of view of PIPE, for which only the proportions of the execution time values are of importance.

Operation	Execution time
add	1
sub	1
mul	2
div	2
cmp	1

TABLE I
EXECUTION TIME VALUES MEASURED IN CLOCKCYCLES FOR THE ELEMENTARY OPERATIONS

B. *Hardware acceleration for the calculation of the Euclidian distance*

As it turns out from the analysis of section III, the most logical choice for hardware acceleration is the function `CompareDistance`, because this function alone is responsible for 65% of the execution time. It is also important to note that this function is rather simple which makes a hardware solution even more favorable.

`CompareDistance` takes as input the weights of a neuron `ne` (which is an array of numbers of dimension `nb_inputs` that is in our example 9), the new `pattern` (also an array of the same dimension) and a reference to a number (`mindist`) which is the minimal distance found so far. The function essentially consists of a loop of at most `nb_inputs` iterations, in which the (squared) distance of `ne` and `pattern` is calculated (`dist`).

At the end of the function it is investigated whether `dist` is smaller than the previously found minimum. If it is, then `mindist` is updated accordingly and the function returns 1. Otherwise, `mindist` is unaltered and the function returns 0.

Analyzing the performance of this function leads to the conclusion that emphasis should be laid on accelerating the first part of the function, *i.e.* the loop that computes the distance, because the rest is hardly time-consuming. Fortunately, this loop is not recursive, that is, the calculation in the $i+1$. iteration does not depend on the results of the i. iteration (or any former iterations). This way, the loop can be fully parallelized if necessary.

B.1 Elementary operation graph

It is assumed that `nb_inputs` is 9, so that the hardware unit for the calculation of the Euclidian distance (called `Compute_distance`) will receive as input 2x9 numbers denoted by $a_1, \ldots, a_9$ and $b_1, \ldots, b_9$. The result is the squared distance of the two vectors. The corresponding EOG can be seen in Figure 2.

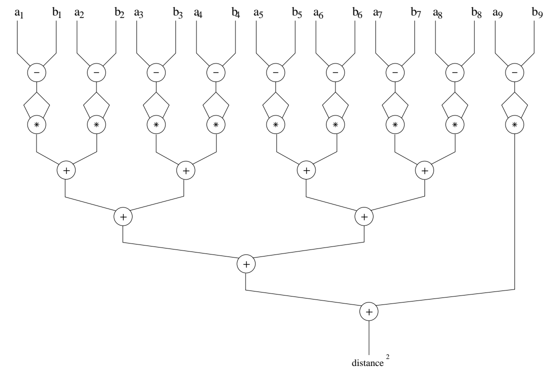

Fig. 2. Elementary operation graph of `Compute_distance`

Note that this is only a data flow graph, *i.e.* it is not sure that all the operations that are shown to be parallel will really be parallel in the realization. This graph only specifies the possibilities for parallelism, and PIPE will try to take into account as much as possible of them, but also trying to reduce the restart time and the cost of the unit. This way, a suboptimal implementation caused by inappropriate human decisions can be prevented.

After having constructed the elementary operation graph of the `Compute_distance` unit, this graph has to be specified in the input format of PIPE. The code listing in Table II shows this specification.

Most of this listing should be obvious to understand when interpreted in conjunction with Figure 2. The only line that needs some explanation is the following:

```
#    map mul ALU add ALU sub ALU
```

The effect of this line is that all of the used elementary operations (which are in this case `add`, `sub` and `mul`) are mapped to ALUs. That is, PIPE will not assume separate

```
#    type      add 1
#    type      sub 1
#    type      mul 2
#    type      SYSTEM 0

#    map mul ALU add ALU sub ALU

     IN        SYSTEM

##   subtract INi+1 from INi
     sub1      sub       IN IN
     sub2      sub       IN IN
     sub3      sub       IN IN
     sub4      sub       IN IN
     sub5      sub       IN IN
     sub6      sub       IN IN
     sub7      sub       IN IN
     sub8      sub       IN IN
     sub9      sub       IN IN

##   squaring the differences
     mul1      mul       sub1 sub1
     mul2      mul       sub2 sub2
     mul3      mul       sub3 sub3
     mul4      mul       sub4 sub4
     mul5      mul       sub5 sub5
     mul6      mul       sub6 sub6
     mul7      mul       sub7 sub7
     mul8      mul       sub8 sub8
     mul9      mul       sub9 sub9

##   building the square of the distance
     add1      add       mul1 mul2
     add2      add       mul3 mul4
     add3      add       mul5 mul6
     add4      add       mul7 mul8
     add5      add       add1 add2
     add6      add       add3 add4
     add7      add       add5 add6

##   return the square of the distance
     out       add       add7 mul9

##   system ports
#    in        IN
#    out       out
```

TABLE II

The input of PIPE for the Compute_distance algorithm

processing units for the three kinds of elementary operations but rather a single type of processing units (ALUs) that can realize all of the elementary operation types.

B.2 Results

We have run PIPE with the elementary operation graph presented in subsection IV-B.1. We made several experiments using different restart time and latency values to get various cost/speed solutions. Of course the restart time is superior to the latency in importance so we visualize the number of necessary processing units (ALUs) as a function of the restart time, and indicate the corresponding best (*i.e.* lowest) latency values only as a number near the dot (see Figure 3).

As expected, the number of necessary ALUs increases when the restart time is decreased. This is natural because a faster hardware solution is obviously more costly. This figure is quite informative because it shows the possible compromises between cost and performance. If the exact criteria are known, it can easily be decided which solution is best. Moreover, if the criteria change over time, it is easy to switch to another solution, without repeating the whole design process.

In the following, we will conduct further analysis on a couple of specific solutions (*i.e.* specific restart time, latency and cost) to demonstrate the output of PIPE. The dots corresponding to these solutions are marked with a box in Figure 3.

The results of PIPE for these specific cases are visualized in Figure 4. The ALUs are represented by vertical tracks. The vertical axis shows the time cycles from 0 to R, folded according to the restart period. The bars in the vertical tracks represent the allocated elementary operations. In the first subfigure of each pair, the length of the bars corresponds to the execution time of the operation; in the second subfigure it corresponds to its busy time. (The amount of time an EO occupies a processor is called busy time. This is not equal to the duration of the EO. The busy time of an EO can be calculated as its duration plus either the duration of its longest immediate successor or plus 1 if such does not exist.)

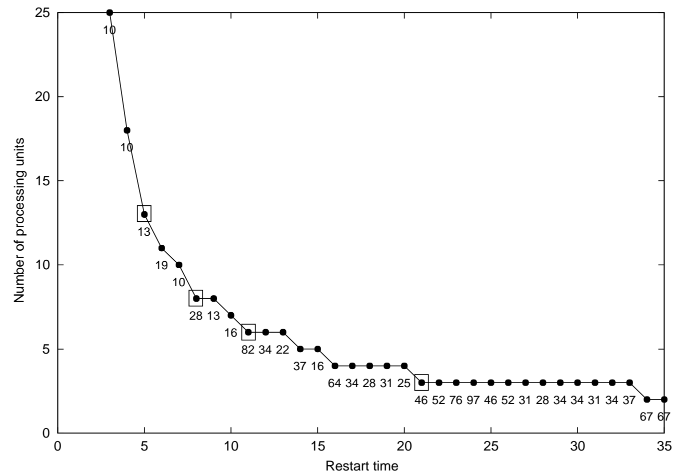

Fig. 3. The number of necessary ALUs for different restart time values. The numbers near the dots indicate the lowest corresponding latency value

It can clearly be seen from these figures that PIPE has indeed found an optimal solution. Especially the figures showing the busy times indicate explicitly that the resource usage is optimal, which means that the optimization stages of PIPE (scheduling and allocation) were very successful.

The first pair of figures show a very fast, but quite expensive solution. In this case, the restart time is 5 which means that (after a short loading time of the pipeline) the system can provide a new output every 5 clock cycles. However, this requires 13 ALUs.

The other cases show some more combinations of increasing restart time and decreasing cost. In the last combination, the restart time is already 21, but this is compensated for by a cost of only 3 ALUs.

Remember that the execution of `Compute_distance` takes on the PC about 130 clock cycles. The above solutions take 5 to 21 clock cycles. This means an acceleration factor of 6 to 26. If performance is much more important than cost then even a solution with restart time of 3 can be achieved which means an acceleration factor of 43.

C. Work in progress

The current work focuses on the hardware acceleration of the other function blocks of the software, with special emphasis on the `ModifyWeights` algorithm. In the first approach a smaller part of that function, the role of which

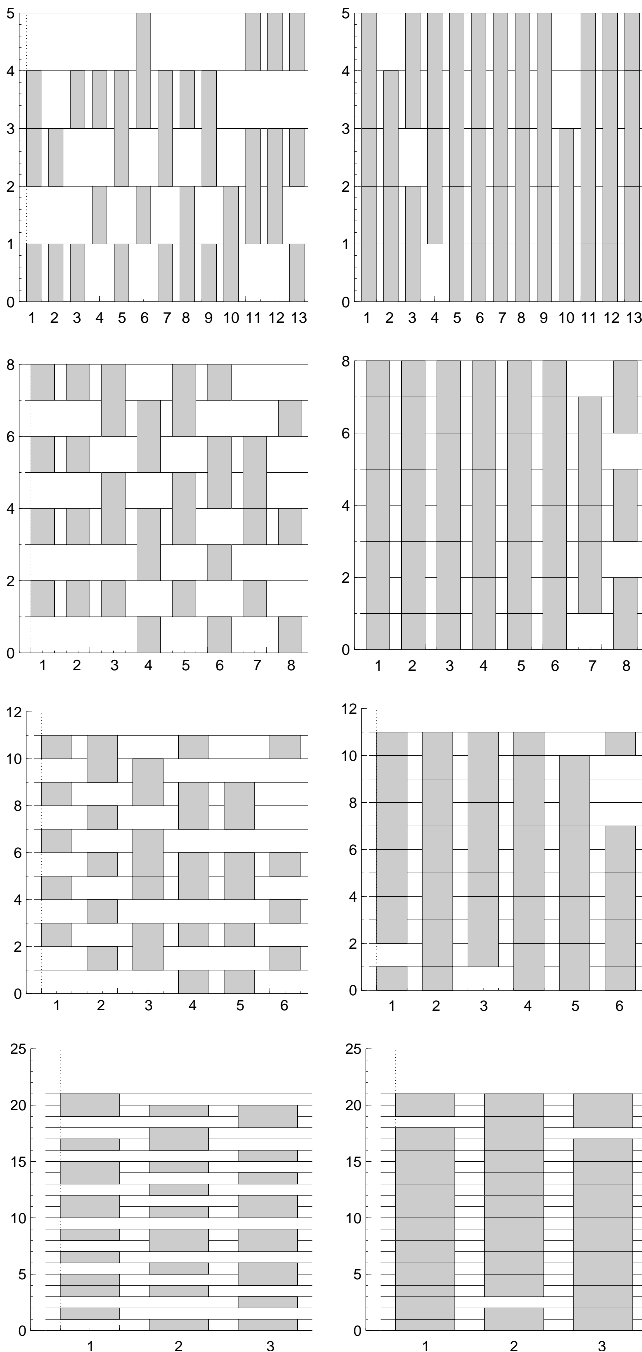

Fig. 4. Allocation results for `Compute_distance`. The pairs correspond to the settings $R = 5, L = 13$; $R = 8, L = 28$; $R = 11, L = 82$; $R = 21, L = 46$, respectively.

is to modify the weights of one neuron only, seems to be worth realizing in hardware.

Another important aspect that needs further elaboration is the communication cost between the software and hardware contexts.

V. Conclusion

The object of the paper was to present the most important principles and methods of system-level synthesis, and show their applicability on the selected problem: Kohonen's self-organizing map.

In Section I the theoretical basis of the used design approach has been explained. It mostly relies on the former results in the field of high-level synthesis: SLS can be regarded as an extension of HLS.

Then we have focused on the hardware/software realization of the SOM algorithm using the SLS methods described before. First the software realization has been presented in Section III, as the cheapest but slowest extreme. Besides, a detailed analysis has been given of the structure and the performance characteristics of the software using a profiling tool. Based on this analysis, the most performance-critical part was selected for possible hardware acceleration.

The hardware re-design of the most performance-critical function by applying SLS techniques was presented in Section IV. Starting from the functional description of the problem, a VHDL-equivalent hardware description structure could be achieved by the utilization of the design tool PIPE. One of PIPE's important features is that it proposes different solutions for different cost/speed trade-offs. The hardware re-design resulted in a possible acceleration of more than an order of magnitude. This acceleration is not caused by assuming faster hardware processing units than the one used to execute the software, but by the design optimized specifically for this particular algorithm.

VI. Acknowledgments

The work of the authors was supported by the grant OTKA T 030178, by the grant of Timber Hill LLC and the PRCH Student Science Foundation.

References

[1] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological cybernetics*, vol. 43, pp. 59–69, 1982.

[2] Péter Arató, Tamás Visegrády, and István Jankovits, *High-Level Synthesis of Pipelined Datapaths*, John Wiley & Sons, Chichester, United Kingdom, first edition, 2001.

[3] Ahmed A. Jerraya, M. Romdhani, C. Valderrama, Ph. Le Marrec, F. Hessel, G. Marchioro, and J. Daveau, "Models and languages for system-level specification and design," in *NATO ASI on System-Level Synthesis, Proceedings*, 1998.

[4] D. Gajski., *High-Level Synthesis*, Kluwer Academic Publishers, 1992.

[5] R. Camposano, "From behaviour to structure: high-level synthesis," *IEEE Design and Test of Computers*, vol. 10, pp. 8–19, 1990.

[6] Z. Á. Mann and A. Orbán, "Optimization problems in system-level synthesis," Submitted for the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications, 2002.

[7] T. Becher, W. Eppler, T. Fischer, H. Gemmeke, and G. Kock, "The MIND-project: building, applying and speeding-up neural networks using the SAND-neuroprocessor," in *Proceedings of the 5th European Congress on Intelligent Techniques and Soft Computing*, 1997.

[8] "The NeuroLution system," *http://www.first.gmd.de/connect/neurolution.html*.

[9] "Neural network hardware," *http://map.web.cern.ch/NeuralNets/nnwInHepHard.html*, 1998.

[10] X. Fang, P. Thole, J. Göppert, and W. Rosenstiel, "A hardware supported system for a special online application of self-organizing map," in *Proc. of ICNN'96*, 1996.

[11] "Homepage Gruppe Purwins," *http://cgi.uni-muenster.de:8410/exec/Physik/ap_puframe.pl?Titel;MenuF;halbleit/neurohardware*.

[12] Free Software Foundation, "GNU gprof manual," *http://www.gnu.org/manual/gprof-2.9.1/gprof.html*, 1998.

[13] A. Fog, "How to optimize for the pentium family of microprocessors, chapter 17: Out-of-order execution," *http://fatphil.org/x86/pentopt/17.html*, 2000.