Efficient Fuzzy Compiler for SIMD Architectures

Enrique Frías-Martínez[1], Julio Gutiérrez-Ríos[2] and Felipe Fernández-Hernández[2]

[1]Computer Science Department, Courant Institute of Mathematical Sciences
New York University
715 Broadway, New York, NY 10003 USA

[2]Dpto. de Tecnología Fotónica, Facultad de Informática
Universidad Politécnica de Madrid
28660 Boadilla del Monte, Madrid, Spain
`jgr@dtf.fi.upm.es`

Abstract. This paper presents a real-time full-programmable fuzzy compiler based on piecewise linear interpolation techniques designed to be executed in SIMD (Single Instruction Multiple Data) architectures. A full-programmable fuzzy processor is defined as a system where the set of rules, the membership functions, the t-norm, the t-conorm, the aggregation operator, the propagation operator, and the defuzzyfication algorithm can be defined by any valid algorithm. The SIMD platforms selected are the Intel Pentium III (using the SSE set of instructions) and the Texas Instruments TI DSP C6x family. The final speed obtained in both implementations is highly satisfactory and better than the speed provided by standard specific hardware.

1 Introduction

Fuzzy logic has been successfully introduced in a wide range of applications, ranging from classical control systems to decision support systems like nuclear plant supervision [12], medical applications [40] or automotive applications [7]. A wide variety of fuzzy logic applications are described in [1,17]. Nevertheless, two are the main drawbacks of fuzzy logic solutions: (1) the processing speed and (2) the limited programmable capabilities of both the specific platforms and the design environments.

The need to process fuzzy knowledge base systems with high speed resulted in the development of fuzzy hardware architectures. The first developments were done in the mid-80's by Togai [34] using a digital architecture, and by Yamakawa [36] using analog techniques. Before that, other ASICs designed to process fuzzy knowledge bases with high speed where developed [6,25]. During the mid-90's due to the advance in the performance of standard architectures, it became possible to use them as a platform for high speed fuzzy processing. In order to obtain the maximum performance possible in standard architectures fuzzy compilers where developed [23,39]. These compilers are based in the idea that the fuzzy syntax of the rules is a useful way of representing knowledge, and the fuzzy algorithm is a useful way of processing it, but they are not the best way of processing a fuzzy logic system in a standard architecture. In order to process fuzzy systems with high speed in standard architectures it is needed a compilation from the fuzzy system syntax to a syntax suitable for a standard architecture.

The design characteristics of a fuzzy system are application dependent. Different applications will require different T-norm, different membership functions, or different defuzzyfication algorithms**.** Besides, the same problem can be solved by different designers using fuzzy systems with different characteristics (different membership functions, different inference mechanism, or different defuzzyfication algorithms, for example). Nevertheless, a common characteristic to almost all fuzzy coprocessors is that only the set of rules and the membership functions of the system can be defined, while the fuzzy algorithm is implemented by hardware, and can not be programmed. This problem is also present in fuzzy compilers; they are designed for a specific fuzzy system (usually Takagi-Sugeno (TS) [22,39]) and, again, only rules and membership functions can be defined. Some fuzzy logic IDE (Integrated Development Environments) do allow programmable capabilities, but the final inference mechanism obtained does not provide high-speed processing.

In this paper a full-programmable fuzzy system is defined as a system where the rules, membership functions, the t-norm, the t-conorm, the propagation operator, the aggregation operator, and the defuzzyfication algorithm can be defined. A full-programmable fuzzy compiler will allow to execute any kind of application using standard hardware.

The platforms selected for the implementation of the proposed compiler are Intel Pentium III [16] and the Texas Instruments DSP C6x family (C6201 [30] and C6701 [30]). These two families are, respectively, good examples of a microprocessor and of a digital signal processor architecture, which are the typical platforms for the execution of fuzzy logic systems. Also both platforms have SIMD architectures which make them ideal to execute the proposed compiler.

In the rest of the paper, first the objectives and the related work are described. Next a full-programmable fuzzy compiler and its implementation on SIMD architectures is introduced. In the last section, the characteristics of the controller implemented are compared with other implementations.

2 Objectives and Related Work

The main objective is the design of a high performance, full programmable compiler for fuzzy systems. This goal makes the model to be involved in two fundamental aspects: first, full programmability and, second, high performance in standard architectures.

Full programmability means to be able to define absolutely all the processes that a fuzzy controller is able to carry out. That definition consists on determining the algorithm that characterizes each process. The programmable characteristics should be: (1) membership functions (any kind of membership function should be possible with the only constrain of being fuzzy numbers), (2) T-norm and T-conorm, (3) propagation operator, (4) aggregation operator and (5) defuzzyfication algorithm.

On the other hand, the term high performance is a time-varying concept depending on the current technological state: nowadays, we may assume that it is acceptable to have an execution speed of the order of tenths of MFLIPS. With a response time in the order of nanoseconds, it is ensured the possibility to serve a majority of applications, as shown in [4].

Some authors have already presented the idea of using a compiler to execute a fuzzy system. In [22,23] a compiler is presented, based on interpolation for TS systems with membership functions with an overlap factor of two. The ASIC implementation of this compiler achieves 2 MFLIPS. In [39] it is presented a coprocessor, the FZP-0401A, based on the compilation of the knowledge base also for TS systems that has a processing speed of 0.48 MFLIPS.

Although the concept of compiler has already been used, as far as we know, no compiler has been developed for full-programmable fuzzy systems. The development of such a compiler can be done using the concept of Approximate Fuzzy Compilation.

3 Approximate Fuzzy Compilation

Standard architectures are not adequate for high performance fuzzy processing, because their structure has been designed for numerical applications and, tough fuzzy inference is usually made in a digital way by fuzzy coprocessors, the nature of the treated information is not essentially numerical. Consequently, our purpose is to introduce a previous compilation of the original fuzzy system in order to adapt it to the operations usually made by a high performance standard processor.

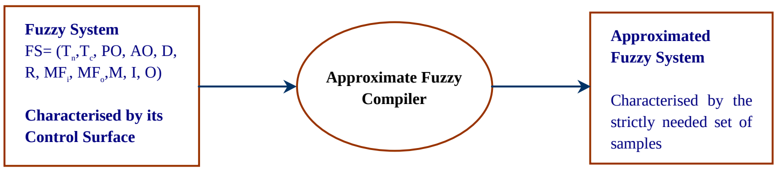
Fuzzy System
FS= (T_n,T_c, PO, AO, D, R, MF_i, MF_o,M, I, O)
Characterised by its Control Surface
Approximate Fuzzy Compiler
Approximated Fuzzy System
Characterised by the strictly needed set of samples

Fig. 1: Concept of Approximate Fuzzy Compilation.

As shown in Fig. 1, the compiler starts from the specification of an n-dimensional fuzzy system (*FS*), defined by the vector:

$$FS=(T_n,T_c,PO,AO,D,R\ ,MF_i,MF_o,M,I,O) \qquad (1)$$

where T_n is the T-norm of the system, T_c is the T-conorm, *PO* is the Propagation Operator, *AO* is the Agregation Operator, *D* is the Defuzzyfication algorithm, *R* is the set of Rules of the system, MF_i are the set of Membership Functions of each input ,

$i=1,...,n$, defined as fuzzy numbers, MF_o are the Membership Functions of the linguistic labels defined on each output, $I=(I_1,\ldots,I_n)$ is the vector of inputs of the system, and $O=(O_1,\ldots,O_n)$ is the vector of the outputs of the system.

The compiler extracts an Approximated Fuzzy System (AFS) characterised by mathematical structures able to make a fast evaluation of the system.

The control surface (CS) is the output of the system for any input values inside the universe of discourse. The main condition that must fulfil a fuzzy model is the ability to generate a control surface CS' such that,

$$CS' \approx CS \qquad (2)$$

Indeed, what becomes important under the point of view of a fuzzy model is to behave as close as possible to the fuzzy system. Although the output of the approximate system is not exactly the same, this is not relevant since a fuzzy controller is not conceived as a precise system. The designer determines the output of the system only in a reduced set of well-selected samples and the rest of the output is obtained as a coherent mix of them. Taking into account these ideas, the compiler must keep the value of the system in the points given by the designer in the fuzzy knowledge base, but can obtain an approximation for the rest of the points for which no value was specified. The compiler will preserve the certainty of the designer in the Approximate Fuzzy System obtained.

The concept of certainty in this context is related with the design of the fuzzy membership functions. For any given membership function, the kernel represents the set of points for which the designer has a complete certainty of the output of the system. The compiler will preserve the output value of these inputs. For the rest of the points, where the membership function goes from 0 to 1 or vice-versa, the designer has only a partial certainty of the output of the system. In these points the compiler can produce an approximation.

In fact, many fuzzy systems are *type-2* [19,21] fuzzy logic systems (FS). In these kind of systems uncertainty is modelled using the third dimension of *type-2* fuzzy sets. A *type-2 FS* is able to consider uncertainties about measurements, rule labels of antecedents and consequents, fuzzy logical operators in use, etc. When all uncertainties disappear, then a *type-2 FS* reduces to a *type-1 FS*. Although a complete theory of *type-2* fuzzy logic systems exists for *general type-2* fuzzy sets, it is only for *interval type-2* fuzzy sets that type-2 *FS*s are commonly practical. This paper uses this uncertainty as a positive factor to reduce the complexity of the corresponding approximate fuzzy algorithms.

More formally, the control surface *CS* of an interval *type-2 FS* can be modeled by an interval function *F*:

$$F\colon \mathrm{x} \rightarrow (z_1, \mathrm{z}_2) \qquad (3)$$

where x is the multivariate input of the MISO system considered and (z_1, z_2) is the corresponding output interval.

To implement this interval fuzzy function *F* an approximation function F' is defined*:*

$$F': \mathrm{x} \rightarrow \mathrm{z}_0 \qquad (4)$$

such that $z_0 \in (z_1, \mathrm{z}_2)$. This approximation is good enough in many practical applications. The main advantage of this consideration is that some functions *F´* can be modeled in many circumstances with a much more simple structure than the original fuzzy function.

A very simple approximator to implement is the multilinear interpolator that is well-known in finite elements analysis and computer graphics. Multilinear interpolators are simple extensions of linear ones to the multidimensional case, and are defined by a sequence of linear interpolations.

These multilinear interpolators can also be specified in a fuzzy way by means of a product-sum TS fuzzy system with triangular fuzzy partitions on the antecedents domain [8,18]. Moreover, it is possible to use the referred characteristic points of the specified antecedent partitions: interval corners of support and core of the corresponding antecedents, to directly specify each initial approximate triangular fuzzy partition. The corresponding domain points can also be considered such a suitable nonuniform sampling of the specified fuzzy rule system, and the values of output function on these characteristic grid points are used to define the approximate multivariate TS system considered:

$$\{R\ i\text{: If x is } Ni \text{ then z is } zi\} \qquad (5)$$

where *Ni* is the corresponding multivariate second order pyramidal spline and *zi* is the corresponding output function in the multidimensional sampling point *i*.

The next section presents the derived model to efficiently compute this TS system in a standard processor architecture.

4 High-Performance Full-Programmable Fuzzy Model

The developed Fuzzy Model has been structured in two steps, the first one compilation and the second one, execution. The compilation step transforms the original fuzzy system (*FS*) into an equivalent approximate fuzzy system (*AFS*) capable of being executed in real time in a standard architecture and keeping the relevant information of the original fuzzy system. The execution step describes how the output is computed using the information obtained in the compilation step.

4.1 Full-Programmable Fuzzy Compiler

Let a fuzzy system *FS* be defined by a vector as stated in (1).

4.1.1 Definitions of some useful functions and spaces

In this section a set of functions and spaces are defined in order to make easier the description of the model:

- $F_{or}(V)$. Let $V=(A_1,..,A_p)$ be a vector with $A_i \in \boldsymbol{R}$. $F_{or}(V)$ sorts the components A_i in ascendant order, obtaining the output $V' = \{A'_1,..,A'_p\}$ with $A'_1<...<A'_p$.

- $F_e(V)$. Let $V=(A_1,..,A_p)$ be a vector with $A_i \in \boldsymbol{R}$. $F_e(V)$ takes out the components of V which are repeated and generates a new vector $V' = (A'_1,..,A'_l)$ with $A'_1 \neq ... \neq A'_l$.

- $F_i(V)$. Let $V=(A_1,..,A_p)$ be a vector with $A_i \in \boldsymbol{R}$. $F_i(V)$ obtains the intervals of vector V:

$$F_i(A_1,..,A_p)=([A_1,A_2),..,[A_{p-1},A_p)). \qquad (6)$$

- Card(A). Gives the cardinality of the parameter A.
- S. Is the input space of the fuzzy system FS.
- $\text{Kernel}_e(A)$. Obtains the two extreme points that define the kernel of A.
- $\text{Supp}_e(A)$. Obtains the two extreme points that define the support of A.

4.1.2 Partition of the input space S

Rule activation depends on the region in which the inputs are included. Consequently, it is important to make a coherent partition of the input space, accordingly to the position of the linguistic labels.

We define the *Activation Points* of the input I_i, AP_i, with $i=1,..,n$, as the set of points determined by the extreme points of the kernels and supports of each membership function of I_i, in ascendant order. Using the functions defined in 4.1.1, we can express the *Activation Points* as:

$$AP_i = F_e\left(F_{or}\left(\bigcup_{k=1}^{Card(MF_i)}\left(Supp_e(MF_{i,k}) \cup Kernel_e(MF_{i,k})\right)\right)\right). \qquad (7)$$

The *Activation Intervals* of an input I_i, AI_i, is defined as the intervals of the *Activation Points*:

$$AI_i = F_i(AP_i). \qquad (8)$$

Let FS be a fuzzy system with activation intervals $AI = \{AI_1, \cdots, AI_n\}$ of the input variables $I=\{I_1, \cdots, I_n\}$, and S the n-dimensional input space defined by the vector I of inputs. The partition *P* of the input space *S* of the system is defined as the cartesian product of the elements of *AI*:

$$P = \bigtimes_{i=1}^{n} AI_i . \tag{9}$$

The next step of the model obtains the output of the *FS* in the vertex that define the cells of the partition *P*. Starting from these values, we construct the approximate control surface *CS*'. The number of cells of the partition *P* is given by the following expression:

$$\mathrm{Card}(P) = \prod_{i=1}^{n} \mathrm{Card}(AI_i). \tag{10}$$

Each one of the cells in *P*, P_k, is determined by 2^n vertex, being *n* the dimension of the system, *n* = Card(*I*). Calling *VP* the total number of the vertexes of *P*, we can write the following expression:

$$VP = \prod_{k=1}^{n} \mathrm{Card}(AP_i) \tag{11}$$

4.1.3 Characteristic Matrix of the fuzzy system FS

We define the *Vertex Matrix V* of a fuzzy system *FS*, as the matrix that contains the vertexes of partition *P*. The *Characteristic Matrix CM* is defined as the n-dimensional matrix containing the output of the *FS* in the vertexes of partition *P*. Being Dim_i = Card(AP_i), the number of elements of the activation points of the dimension *i*, the element $Va_{1,\cdots,}a_n$ of *V* can be written as follows:

$$V_{a_1,\cdots,a_n} = \left(AP_{1,a_1}, \cdots, AP_{n,a_n}\right). \tag{12}$$

Consequently, *V* can be expressed as the Cartesian product of the activation points AP_i of each input to the *FS*:

$$V = \bigtimes_{i=1}^{n} AP_i . \tag{13}$$

Each vertex $Va_1,\ldots,a_n$ in *V* is an identifier of a cell P_k of the partition *P*, where *k* is obtained as:

$$k = \sum_{i=n}^{1}\left((a_i - 1) \prod_{k=i+1}^{n} \mathrm{Card}(AI_k) \right) + 1, \tag{14}$$

with $a_1 = 1 \cdots \text{Card}(AP_1)-1, \ldots, a_n = 1 \cdots \text{Card}(AP_n)-1$. The cell P_k is defined by the set of vertexes VE_k,:

$$VE_k = \left(V_{a_1,a_2,\ldots,a_n}, V_{a_1+1,a_2,\ldots,a_n}, V_{a_1,a_2+1,\ldots,a_n}, \ldots, V_{a_1+1,a_2+1,\ldots,a_n+1}\right) \tag{15}$$

The *Characteristic Matrix CM* is obtained as the output of *FS* in each element of *V*:

$$CM_{a_1,a_2,\cdots;a_n} = FS\left(V_{a_1,a_2,\cdots;a_n}\right). \tag{16}$$

In order to accelerate the execution, the compilation phase defines two more functions: *Equalisation* (*E*) and *Normalisation* (*N*).

4.1.4 Equalisation of the inputs of the system

Let $I=(I_1,I_2,\ldots,I_n)$ be the vector of inputs of the fuzzy system *FS*. We define the equalisation of the dimension k ($k = 1,\ldots,n$) as the natural number indicating the activation interval AI_k in which the input I_k is found. Consequently, being $j=\text{Card}(AP_k)$ for each input I_k, the *Equalisation Function* $E_k(I_k)$ is defined as follows:

$$E_k(I_k) = \begin{cases} 1 \text{ si } I_k \in AI_{k,1} \\ 2 \text{ si } I_k \in AI_{k,2} \\ \ldots \\ j-1 \text{ si } I_k \in AI_{k,j-1} \end{cases} \tag{17}$$

Fig. 2 presents a graphical representation of the function $E_k(I_k)$.

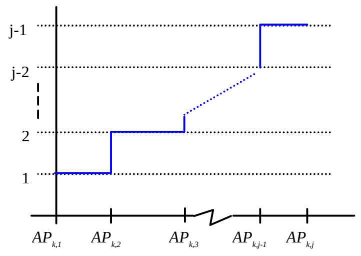

Fig. 2: Equalisation function $E_k(I_k)$

The *Equalisation Vector E* is defined as the set of all the equalisation functions:

$$E=(E_1(I_1),\ldots,E_n(I_n)). \tag{18}$$

In the same way, the *Equalisation Vector* of an input *I*, *E*(*I*), is defined as the set of values of the equalisation functions for that input:

$$E(I)=(E_1(I_1),..,E_n(I_n)) = (a_1,a_2,\cdots, a_n), \tag{19}$$

The *Equalisation Vector* of *I* identifies the cell P_k of partition *P* in which the input *I* is included, as stated in (14) and (15).

4.1.5 Normalisation Functions

The objective of the normalisation functions is to make all inputs to be normalised in the range [0,1) in each activation interval.

Each coefficient a_k of (19) points to the activation interval of I_k which is currently active. Being [*A*,*B*) the activation interval a_k of I_k, the *Normalisation Function* of I_k in the activation interval a_k is defined as follows:

$$N_{k,a_i} = \frac{I_k - A}{B - A}. \tag{20}$$

For each input I_k of the input vector *I*, there will be Card(AI_k) normalisation functions. $N_k(I_k)$ is defined as a vector containing all the normalisation functions of I_k:

$$N_k(I_k) = \left(N_{k,1}(I_k),\cdots,N_{k,Card(AI_k)}(I_k)\right). \tag{21}$$

The *Normalisation Matrix* (*N*) is defined as the set of all normalisation functions:

$$N = \left(N_1(I_1),\cdots,N_n(I_n)\right). \tag{22}$$

Finally, as it was the case for the equalisation functions, the *Normalisation Vector of the input I*, *N*(*I*), is defined as the vector obtained after applying the respective normalisation functions to each one of the dimensions of the input vector *I*:

$$N(I) = \left(N_{1,a_1}(I_1),\cdots,N_{n,a_n}(I_n)\right) \tag{23}$$

where, as stated in (19), ($a_1,a_2,\cdots, a_n$) represent both, the activation intervals of *I*, and the vertex of the cell in which the input *I* is included.

4.1.6 Approximate Fuzzy System

As a product of the compilation we obtain three items:

1. Characteristic Matrix (*CM*) described in 4.1.3, expression (16)
2. Equalisation Vector defined in (18): $E=\{E_1(I_1),..,E_n(I_n)\}$

3. Normalization Matriz defined in (22): $N = \{N_1(I_1), \ldots, N_n(I_n)\}$

The set of these three items constitutes the so called *Approximate Fuzzy System* (*AFS*) of the original Fuzzy System FS:

$$AFS = \{MC, E, N\}. \qquad (24)$$

4.2 Execution Step of the Full-programmable Fuzzy System

Once the AFS has been obtained, the first step for the computation of the output of the model is to get the relevant information. With this purpose, it is necessary to calculate the equalisation vector E(I) of the input I using (19). Using the equalisation vector E(I) as an index in the characteristic matrix CM, the outputs of FS for the vertexes of the cell Pk in which the input I is included are obtained.

We define the *Characteristic Vector* $CV(I)$ of the input I, as the set of values of the output of the *FS* on the vertexes of the cell P_k in which the input I is included:

$$CV(I) = \left(CM_{a_1,a_2,\cdots,a_n}, CM_{a_1+1,a_2,\cdots,a_n}, CM_{a_1,a_2+1,\cdots,a_n}, \cdots, CM_{a_1,a_2,\cdots,a_n+1}\right) \qquad (25)$$

The vector $(a_1, a_2, \cdots, a_n)$, has the necessary information to obtain $N(I)$ applying (23):

$$N(I) = \left(N_{1,a_1}(I_1), N_{1,a_2}(I_2), \cdots, N_{1,a_n}(I_n)\right) \qquad (26)$$

The *Approximate Fuzzy System Function* of the input I, $AFS(I)$, is defined as the computation of the *Characteristic Vector* $CV(I)$ and the *Normalisation Vector* $N(I)$ making use of the information provided by the *Approximate Fuzzy System AFS*:

$$AFS(I) = \{CV(I), N(I)\} \qquad (27)$$

The second step of the execution phase, obtains the output of the system using the data obtained in (27). Being O be the output of the model, O is obtained as:

$$O = F_O(AFS(I)) = F_O(CV(I), N(I)) \qquad (28)$$

where F_O is the *Approximation Function* that produces the output that the model provides. The *High-performance Full-programmable Fuzzy Model*, or *Model M* for short, is defined as a tuple formed by *AFS* and F_O:

$$M = \{AFS, F_O\}. \qquad (29)$$

4.2.1 Choosing the Approximation Function F_o

The *Approximation Function* F_O should verify the following characteristics:

- F_O must be defined as:

$$F_O\left(R^{2n}, R^n\right) \to R \tag{30}$$

- The result of F_O for an input I must be as near to the original FS as possible:

$$FS(I) \approx F_O\left(AFS(I)\right) \tag{31}$$

- F_O must be able to be efficiently implemented in a SIMD architecture.

The most immediate *Approximation Function* F_O that fulfills these premises is *Multi-linear Interpolation.*

4.3 Generalisation of the Model *M* to Multiple Output Systems

Let $FS=(T_n,T_c,PO,AO,D,R,MF_i,MF_o,M,I,O)$ be a multiple output (*MIMO*) fuzzy system, with $O = (O_1, \cdots, O_m)$, where m is the number of outputs. The adaptation of the Model M to the *MIMO FS* consists on considering the original FS as composed by m fuzzy systems:

$$\begin{array}{c} FS_1=(T_n,T_c,PO,AO,D,R\ ,MF_i,MF_o,M,I,O_1) \\ \cdots \\ FS_m=(T_n,T_c,PO,AO,D,R\ ,MF_i,MF_o,M,I,O_m) \end{array} \tag{32}$$

From each one of the fuzzy systems $\{FS_1, \cdots, FS_m\}$ an associated model is obtained:

$$M_1 = \left(AFS_1, F_O\right) \quad \cdots \quad M_m = \left(AFS_m, F_O\right). \tag{33}$$

Consequently, the *Model M* of the *MIMO FS* is given by the set of all the submodels:

$$M = \{M_1, \cdots, M_m\}. \tag{34}$$

4.4 Validation of the Model *M*: The Model *M* as a Takagi-Sugeno Compiler

The validity of the model M can be proved demonstrating that it produces an approximation of the control surface CS as stated in (2), and that this approximation can be made as close to the original surface as needed. The proof consists only on proving that the proposed compiler provides a Takagi-Sugeno system.

The fact that the Model *M* provides an approximation to the original *CS* of *FS* is based on the following results:

- A Zero-order Takagi-Sugeno system with triangular linguistic labels, unity partition, overlapping factor of two, product as T-norm and weighted sum as T-conorm is a *multi-linear interpolator*. This result can be found, for example, in [18].
- A Zero-order Takagi-Sugeno system with triangular linguistic labels, unity fuzzy partition, overlapping factor of two, product as T-norm and weighted sum as T-conorm, is a universal approximator. Castro [3] demonstrated that Takagi-Sugeno systems, as well as many others, are universal approximators. Similar results may be seen in [20,37,38].

Starting from those results the proof is immediate: as stated in [18] a zero order Takagi-Sugeno system with triangular linguistic labels, unity fuzzy partition, overlapping factor of two and product-sum as T-norm and T-conorm respectively, is equivalent to carrying out a multi-linear interpolation.

The Model *M* uses multilinear interpolation as the *Approximation Function* F_O in each cell P_k of the input space. Consequently, in all those cells it is possible to obtain an equivalent Takagi-Sugeno system with the previous characteristics.

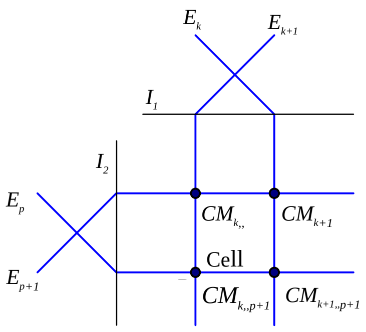

Fig. 3: Takagi-Sugeno System generated in a concrete cell.

For a system of two inputs, Fig. 3 shows a cell P_k of P with the linguistic labels of the equivalent Takagi-Sugeno system and the output of the *FS* in the vertexes of the cell. The four rules which would define the equivalent Takagi-Sugeno system are the following ones:

$$\begin{array}{l} \text{If } I_1 \text{ is } E_k \text{ and } I_2 \text{ is } E_p \text{ then } Z = CM_{k,p} \\ \text{If } I_1 \text{ is } E_{k+1} \text{ and } I_2 \text{ is } E_p \text{ then } Z = CM_{k+1,p} \\ \text{If } I_1 \text{ is } E_k \text{ and } I_2 \text{ is } E_{p+1} \text{ then } Z = CM_{k,p+1} \\ \text{If } I_1 \text{ is } E_{k+1} \text{ and } I_2 \text{ is } E_{p+1} \text{ then } Z = CM_{k+1,p+1} \end{array} \tag{35}$$

Making a generalisation of these concepts, the conclusion is that it is possible to get the equivalent Takagi-Sugeno system for all the cells derived from the compilation step of the Model *M*, using partition *P* and the characteristic matrix *CM.* The Model *M* provides a Takagi-Sugeno system which, according to [3], is a universal approximator. Then, the Model *M* is a universal approximator.

As a particular example, any zero-order Takagi-Sugeno system with triangular linguistic labels and overlapping factor of two is approximated by the Model *M* without error.

4.5 Estimations of the Model M

Once the Model *M* has been defined, it is important to evaluate the computational cost. With this purpose, the most important parameters are the amount of memory needed to store the data structures generated by the model *M,* and the execution time.

If we call *B* the number of bytes of a real number or an integer in the chosen architecture, the amount of necessary memory will be given by:

$$Mem_1 = B\prod_{i=1}^{n}\mathrm{Card}(AP_i). \tag{36}$$

In the case of having the *Equalisation* and *Normalisation Functions* digitised, they would involve tables of memory. Let p_i ($i = 1, \cdots, n$) be the number of bytes used by the sensors of every one of the inputs of the system, B_1 and B_2 the size in bytes of the equalisation and the normalisation of an input, and *n* the dimension of the system, the amount of necessary memory to keep the tables of digitisation M_{dig} would be:

$$M_{dig} = \sum_{i=1}^{n} 2^{p_i}(B_1 + B_2). \tag{37}$$

The execution time of the model *M* is highly dependent on the *Approximation Function* F_o employed. The values given in this analysis are referred to multi-linear interpolation. The number of interpolations to be made in an n-dimensional system equals 2^n-1. Consequently, being τ the processing time for a single interpolation, the total amount of time T_1 necessary to carry out the multi-linear interpolation will be:

$$T_1 = \tau(2^n - 1). \tag{38}$$

The estimation of τ may be done as the addition of the processing time of a product, an addition and a substraction:

$$\tau = \tau_{prod} + \tau_{sum} + \tau_{sub}. \tag{39}$$

In order to estimate the latency to obtain the characteristic vector *CV*(*I*) and the normalisation vector *N*(*I*), we will distinguish between discrete inference engines and non-discrete inference engines. Being *c* the evaluation time of an equalisation function, *p* the evaluation time of a normalisation function and *l* the access time to memory, the total amount of time *T* to obtain the output in a non-discrete engine is:

$$T = T_1 + nc + np + 2^n l \, . \tag{40}$$

In the case of discrete engines, the evaluation time of the equalisation and normalisation functions is reduced to a single memory access:

$$T = T_1 + nl + 2^n l \, . \tag{41}$$

However, as it will be described latter, if the architecture of the processor is suitably designed and a cache memory is installed in order to keep the last accessed values, the evaluation time of *N*(*I*) and *CV*(*I*) become negligible. In this case, the total amount of time necessary to get the output *T*, is reduced to the latency of the multilinear interpolation:

$$T = T_1 \, . \tag{42}$$

This reveals the crucial importance of the computational efficiency of the approximation function F_O.

Taking into account (36) and (37), it is easy to see that the amount of memory grows exponentially with the number *n* of dimensions of the system, and also linearly with the number of activation points of each dimension. Similarly, the execution time also grows exponentially with the number of dimensions. Thus, the number of dimensions presents a serious problem, since the model could become unfeasible. Of course, this is not an exclusive problem for the Model *M*, but for all the fuzzy models, being especially critical for real-time applications.

However, in a practical system, that kind of problems are usually avoided, since complex systems are generally designed in a tree structure of smaller sub-controllers. This makes the memory to grow linearly instead of exponentially with the dimension of the system. Concerning to execution time, the tree structure provides such a degree of parallelism that increment is linear with the dimension of the system.

5 Architecture of the Controller

The controller implemented is divided in four modules: the Sensor, which receives the input of the system and produces de Equaliaation and Linealization of that input, the Interpolation Cache, which contains also the Model Memory and the Inference Engine, which from the Interpolation Values obtained by the Interpolation Cache produces the output of the system. The architecture of these components is presented in Fig. 4.

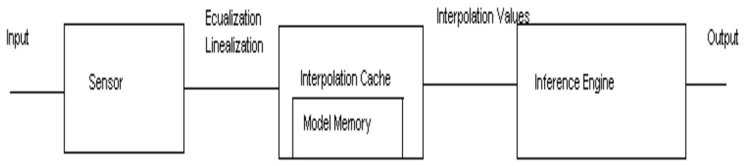

Fig. 4: Interconnection of the modules of the Controller

Sensor
The sensor implements the set of Equalization and Normalization functions, $E_k(I_k)$, $N_{k,p}(I_k)$. From the input of the system I, the sensor obtains $(a_1,...,a_n)$ and $N(I)$.

Memory Model
The Memory Model stores the Characteristic Matrix of the system, *CM*.

Interpolation Cache
The Interpolation Cache receives the equalization and the normalization of the input, $(a_1,...,a_n)$ and $N(I)$. From the equalization of the input, it obtains the Characteristic Vector *CV* accessing the Memory Model.

Due to the locality of the inputs, if an input is in a cell P_k, the most probable situation is that the next input of the system I will be in the same cell P_k. This means that the Inference Engine will have to work with the same Characteristic Vector as in the previous inference, so the Interpolation Cache does not need to access the Memory Model to obtain it. The output of the interpolation cache is the Characteristic Vector *CV* and the normalization of the input.

5.1 Inference Engine on Intel Pentium III

The inference engine receives the *Characteristic Vector CV* and the normalization of the input $N(I)$ and obtains the output O applying multilinear interpolation. Two different versions of the inference engine have been implemented for Pentium III: the first one is based on the compiler optimization capabilities and the second one uses the set of SSE instructions of the processor.

5.1.1 Inference Engine using Compiler Optimizations

This implementation directly codes the multi-lineal interpolation and uses the optimizations of the C compiler [15] to produce efficient code. In order to obtain optimum performance, *Vtune* Performance Analyzer [16] has been used. Also, some rules have been considered when developing the code:

- Unwind of the existing loops to increase the execution speed.
- To gather the calculations for a better use of the temporal values [35]. *Intel C* compiler is capable to minimise the number of memory accesses, and optimise the use of temporal variables. Because of that, the calculation of the output of the system is carried out in a single line of code. The compiler will generate the optimal structure in assembler language.

Table 1: Measurements of the model *M* in an *Intel Pentium III 1GHz*

System	Time	*MFRPS*	*MFLIPS*
2 Inputs / 1 Output	8.67 ns	5390	111
3 Inputs / 1 Output	12.98 ns	26675	77
4 Inputs / 1 Output	28.97 ns	80025	33

Table 1 presents the values of the inference engine in a *Intel Pentium III 1GHz* processor. The number of *MFRPS* (*Mega Fuzzy Rules Per Second*) has been made assuming seven linguistic labels per input and a complete rule-base.

The maximum number of clock cycles per interpolation is three. This happens in the case of two inputs systems becuase the number of operations to be made is not enough to exploit all the resources of the processor. Being *n* the number of inputs, *m* the number of outputs, and *f* the clock rate, the upper bound of the execution time *T* is:

$$T \leq \left(3f^{-1} * \left(2^{n} - 1\right)\right) \cdot m \quad \text{(ns.)} \tag{43}$$

5.1.2 Inference Engine Using the SSE instructions

The SSE set of instructions [15] of an Intel Pentium III is an excellent environment to implement a multi-linear interpolation. This can be done using the SIMD [26] architecture of the processor.

The Pentium III processor has a 128 bit register which can be accessed as a __*mm128* data type. This data type can be also viewed as four 32 bit registers. Using this structure, one operation can be done in parallel to four different data, as can be seen in Fig. 5.

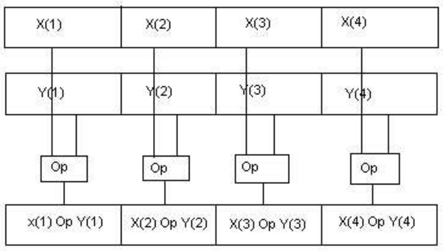

Fig. 5: SIMD data structure of a Pentium III

To access and operate with *__m128* data Intel Compiler has a set of *intrinsics* [15]. An intrinsic is a C function that directly translates into an assembler operation that can not be obtained using standard C code. Table 2 has examples of some *intrinsics*.

Table 2: Example of *Intrinsics*.

Intrinsic	Description
__m128 _mm_mul_ps (*__m128 x, __m128 y*)	Multiplies two _m128 data
__m128 __mm_add_ps(*__m128 x, __m128 y*)	Adds two _m128 data

The code presented in Fig. 6 implements four inference engines (one in each element of the SIMD vector) of two inputs and one output, using the set of SSE instructions.

```
OutputG.OutputI=_mm_add_ps(_mm_add_ps(CV_G.CV_I[0],
_mm_mul_ps(N_G.N_I[1],_mm_sub_ps(CV_G.CV_I[1],CV_G.CV_I[0])))
,_mm_mul_ps(N_G.N_I[0],_mm_sub_ps(_mm_add_ps(CV_G.CV_I[2],
_mm_mul_ps(N_G.N_I[1],_mm_sub_ps(CV_G.CV_I[3],CV_G.CV_I[2])))
,_mm_add_ps(CV_G.CV_I[0],_mm_mul_ps(N_G.N_I[1],
_mm_sub_ps(CV_G.CV_I[1],CV_G.CV_I[0]))))));
```

Fig. 6: Code of the inference engines

where *OutputG* is the output of the system, *CV_G* is the Characteristic Vector of the input, and *N_G* the normalization vector.

The feedback of the controller the controller will be given by the architecture of the implemented fuzzy system. This makes possible that the code given in 6 can implement different architectures. Some examples can be: four controllers of 2 inputs-1 output, 1 controller of 8 inputs-4 outputs, 1 controller of 5 inputs- 1 output, etc.

Each inference engine implemented in 6 has a speed of 55 MFLIPS for a 2-inputs/1-output system. This gives a total throughput of 220 MFLIPS in a Pentium III 1 GHz. (55 MFLIPS each inference system multiplied by 4 inference engines implemented). Table 3 presents the speed obtained for four inference engines of 2-inputs/1-output and four inference engines of 4-inputs/1-output in a Intel Pentium III 1 GHz.

Table 3: Speed obtained with the proposed model.

	Inference Engine	Throughput
4 Systems 2 I / 1 O	*55 MFLIPS*	*220 MFLIPS*
4 Systems 4 I / 1 O	*7.04 MFLIPS*	*28.16 MFLIPS*

5.2 Inference Engine on Texas Instruments TI DSPC6x family

As in the previous case, the inference engine receives the *Characteristic Vector CV* and the normalization of the input *N*(*I*) and obtains the output *O* applying multilinear interpolation. The inference engine has been implemented in a fixed-point DSP, the TMS320C6201 and in a floating-point DSP, the DSP TMS320C6701.

The code has been developed in C language for both implementations, and has been optimized successfully using TI compiler options. Although the code has been developed in a high level language the speed achieved is very satisfactory. This has an important advantage, the inference engine developed can be executed in any architecture with C support. The code has also been manually optimized using Code Composer Studio.

Code Composer Studio [33] (CCS) is an Integrated Development Environment (IDE) that provides a variety of tools to simplify the coding process and accelerate development time. CCS includes a debugger, an editor, a profiler, and a project manager. It also has a highly efficient optimising C compiler and an Assembly Optimiser, which allows to choose the level of performance required. The compiler enables optimising features such as instruction packing, conditional branching, *intrinsics*, in-line assembly and program-level optimisation.

The main optimisations introduced in the code are:

- Unwind of the loops [31,32] of the *Approximation Function* F_O.
- Segmentation of the operations [5,31,32]. TI *C* compiler works better when instructions are very partitioned, because it is able to optimise the multiples units to be handled, and to reach a higher degree of parallelism.

5.2.1 Inference Engine in a TMS320C6201

The TI TMS320C6201 [30] is a fixed point DSP developed by Texas Instruments, and the first of the ' C62x generation. It is designed with TexasInstrument´ sVelociTI architecture, which is an advanced Very Long Instruction Word (VLIW) architecture. The CPU core of the DSP consist on two data paths with a general purpose register and 4 functional units each path.

The performance of the inference engine in this platform is given in Table 4.

Table 4: Response time of the model *M* with a *TI TMS320C6201*

System	Clock cycles	Time (ns)	*MFRPS*	*MFLIPS*
2 Inputs / 1 Output	23	115	421,4	8.6

3 Inputs / 1 Output	45	225	1509	4.4
4 Inputs / 1 Output	126	630	3601	1.5

The estimation of number of *MFRPS* (*Mega Fuzzy Rules Per Second*) has been made assuming seven linguistic labels per input and a complete rule-base. Table 4 reveals that the maximum time to make an interpolation is 42 ns. Being *n* the number of inputs and *m* the number of outputs, an upper bound of the execution time is:

$$T \le \left(42 * \left(2^n - 1\right)\right) \cdot m \quad (\text{ns.}) \tag{44}$$

5.2.2 Inference Engine in a TMS320C6701

The TI TMS320C6701 [30] is a floating point DSP developed by Texas Instruments, and basically has the same architectural characteristics as the C62x. The performance of the inference engine in this implementation is given in Table 5.

Table 5: Response time of the model *M* with a *TI TMS320C6701*

System	Clock cycles	Time (ns)	*MFRPS*	*MFLIPS*
2 Inputs / 1 Output	30	201	245	5
3 Inputs / 1 Output	43	288	1200	3.5
4 Inputs / 1 Output	66	442,2	5282	2.2

Table 5 reveals that the maximum time to make an interpolation is 67 ns. Being *n* the number of inputs and *m* the number of outputs, an upper bound of the execution time is:

$$T \le \left(67 * \left(2^n - 1\right)\right) \cdot m \quad (\text{ns.}) \tag{45}$$

6 Features of the High-Speed Full-Programmable Fuzzy Model

Fig. 7 shows the original control surface of a two input-one output fuzzy system *FS* with Max-Min as the inference method and COG (Center Of Gravity) as the defuzzy-fication algorithm.

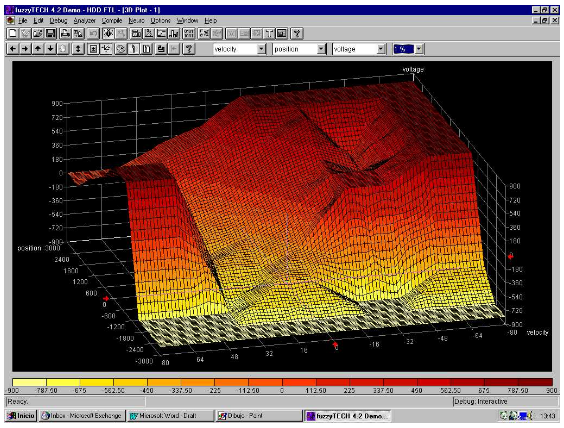

Fig. 7: Control Surface of *FS*

Both the inputs and the output have seven linguistic labels: *negative big* (*nb*), *neagtive medium* (*nm*), *negative small* (*ns*), *zero* (*z*), *positive small* (*ps*), *positive medium* (*pm*) and *positive big* (*pb*). The shapes of the membership functions are trapezoidal or triangular. The rule-base contains 49 rules.

6.1 Output comparison of the Model *M*

This section compares the output of the system *FS* of Fig. 7 with the output obtained with the approximated fuzzy system *AFS* obtained from the compilation of *FS* with the proposed Model *M*, shown in Fig. 8.

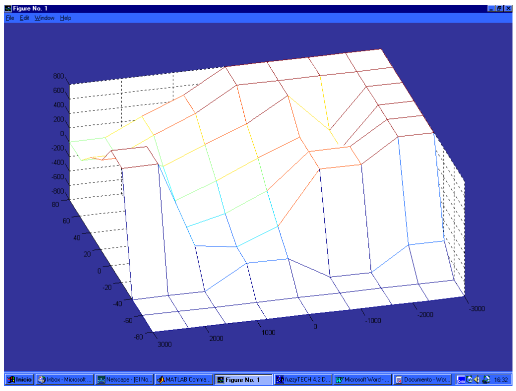

Fig. 8: Control Surfaced produced by the compilation of *FS* with Model *M*.

The values of the control surface of *FS* where obtained with the fuzzy IDE (Integrated Development Environment) *FuzzyTech* [14]. The output values of the approximated fuzzy system *AFS* were obtained executing the inference engine on a Intel Pentium III 1 GHz.

The comparison of the value of 48,682 points of both systems shows that: (1) 35% of the points of *AFS* have exactly the same value as in *FS*, (2) the deviation of the other 65% of the points is smaller than 0.5% with respect to the output range and (3) the average deviation of all the points is 0.57%.

All that, in spite of the fact that the original *FS* uses triangular membership functions to define some of the inputs. This is the worst case scenario, because triangular membership functions only have one point of certainty, and the Model *M* is designed to keep the certainty of the designer. This means that when the original systems *FS* uses trapezoidal labels, or any label whose kernel is wider than just one point, the approach produced by *AFS* would be even better.

A useful advantage is derived from using multi-linear interpolation as an approximation function of the Model *M*: the elimination of small variations in the original control surface. As a matter of fact, triangular or trapezoidal labels usually are naive representations of the real concept and they produce non desired variations in the control surface whose filtering provides benefices in control applications [9,10].

Model *M* is based on the fact that the design of *FS* is tolerant by nature. The designer knows the output he wants, only in several points of reference and the rest of the response of the system is deduced by means of a combination of the output in those points. The output of the system in the reference points is expressed by rules, and the algorithm of combination is given along with the *FS* programming. Because Model *M* generates a control surface which preserves the positions where the output is well known by the designer, and approaches the rest of the positions, the result is that model *M* preserves the certainty of the designer.

6.2 Speed comparison of the Model M

Table 6 gives the processing time of some commercial fuzzy coprocessors. WARP 2.0 [25] and SAE81C99 [6] are fuzzy commercial coprocessors of ST Microelectronics and Siemens respectively.

Table 6: Processing Time of some commercial Fuzzy Coprocessors

System	WARP 2.0	SAE81C99
3 I / 1 O	----------------	12 μs
4 I / 2 O	33.1 μs	--------------

Values of Table 6 show that Model *M* implemented in DSP's reduces the execution time of a fuzzy system, in the worst case, by a factor of fifteen. Implementation on *Intel Pentium III* 1GHz. reduces the execution time by a factor between 550 and 990, depending on the number of inputs and the considered co-processor.

Usually high-speed fuzzy processing systems are based on ASIC developments or on compilers that adapt the original fuzzy system into some representation suitable of being executed with high speed. In this section we present some of the most relevant related work in order to compare the speed achieved with the Model *M*.

Some of the most relevant solutions based on *ASICs* are:

- Project *HEPE* (*High Energy Physic Experiments*) uses fuzzy logic to compute the trajectory of particles in an accelerator. In [11] it is described the design and implementation of a fuzzy coprocessor of four inputs and one output which employs *PROD-SUM* as T-norm and T-conorm. This coprocessor obtains a processing speed of 50 *MFLIP*'s without defuzzyfication, but only 12 *MFLIP*'s including defuzzyfication.
- In [13] it is also presented a *MIN-MAX* co-processor. It makes deffuzzification by the method of centre of gravity (*COG*) which is computationally heavy. Its maximum execution speed is of 6 MFLIPS.
- The processor shown in [24] reached 10 MFLIPS using *Lukasiewicz* as T-norm, with a pipelined architecture.

Some of the most relevant solutions based on compilation of the rule-base are:

- In [39] it is described a fuzzy co-processor, *FZP-0401A* based on the compilation of the rule-base. The authors also prove that a Takagi-Sugeno *FS* is equivalent to an interpolator. This result is the base of the co-processor implemented on an *ASIC.* The speed obtained is 0.48 MFLIPS.
- Rovatti [22] developed a model of inference based on simplicial interpolation, starting from Takagi-Sugeno systems. It uses *Min-Max* as an inference method, triangular labels with degree of overlapping of two in the input, and singletons at the consequent. This model was implemented on an *ASIC* and a speed of 2 MFLIPS was obtained.

As shown in Table 1, Table 4 and Table 5, the speed obtained by Model *M* in a DSP is faster than the best part of related work, and when comparing the Pentium III implementation, the speed is remarkably improved when compared with related work. At the same time it is important to take into account that model *M* is fully programmable. Therefore, it can be adapted to any inference procedure, shape of labels or defuzzyfication method, and do not present any restriction, being able to implement any fuzzy solution.

7 Conclusions and Future Work

This paper has presented a compiler that allows to execute real-time full-programmable fuzzy systems using standard SIMD architectures.

This compiler makes it possible to introduce fuzzy logic techniques to applications that require real-time processing and more programmable capabilities. Using a standard architecture as a DSP or a microprocessor also makes possible to develop the final solution very fast and at a low cost, compared to specific fuzzy circuits. Also, a full-programmable model allows to use the same hardware and the same model for any kind of application.

The future work is divided in two main areas: (1) optimize the Model *M* in order to reduce the number of computations needed to obtain the output and (2) implement the Model *M* in new platforms that will increase the speed obtained.

Some improvements that can be made as part of the first area are the optimization of the normalization and equalization functions, especially to avoid unneeded output calculations.

The second area should explore the implementation of the Model *M* in new platforms such as TMS320C64 [27], TMS320C6211 [29] and TMS320C6711 [28] and Pentium IV. The design of an ASIC that implements the execution phase of the Model *M* could improve the performance of the system. An interesting intermediate solution between standard architectures and ASICs are CISCP (Configurable Instruction Set Processor) architectures, such as [2]. In this case the design of an interpolation instruction should improve the processing speed.

References

[1] C.Von Altrock, Fuzzy Logic&Neurofuzzy Applications in Busines, Prentice-Hall 1997

[2] ARC Cores Ltd., "CISP Spells the End of Off-the-Peg Processors" in Electronic Express Europe, Oct. 1999, pp. 22

[3] J.L. Castro, "Fuzzy Logic Controllers Are Universal Approximators" in IEEE Transactions on Systems, Man and Cybernetics, Vol. 25, No. 4, Abr. 1995, pp. 629-635

[4] A. Costa, A. de Gloria, “Hardware Solutions for Fuzzy Control”, in *Proc. of the IEEE*, Vol. 83, No 3, Mar. 1995, pp. 422-434

[5] J. Edwards, "Optimizing code for advanced DSPs" in Embedded Systems Programming Europe, Mar. 1998, pp. 8-19

[6] H. Eichfeld, "A 12b General-Purpose Fuzzy Logic Controller Chip", in *IEEE Transactions on Fuzzy Systems*, Vol 4, No. 4:460-475 ,1996

[7] D. Elting, M. Fennich, R. Kowalczyk, "Fuzzy Anti-Lock Brake System Solution" in *Intel C. Automot.. Oper Center*, http://developer.intel.com/design/mcs96/designex/2351.htm, 1998

[8] F. Fernández, J. Gutiérrez, “The FRISC Model” Technical report, Dept. de Tecnología Fotónica – Universidad Politécnica de Madrid, Nov. 1998.

[9] F. Fernández, J. Gutiérrez, J.C. Crespo, G. Triviño, “A Shape-preserving Affine Takagi-Sugeno Model based on a Piecewise Constant Nonuniform Fuzzyfication Transform” *Int. Conference WSEAS 2002*. 2002.

[10] F. Fernández, J. Gutiérrez, J.C. Crespo, G. Triviño, “Construction of Shape-preserving First-order Takagi-Sugeno Systems via Linear-rational spline Fuzzy Partitions” *4th Int. Conference on Recent Advances in Soft Computing.* Dec. 2002

[11] A. Gabrielli, E. Gandolfi, "A Fast Digital Fuzzy Processor" in IEEE Micro, Jan-Feb 1999, pp. 68-79

[12] J. Gebhardt, C. Von Altrock, "Recent Successful Fuzzy Logic Applications in Industrial Automation", in *Proceedings of the 5th IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 96*, New Orleans, Sep. 1998

[13] S. Guo, L. Peters, "A High-Speed, Reconfigurable Fuzzy Logic Controller" in IEEE Micro, Dic. 1995, pp. 1-11

[14] Inform GmbH, FuzzyTECH 4.1 Reference Manual, 2000

[15] Intel Corporation, "Intel C/C++ Compiler User's Guide, with support for the Streaming SIMD Extensions", 2000

[16] Intel Corporation, Vtune Performance Enhancement Environment, 1999

[17] M. Jamshidi, A. Titli, L.A. Zadeh, S. Boverie, Applications of Fuzzy Logic. Towards a High Machine Intelligence Quotient Systems, Prentice-Hall, 1997

[18] A. Jiménez, F. Matía, "A Fast Picewise-Linear Implementation of Fuzzy Controllers" in Proc. of the 1994 First International Joint Conference of NAFIPS-IFIS-NASA, San Antonio, TX, Dec. 1994, pp. 27-31

[19] J. Klir, B. Yuan, Fuzzy Sets and Fuzzy Logic. Theory and Applications, Prentice Hall PTR, 1995

[20] V. Kreinovich, H.T. Nguyen, Y. Yam, "Fuzzy Systemas are Universal Approximators for a Smooth Function And Its Derivatives", en Technical Report CUHK-MAE-99002, The Chinese Univ. of Hong Kong, Ene. 1999, http://citeseer.nj.nec.com/kreinovich99fuzzy.html

[21] J.M. Mendel, "Uncertainty, fuzzy logic, and signal processing", en *Signal Processing*, Volume 80, No. 6, Jun. 2000, pp. 913-933

[22] R. Rovatti, C. Fantuzzi, S. Simani, "High Speed DSP-based implementation of picewise-affine and picewise-quadratic fuzzy systems", in *Signal Processing*, Vol. 80, No 6, Jun. 2000, pp. 951-963

[23] R. Rovatti, "Linear and Fuzzy Piecewise-Linear Signal Processing with an Extended DSP Architecture", in *Proceedings of the 7th IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 98*, pp. 1082-1087

[24] L. de Salvador, Modelo de un Microprocesador Borroso de Altas Prestaciones y su Diseño, Ph.D. Thesis, Universidad Politécnica de Madrid, 1996

[25] SGS-Thomson Microelectronics, "Weight Associative Rule Processor WARP 2.0", 1994

[26] S. Thakkar, "Streaming SIMD Extension", in IEEE Computer, Dec. 1999:26-34

[27] Texas Instruments, *TMS320C64x Technical Overview*, Feb. 2000

[28] Texas Instruments, "TMS320C6711, Floating-Point Digital Signal Processor" in http://www.ti.com/sc/docs/products/dsp/tms320c6711.html, Sep. 2000

[29] Texas Instruments, "TMS320C6211, Fixed-Point Digital Signal Processor" in http://www.ti.com/sc/docs/products/dsp/tms320c6211.html, Sep. 2000

[30] Texas Instruments, TMS320C62x/C67x CPU. Reference Guide, Mar. 1998

[31] Texas Instruments, TMS320C6x Optimizing C Compiler. User' s Guide, Feb. 1998

[32] Texas Instruments, TMS320C62x/C67x. Programmer' s Guide, Feb. 1998

[33] Texas Instruments, TMX320C6000 Code Composer Studio. Tutorial, 1999

[34] M. Togai, "Expert System on A Chip: An Engine for Approximate Reasoning", in *IEEE Expert*, Vol. 1, No. 3:55-62, 1986

[35] J.H. Wolf, "Programming Methods for the Pentium III Processor' s Streaming SIMD Extensions Using the Vtune Performance Enhancement Environment" in Intel Technology Journal Q2, May. 1999, http://developer.intel.com/technology/itj/index.htm

[36] T. Yamakawa, "A simple fuzzy computer hardware system employing MIN&MAX operations", in *Proceedings of the 2nd IFSA Congress*, pp. 122-130, 1987

[37] H. Ying, "General Takagi-Sugeno Fuzzy Systems are Universal Approximators" in Proc. of the 7th IEEE International Conference on Fuzzy Systems, Anchorage, Alaska, May. 1998, pp. 819-823

[38] H. Ying, Y. Ding,. S. Li, "Typical Takagi-Sugeno and Mamdami Fuzzy Systems as Universal Approximators: Necessary Conditions and Comparison" in Proc. of the 7th IEEE International Conference on Fuzzy Systems, Anchorage, Alaska, May. 1998, pp. 824-828

[39] N. Yubazaki, M. Otani, A. Mutuo, T. Ashida, "Fuzzy inference chip FZP-0401A based on interpolation algorithm" in Fuzzy Sets and Systems, Vol. 98(3) pp. 299-310

[40] G. Zahlman and M. Scherf, "Monitoring Glaucoma by Means of a NeuroFuzzy Classifier" in *Fuzzy Logic Aplication Note series*, Inform Corp, http://www.fuzzytech.com, 1998