

Abstract—In this paper, we describe application-specific extensions for fuzzy processing to a general purpose processor. The application-specific instruction set extensions were defined and evaluated using hardware/software codesign techniques. Based on this approach, we have extended the MIPS instruction set architecture with only a few new instructions to significantly speed up fuzzy computation with no increase of the processor cycle time and with only minor increase in chip area. The processor is implemented using a reconfigurable processor core which was designed as a starting point for application-specific processor designs to be used in embedded applications. Performance is presented for three representative applications of varying complexity.

Index Terms—Application-specific instruction set, fuzzy processing, fuzzy systems, hardware/software codesign, reconfigurable processor core, subword parallelism.

I. INTRODUCTION

To provide high-performance fuzzy computation [1], [2], several special purpose fuzzy circuit implementations and processors have been proposed. However, an approach based on specialized computing engines for performing fuzzy computation does not take into account that fuzzy processing is often not the only task required of the processing unit. Fuzzy processing is often embedded into a complex system, requiring input/output (I/O) management, and other crisp operations.

A few processors can support both general purpose computing and fuzzy computing, but they are mostly available at the low-performance microcontroller level. In this paper, we show how fuzzy processing can be implemented efficiently on general purpose processors and what functionality is required to achieve peak performance.

We have extended a general purpose reduced instruction set computing (RISC) processing unit with specialized fuzzy control operations to achieve high fuzzy processing performance with only minor changes in the processor, thereby preserving general purpose computing performance. This approach to designing application-specific processor variants based on a common reconfigurable processor core allows to build systems-on-a-chip with high fuzzy processing performance. Alternatively, the extensions can be integrated into a standard microprocessor part to create product differentiation for particular markets.

To optimize tradeoffs between hardware resource utilization and fuzzy processing performance, we have employed a technique called hardware/software coevaluation [3] to evaluate different instruction sets and find the most promising in terms of performance and hardware complexity. We refer to the application-specific instructions for fuzzy processing support as MIPS-F, for MIPS with fuzzy processing support. Amongst the new instructions, only one instruction is fuzzy specific, whereas all others are also useful for general purpose programs.

To achieve maximum performance, we have used a technique called subword parallelism to pack multiple fuzzy data in a single processor word. This approach utilizes processor resources to the fullest, parallelizing multiple fuzzy inference steps and reducing memory traffic. This approach reduces both memory access penalty and power consumption, which is of major concern in embedded applications where most fuzzy computation is used.

The main contributions of this paper are: 1) the description of an architecture extension approach based on a reconfigurable processor for embedded systems and system-on-a-chip design; 2) the use of subword parallelism for fuzzy processing; and 3) a detailed performance analysis of design options for application-specific instruction extensions for fuzzy processing.

This paper is organized as follows. In Section II, we describe our approach to designing fuzzy extensions to a general purpose processor. Section III describes how the hardware/software coevaluation approach to evaluating fuzzy processor has been applied for collecting statistics, and describes all evaluated instructions. In Section IV, we describe the application-specific MIPS-F instruction set extensions. We illustrate the implementation of several typical fuzzy applications on the application-specific MIPS-F processor in Section V and evaluate its performance in Section VI. Section VII gives a brief overview of digital fuzzy logic control implementations and we draw our conclusions in Section VIII.

II. DESIGN APPROACH

Although fuzzy computation can be implemented on any general purpose processor, the resulting implementation is often inefficient because instruction set architectures (ISAs) were not designed with fuzzy computation in mind. A possible approach to improve fuzzy computation performance is to introduce instruction set primitives optimized for fuzzy processing.

To improve performance, we have explored instruction set architecture extensions for fuzzy computation to a general purpose processor. Our fuzzy processor is based on the MIPS instruction set architecture [4]. The choice of this architecture was based on a number of factors, such as the architecture’s popularity for embedded control applications, and software tool support (such as compilers, assemblers, etc.).

1063-6706/00$10.00 © 2000 IEEE
A. Mathematical Background

Previous research [5] has shown that hardware support for the minimum and maximum operations leads to substantial performance improvements. The bottleneck for further increase in fuzzy processing performance is memory access. To address this issue, Watanabe suggests the use of vector instructions to keep membership vectors for entire fuzzy sets on-chip. This approach is similar to [6], which uses long registers to store membership data for fuzzy sets.

The representations in both works is based on representing fuzzy sets as vectors of membership values for all set elements \( u_i \):

\[
F = (\mu_F(u_i)), \quad i = 1 \ldots n.
\]

Here, \( F \) is a fuzzy set in a universe of discourse \( U \) characterized by a membership function \( \mu_F: U \rightarrow [0, 1] \) and \( n \) is cardinality of \( U \). While intuitive, this representation uses long vector representations. We observe that most input values for fuzzy control are only members in a limited number of fuzzy sets and most \( \mu_F(u_i) = 0 \).

In this work, we use an alternative approach which is less resource-consuming and hence easier to integrate into existing architectures. This is based on a representation that enumerates all sets \( F_j \) in which an element \( u_i \) from the universe of discourse \( U \) is member and the associated membership degree.

\[
\text{set}_{u_i} = \{(F_j, \mu_F(u_i))\}, \quad \text{for all sets } j.
\]

We represent tuples \( (F_j, \mu_F(u_i)) \) explicitly iff \( \mu_F(u_i) \neq 0 \), the membership degrees of all nonenumerated sets default to zero.

Throughout this work, we use the following methods for fuzzy processing: for fuzzification we convert an input \( x_0 \) from a sensor into a fuzzy value \( \mu_A(x_0) \). For fuzzy reasoning we employ Mamdani’s minimum operation inference and for defuzzification we employ the center of area (COA) method [7], [8].

The intersection and union operations used in this work are defined as follows for all \( \mu_A(u_i), \mu_B(u_i) \):

- intersection \( \mu_A(u_i) \land \mu_B(u_i) = \min(\mu_A(u_i), \mu_B(u_i)) \)
- union \( \mu_A(u_i) \lor \mu_B(u_i) = \max(\mu_A(u_i), \mu_B(u_i)) \).

We have implemented fuzzy reasoning using Mamdani’s minimum operation inference rule. Assume we have two inputs from the sensors, \( x_0 \) and \( y_0 \), and two control rules of the form

- \( R_1 \): if \( x \) is \( A_1 \) and \( y \) is \( B_1 \) then \( z \) is \( C_1 \),
- \( R_2 \): if \( x \) is \( A_2 \) and \( y \) is \( B_2 \) then \( z \) is \( C_2 \).

Here, \( x, y \) and \( z \) are linguistic variables representing system state variables and the control variable and \( A_i, B_i, \) and \( C_i \) are linguistic values of the variables \( x, y, \) and \( z \) in the universes of discourse \( U, V, \) and \( W \), respectively, with \( i = 1, 2 \).

The firing strengths \( \alpha_1 \) and \( \alpha_2 \) of the rules \( R_1 \) and \( R_2 \) are

\[
\alpha_1 = \mu_{A_1}(x_0) \land \mu_{B_1}(y_0), \quad \alpha_2 = \mu_{A_2}(x_0) \land \mu_{B_2}(y_0),
\]

The rules lead to control decisions

\[
\mu_{C_1}(w) = \alpha_1 \land \mu_{C_1}(w), \quad \mu_{C_2}(w) = \alpha_2 \land \mu_{C_2}(w)
\]

where \( \mu_{C_i}(w) \) is the membership function of the fuzzy set \( C_i \) defined for all \( w \in W \).

The membership function \( \mu_C(w) \) of the implied consequence \( C \) is given by

\[
\mu_C(w) = \mu_{C_1} \lor \mu_{C_2} = \left[ \alpha_1 \land \mu_{C_1}(w) \right] \lor \left[ \alpha_2 \land \mu_{C_2}(w) \right].
\]

The control action of the system is determined through defuzzification. We employ the center of area method, which generates the center of area of probability distribution of a control action. For a discrete universe of discourse \( W \), this method yields

\[
z_0 = \frac{\sum_{j=1}^{m} \mu_C(w_j) \cdot w_j}{\sum_{j=1}^{m} \mu_C(w_j)}
\]

where \( m \) is the number of quantization levels of the output and \( z_0 \) is a crisp control action.

B. Subword Parallelism

In our approach, we pack membership degrees in multiple fuzzy sets in a single processor word. This technique is referred to as subword parallelism. Subword parallelism has lately gained widespread acceptance to support operation on multiple related data items in a single cycle for applications such as media processing, video conferencing, or multimedia and communication applications [9].

Subword parallelism can be applied to fuzzy computation, as in fuzzy calculation 8-b offer sufficient precision for representing inputs, outputs, and membership degrees of fuzzy sets [10]. Several authors use even less bits for encoding membership functions, e.g., 6 b as used by [11] and [12].

We use subword parallelism to store and process multiple membership degrees in different sets in a single processor word. Using 8 b for encoding membership degrees \( \mu_{F_j}(u_i) \) and 4 b for set identification \( F_j \), a total of 12 b is required for a membership degree. Thus, in a 32-b processor, we can operate on two overlapping sets concurrently. This concept can be extended to five sets on a 64-b processor. This is usually sufficient for fuzzy control applications.

In our design, crisp input values are fuzzified with a single memory access, retrieving a data word encoding the fuzzy set and the associated membership degree (see Fig. 1). For each memory access, the membership function of a crisp input is loaded for two fuzzy sets. The membership degree of all other fuzzy sets defaults to zero.

This memory organization minimizes the number of memory accesses, requiring only one load operation per input independently of the number of fuzzy sets, thus speeding up the execution. In several lookup-table-based implementations [13], membership functions are stored in memory, typically requiring one load of membership degree for each input and fuzzy set effectively requiring \( n \times m \) loads and \( n \times m \) registers for a rule base.
with \( n \) inputs and \( m \) fuzzy sets. Our approach, in contrast, requires only \( n \) loads and \( n \) registers, independently of the number of fuzzy sets.

With the growing complexity of fuzzy problems (more inputs, more fuzzy sets per input) the register pressure grows, as more values are to be kept in registers than there are available registers. The contents of registers have to be spilled temporarily (e.g., stored on the stack), slowing down execution by requiring more slow memory accesses. In contrast, by using subword encoding, register pressure is reduced because fewer registers are required and memory traffic is reduced because less loads per inference are required. Because of the reduced number of accesses to the lookup table, cache thrashing is reduced, resulting in less cache reloads and spills and offering overall better memory performance.

Thus, using subword parallelism for determination of multiple membership functions not only reduces the actual cycle count in the pipeline, but also reduces memory demand and penalty. In addition, memory requirements for this organization are minimal and independent of the number of fuzzy sets. The memory consumption for each input value depends on resolution of the input and is only 1 kB if mapping 256 crisp input values to a membership function. Memory usage can be reduced by lowering the resolution of the fuzzification step.

III. EVALUATION OF THE FUZZY PROCESSOR

To improve performance of RISC processors for fuzzy calculation, we have explored different ISA extensions for fuzzy workloads. For this purpose, we have extended the original MIPS instruction set architecture with several instructions specialized for fuzzy computation.

The evaluation of the instruction set architecture is a major issue in the design of application-specific instruction processors [14]. Due to the many factors involved in performance optimization, suggested optimization solutions often minimize only the number of instructions necessary to solve a problem or, at best, the number of cycles.

To evaluate our extended instruction set architecture optimized for fuzzy calculation, we have employed hardware–software coevaluation [3]. This approach combines software evaluation methods with hardware synthesis of the application-specific processor to collect data for the evaluation process. Software evaluation establishes the cycle count and logic synthesis estimates cycle time and chip area. We have performed software evaluation using a cycle-accurate instruction set simulator and for hardware evaluation, we have performed logic synthesis to an application specific integrated circuit (ASIC) technology.

A. Software Evaluation

To determine the performance of the processor, we have used a cycle-level simulation of the instruction set architecture for the selected fuzzy workload to derive cycle count [15]. The cycle-level simulator we have used is based on the SPIM simulator [16] of the MIPS processor.

Using this simulator, we have determined the cycle count required for executing both MIPS and MIPS-F code for various fuzzy problems. Fig. 2 shows a snapshot of the simulator.

B. Hardware Evaluation

Hardware evaluation is performed using rapid prototyping based on logic synthesis. To evaluate hardware effects of instruction set extensions, we have designed an extendible RISC processor core in the VHDL [17] hardware description language (HDL). A detailed description of the reconfigurable processor core implementation can be found in [18].
Processor extensions for the fuzzy application are generated by adapting the processor core HDL description. The new functionality is added to the HDL modules affected and the extended processor core is synthesized using logic synthesis to a target ASIC process. This enables the evaluation of the resource usage and cycle time of the fuzzy processor. Early prototyping can be performed using FPGAs [19].

### C. Evaluated Instructions

We have evaluated the impact of several instructions (listed in Table I on program execution time). The third column of the table shows the result of the evaluation process, i.e., the instructions selected for implementation.

In addition to those instructions selected, we have evaluated subword parallel arithmetic instructions, as well as several multiply-and-accumulate variants. The performance gains from using subword parallel arithmetic was not sufficient for justifying their inclusion. While multiply-and-accumulate yielded significant improvements, it also led to inordinate complication of the hardware design and had negative impact on the cycle time.

We also explored possible synergistic effects between different instruction extensions by evaluating configurations consisting of different subsets of instructions. The selected instructions corresponded to a sweet spot in the design space. A detailed description of the evaluation process of several selected instruction set architectures optimized for fuzzy processing can be found in [20]. We give a detailed description of the selected instructions in Section IV.

We call the selected instruction set architecture MIPS-F for MIPS with fuzzy processing support. The implementation of our extended model does not increase the cycle time of the chip and causes only minor increase in the chip area. The optimized instruction set for fuzzy-specific workloads reduces the number of instructions which have to be executed reducing the execution time. As the cycle time of the processor is not degraded through complex hardware implementation, the gain in the performance is significant, as we show in Section VI.

### IV. MIPS-F Instructions

We have extended the original MIPS instruction set with the following application-specific instructions:

- **slw** loads fuzzified values from the memory;
- **rulev** evaluates fuzzy rules from the rule base;
- **min** minimum function for rule evaluation;
- **max** maximum function for rule evaluation.

In some processor implementations, the commands **min** and **max** may be replaced by conditional move. Note that although our work is based on the MIPS ISA, a similar approach can be used in conjunction with other ISAs.

#### A. Scaled Load Instruction

To optimize access to arrays, a register plus shifted register memory addressing mode (scaled load) can be used. This instruction is not specific to fuzzy calculation as it improves performance of array accesses. It can be found in general purpose programs and is included in a number of microprocessors, such as most CISC and several RISC processors.

The instruction **slw** takes two operands to specify a memory address—the operand stored in the register 

\[
R_k, R_x
\]

takes two operands to specify a memory address—the operand stored in the register \(R_x\) specifies the base address of the array and the operand in the \(R_k\) the index of the array. To generate the actual memory address, the index is multiplied by the data size (4 bytes) and added to the base address.

In fuzzy processing, we use the scaled load instruction to implement the table lookup for fuzzification in a single processor cycle.

#### B. Rule Evaluation Instruction

For improving inference performance, we have included a rule evaluation instruction **rulev**, which evaluates a fuzzy rule by determining the minimum of the rule premises in a single cycle.

Suppose we have a single rule with two premises

\[
R_1 : \text{if } x \text{ is } A_1 \quad \text{and} \quad y \text{ is } B_1 \text{ then } z \text{ is } C_1
\]

where \(A_1, B_1,\) and \(C_1\) are fuzzy sets. We implement Mamdani’s minimum operation inference using the following MIPS-F instruction

\[
\text{rulev } R_{\delta}, R_x, R_y, A_1, B_1.
\]

For the current input \(x_0\), the source register \(R_x\) contains membership degrees \(\mu_{A_1}\) and \(\mu_{A_2}\) for fuzzy sets \(A_1\) and \(A_2\), respectively, together with set identifiers as shown in Fig. 3. Similarly, in the register \(R_y\) are membership degrees \(\mu_{B_k}\) and \(\mu_{B_l}\) for sets \(B_k\) and \(B_l\) of the current input \(y_k\).

Activation values of the premises \(p_x\) and \(p_y\) are calculated by

\[
p_x = \begin{cases} 
\mu_{A_1}, & \text{if } A_1 = A_1 \\
\mu_{A_2}, & \text{if } A_1 = A_2 \\
0, & \text{otherwise}
\end{cases}
\]

\[
p_y = \begin{cases} 
\mu_{B_k}, & \text{if } B_1 = B_k \\
\mu_{B_l}, & \text{if } B_1 = B_l \\
0, & \text{otherwise}
\end{cases}
\]
Fig. 4. The four premises in registers $Sx$ to $Sz$ are evaluated using two rulev instructions. From the intermediate results, the minimum instruction yields the final result in the register $Sx$.

The firing strength of the rule is determined by

$$R_d = \min(p_x, p_y).$$

If the values contained in the source registers are premises of the rule under evaluation, the minimum value is determined and stored in the destination register $R_d$.

C. Minimum Instruction

The minimum instruction is used to evaluate rules with more than two premises. The instruction $\min R_d, R_s, R_t$ finds the minimum of source registers $R_d$, $R_s$ and $R_t$ and stores the result in the destination register $R_d$. This instruction allows to combine multiple rulev statements with two premises each to evaluate more complex rules. We list an example for evaluating rules with four premises in Fig. 4.

D. Maximum Instruction

We have included the maximum instruction $\max$ to perform result collection if several rules have the same conclusion. This corresponds to the evaluation of fuzzy rules of the form

if $(x$ is $A_1$ and $y$ is $B_1)$ or $(x$ is $A_2$ and $y$ is $B_2)$ then $z$ is $C_1$.

The instruction $\max R_d, R_s, R_t$ finds the maximum of source registers $R_d$, $R_s$ and $R_t$ and stores it in the destination register $R_d$. In Fig. 5, we show how to use this instruction to combine two rules with the same conclusion.

V. SAMPLE FUZZY APPLICATIONS AND PERFORMANCE

We have evaluated the performance of the processor for several fuzzy workloads. This performance analysis is based on case studies which we have selected from previously published fuzzy applications. This also illustrates the implementation of fuzzy processing using the enhanced MIPS-F instruction set. First, we implement a simple fuzzy controller to explain the usage of particular instructions. After that, we show how to efficiently implement more complex fuzzy applications.

In this case study, we have used the lookup table approach for fuzzification, for fuzzy reasoning we have employed the Mamdani’s minimum operation inference and for the conclusion part we have implemented Yager’s defuzzification method, which is an approximation for calculating the center of gravity [21]. Alternatively, other defuzzification approaches can be used on the intermediate results generated by the rule evaluation phase [22].

A. Fuzzy Controller for Gas Heating

A simple fuzzy controller for gas heating has two inputs delivered by sensors: the room temperature and the current gas heating value, and one output for controlling the valve opening angle for the gas supply. The membership functions of inputs have three fuzzy sets each, and of the output five fuzzy sets, as shown in Fig. 6(a). The sample controller uses nine rules as displayed in Table II.

In Fig. 7, we list the MIPS-F code for this controller. The first two instructions perform the fuzzification of the inputs in registers $Sx$ and $Sy$. The base addresses of the two lookup tables for fuzzification of the temperature and gas quality are in registers $S$ and $T$. Membership degrees for the current input values are loaded in registers $S$ and $T$.

The rules from the rule base are evaluated in program lines labeled 3 through 15. The code line labeled 3 evaluates rule $R_1$, checking whether registers $S$ and $T$ contain the membership functions of the fuzzy sets COOL and POOR for the corresponding inputs. If so, the minimal membership degree is determined and stored in the register $S$, otherwise the result is zero.

The instructions in lines 4 and 5 evaluate rules $R_2$ and $R_3$ storing the results in registers $S$ and $T$, respectively. As these two rules conclude in the same fuzzy set, their maximum value

TABLE II

<table>
<thead>
<tr>
<th>Rule</th>
<th>Temperature</th>
<th>Gas quality</th>
<th>Valve angle</th>
</tr>
</thead>
<tbody>
<tr>
<td>R11</td>
<td>COOL</td>
<td>POOR</td>
<td>HUGE</td>
</tr>
<tr>
<td>R21</td>
<td>COOL</td>
<td>MEDIUM</td>
<td>LARGE</td>
</tr>
<tr>
<td>R22</td>
<td>TEPID</td>
<td>POOR</td>
<td>LARGE</td>
</tr>
<tr>
<td>R31</td>
<td>COOL</td>
<td>HIGH</td>
<td>HALF</td>
</tr>
<tr>
<td>R32</td>
<td>TEPID</td>
<td>MEDIUM</td>
<td>HALF</td>
</tr>
<tr>
<td>R41</td>
<td>TEPID</td>
<td>HIGH</td>
<td>SMALL</td>
</tr>
<tr>
<td>R42</td>
<td>WARM</td>
<td>POOR</td>
<td>SMALL</td>
</tr>
<tr>
<td>R51</td>
<td>WARM</td>
<td>HIGH</td>
<td>TINY</td>
</tr>
<tr>
<td>R52</td>
<td>WARM</td>
<td>MEDIUM</td>
<td>TINY</td>
</tr>
</tbody>
</table>
is determined using the \texttt{max} instruction in the code line 6 and the result is stored in the register $13$. Other rules from the rule base are evaluated similarly, accumulating intermediate results in registers $10$ to $14$.

The rest of the program performs the defuzzification. In code lines 16 to 39, the singletons of the output fuzzy sets are loaded into registers $15$ through $19$. If an intermediate result is greater than zero, a multiplication by the singleton of the corresponding output fuzzy set is performed, as shown in code lines 16 to 39. The dividend and divisor are accumulated in registers $20$ and $21$, respectively. The output control value is determined by a final division, yielding the result in the register $22$.

### B. Fuzzy Policer for Traffic Control in ATM Networks

The next case study evaluates the performance of a more complex fuzzy workload having three inputs and eighteen rules. We have encoded a fuzzy controller for policing real-time traffic in asynchronous transfer mode (ATM) networks, as described in [12].

In ATM networks, two controlling mechanisms are required: admission control and source policing. The admission controller accepts a new connection and allocates traffic parameters to it. The source policer assures that each source conforms to its traffic parameters detecting nonconforming sources and minimizing their negative effects. In the fuzzy policer, fuzzy logic is employed for making a decision whether or not a source respects assigned traffic parameters.

The fuzzy policer has three inputs: the average number of packages received per time window since the start of connection $A_w$, the number of packages received in the current time window $A_t$, and the maximum number of packages $N_i$ which can be accepted in that window. The output of the fuzzy policer is an adjustment $\Delta N_{i+1}$ to the threshold $N_{i+1}$ in the next window.

Membership functions of the inputs have three overlapping sets (Low, Medium, and High), whereas the output variable $\Delta N_{i+1}$ is specified with seven output fuzzy sets (ranging from NB for negative big to PB for positive big), as shown in Fig. 6(b). The rule base of the fuzzy policer is given in Table III.

### C. Fuzzy Classifier for Encoding Video Sequences

Our last case study evaluates the performance of a fuzzy classifier for encoding video sequences, as presented in [23].

In encoding video signals, each video frame is divided in nonoverlapping blocks of $16 \times 16$ pixels called macroblocks. To optimize the quality of subjective perception of each encoded video frame according to the human visual system, a fuzzy controller determines a relevance factor for each macroblock.

The relevance factor of a macroblock is determined depending on the energy ($E_k$) of the macroblock $k$ on its entropy window $A_w$, and the maximum number of packages $N_i$ which can be accepted in that window. The output of the fuzzy policer is an adjustment $\Delta N_{i+1}$ to the threshold $N_{i+1}$ in the next window.

Membership functions of the inputs have three overlapping sets (Low, Medium, and High), whereas the output variable $\Delta N_{i+1}$ is specified with seven output fuzzy sets (ranging from NB for negative big to PB for positive big), as shown in Fig. 6(b). The rule base of the fuzzy policer is given in Table III.
(H_k) and on its current motion activity (MV_k). The result of this classification is a relevance factor R_k. Lower relevance value means that the distortion in that area is less perceptible by the human visual system.

Membership functions of the inputs E_k and H_k to the fuzzy classifier are defined with four overlapping sets (as shown in Fig. 8) ranging from VL for very low to VH for very high. Membership function of the input MV_k has three fuzzy sets and the output is determined from five output fuzzy sets. The rule base of the classifier is given in Table IV.

### VI. Processor Performance

To evaluate the performance of the processor, we have collected statistics for executing the three described fuzzy workloads and compared them to the data collected if executing the same workloads on the original MIPS processor architecture. The most important aspect of processor execution is the execution time. We discuss the cycle count required on both processors but also the size of the machine code (this aspect is of particular importance for embedded applications), and hardware aspects of the processor design such as chip area and cycle time.

The cycle count for executing the three applications is given in Table V. We list the cycle count required for each of the three processing steps in fuzzy calculation for executing on the MIPS processor and on the MIPS-F processor.

Fig. 9 shows that the execution profile of fuzzy problems changes considerably with problem complexity. With growing program complexity, the impact of fuzzification and defuzzification on the overall execution time decreases whereas the computing time for rule base evaluation becomes more significant.
Thus, while simple problems spend much of their execution time in defuzzification, inference takes up as much as 70% of execution time in complex problems. With growing complexity, more time will be spent in rule evaluation, reducing the relative importance of data conversions in the fuzzification and defuzzification phases. This has influenced our decision to concentrate our efforts on accelerating rule evaluation.

Extending the MIPS instruction set with the application-specific MIPS-F instructions results in significant reduction in program execution time. Fig. 10 gives an overview of the cycle count required for executing fuzzy workloads of varying complexity. The newly introduced instructions reduce cycle count for fuzzification and inference, showing a speedup of 237% to 350% for fuzzification and of 123% to 254% for inference. The execution of the whole program is improved 37% to 85% depending on application complexity with complex application profiting the most.

As a result, the execution profile of fuzzy applications executed on the MIPS-F processor changes considerably. As shown in Fig. 11, the most time consuming task on the MIPS-F processor is defuzzification. This process requires from 94 to 100 cycles for the selected applications, depending on problem complexity. As we did not introduce any instruction in the MIPS-F processor to speed up defuzzification, the cycle count required for this step stays unchanged, but grows in percentage of the total execution time.

The command `rulev` not only speeds up inference, but also fuzzification because fewer data have to be fetched from memory. As an example, the introduction of the `rulev` command reduces the cycle count required for fuzzification from 18 to 4 cycles for the ATM network fuzzy policer application.

Another important metric is code size, especially for embedded applications, where memory resources are limited. Using the MIPS-F instruction set for fuzzy calculation we have fewer instructions and we reduce the code size significantly, as shown in Fig. 12 (the code size is reduced by 104% to 173% for the selected applications).

In addition to code size, the memory resources required for storing membership functions of the inputs are also reduced significantly, because of the subword data memory organization. Thus, the memory requirements for membership functions decreased by 200% to 266% for the selected workload.

The application-specific fuzzy processor implementing the MIPS-F instructions operates at a cycle time of 15 ns as the original MIPS core. The fuzzy-calculation specific instructions consume 1.3% more chip area than the basic chip, resulting in a chip which occupies 24 525 sq mil on the AMS 0.6μ process [24], implemented using logic synthesis based on a MIPS processor core.

VII. RELATED WORK

Fuzzy computation can be implemented on any general purpose processor, but for obtaining higher execution speeds fuzzy specific hardware is needed. Many fuzzy-optimized hardware implementations have been proposed in the literature. Generally, these implementations can be grouped into custom fuzzy implementations and fuzzy programmable solutions.

Custom fuzzy implementations hard wire a particular fuzzy application in hardware. Hardware design is highly optimized for a target fuzzy application and is mapped to an ASIC or FPGA process using logic synthesis. Such implementations offer high processing speed but are dedicated to a single fuzzy application. In addition, custom fuzzy solutions are suitable only for fuzzy applications of limited complexity. Several custom fuzzy implementations are presented for various fuzzy applications such as for an ATM network controller [12] or libraries of components are provided for synthesizing a particular application [22], [25].

To allow both flexible execution and to increase the speed of fuzzy computation (compared to general purpose processor), a fuzzy-optimized hardware implementation has to be programmable. This can be realized by either designing a specialized coprocessor for fuzzy tasks or extending an existing general purpose processor with some fuzzy add ons.
Fuzzy coprocessors work in conjunction with a host processor. They are not dedicated to a single fuzzy application offering higher flexibility to be used in system design of varying complexity. However, fuzzy processing performance is usually lower compared to a custom fuzzy implementation. Several fuzzy coprocessors are available as commercial components, such as the Omron FP-3000 [26], Togai Infra Logic FC110 [27], Siemens SAE 81C99 [28], NeuraLogix NLX230 [29], and SGS-Thompson WARP [30], [31]. A detailed discussion of issues arising in the implementation of such coprocessors can be found in Patyra et al. [32].

Another possibility to implement a flexible fuzzy solution with high computation speed is to extend an existing processor with several fuzzy-specific instructions adding some functional blocks to support fuzzy computation. Such solutions offer even more flexibility yet preserving their general-purpose computation capabilities. In addition, these solutions support higher speed compared to general-purpose processors, but reach lower performance than custom fuzzy implementations.

Watanabe [5], [33] analyzes the performance impact of the availability of various instructions on fuzzy processing performance. He finds that min and max operations are used heavily in inference and significant speedups can be obtained by hardware support for these primitives. He also considers the use of vector processing to process membership vectors.

Patyra and Braun [6] describe the implementation of a proprietary, RISC-style central processing unit (CPU) called F/S RISC in “M” and its simulation using “Lsim.” This implementation uses long internal registers to represent membership vectors, an approach similar to the vector registers suggested by Watanabe [33].

Costa et al. [13] describe the implementation of FLORA, a general-purpose RISC processor with fuzzy support. The FLORA instruction set supports minimum and multiply-and-accumulate instructions to aid in fuzzy processing. The processor operates at 40 MHz on 32-b-wide data.

An example of a commercial processor incorporating extensions to support fuzzy processing is the M68HC12 microcontroller from Motorola [34]. This processor includes several fuzzy-specific instructions to support fuzzy calculation. Fuzzification is supported by the instruction mem for calculating membership functions, inference by the instructions rev and revw for fuzzy rule evaluation and evaluation of weighted fuzzy rules, respectively. Defuzzification is improved by the instruction wav for calculating weighted sum or emacs for multiply-and-accumulate operation. The processor operates at 8 MHz on 16-b-wide data.

VIII. CONCLUSION

In this paper, we have introduced application-specific extensions for fuzzy processing to the MIPS architecture. Application-specific performance is achieved by extending a general-purpose high-performance RISC processor core with instructions critical to the efficient execution of the fuzzy processing problem domain.

The selection of instructions which improve fuzzy calculation most efficiently is performed using hardware/software code evaluation techniques. The evaluation process has shown that the scaled load instruction slw for fuzzification, and rule evaluation rulev, minimum min and maximum max instructions for inference speedup should be added to a RISC processor core based on the MIPS instruction set architecture. Of these instructions, only the rulev instruction supporting hardware-assisted rule evaluation is specific to fuzzy computation.

We have shown that by extending the MIPS instruction set architecture with these few instructions, fuzzy calculations show a significant speedup. To evaluate the performance of the processor, we have collected statistics in executing three representative fuzzy applications. Comparing these statistics to those of the original MIPS processor, we have shown that the execution time on the MIPS-F processor has a speedup by 37% to 85%, the machine code size is reduced by 104% to 173%, and the memory requirements are decreased by 200% to 266%, depending on problem complexity. The fuzzy extensions to the MIPS RISC processor cause no increase of the processor cycle time and only a minor increase in chip area.

ACKNOWLEDGMENT

The author would like to thank M. Gschwind for his valuable help in the evaluation of the MIPS-F instruction set and in the preparation of this manuscript. She would also like to thank the reviewers whose suggestions have contributed to the quality of this work.

REFERENCES


