Hardware-Software Codesign of a Fingerprint Identification Algorithm

Nicolau Canyellas¹, Enrique Cantó¹, Giuseppe Forte¹, and Mariano López²

¹ Escola Tècnica Superior d’Enginyeria ETSE-URV, Tarragona, Spain
ncanyell@etse.urv.es
http://www.etse.urv.es

² Escola Universitària Politècnica de Vilanova i la Geltrú EUPVG-UPC, Spain
http://www.upc.es

Abstract. Automatic Fingerprint Authentication Systems are rapidly being incorporated in a wide range of applications, satisfying the society demand of accurate identification frameworks in order to prevent unauthorised accesses or fraudulent uses. Most of biometrics based personal identification systems run on high-performance computer based platforms, which execute a set of complex algorithms implemented in software. Those solutions cannot be applied to small, low-cost and low-power embedded systems, based on microprocessors without floating-point arithmetic unit. In this article we present a hardware/software implementation of a fingerprint minutiae extraction algorithm. The proposed system consists of a microprocessor and a coprocessor implemented in an associated FPGA. In order to develop an efficient implementation, fixed-point computations have substituted the floating-point ones. Due to the low feature requirements the whole system is suitable for a SoC with embedded flexible hardware.

1 Introduction

Fingerprints, which have been used for about 100 years in personal identification [6][10], are the oldest biometrics signs of identity, and the most widely used today. Fingerprint biometrics offers a set of key advantages over other biometrics: small and low-cost capacitive or thermal sensors, very low variation of a fingerprint along time, suitability in medium-high security authentication applications, and commodity to users. Any Automatic Fingerprint Authentication System (AFAS) is structured in three different stages: image acquisition, feature extraction and matching, this work covers the minutiae extraction stage, the one with highest computational requirements.

The most common representation used in fingerprint identification is the Galton features or minutiae [10]. Minutiae are the set of points where the ridges of the fingerprint end or split, and minutiae-matching is the usual way to perform automatic fingerprint comparison. It offers a good saliency property (features extracted from fingerprint images contain distinctive information), although not so good suitability (easy feature extraction, low storing requirements and useful matching) due to the high computational capacity required during minutiae extraction.

Several approaches to automatic minutiae extraction have been proposed [13][17]. Most of them use orientation field computation, directional filters, binarization and thinning in order to obtain a binary representation of the fingerprint image, where the
thickness of the ridge lines is reduced to one pixel, from where minutiae is easily extracted. In our work we use the Maio and Maltoni ridge line following algorithm [14], because it permits minutiae extraction directly from the gray-scale fingerprint image. This method is computationally less expensive than others, and it can be re-written to be implemented without floating point operations. These facts are very important, bearing in mind that the goal is to implement a high-speed and low-cost embedded system. In order to implement a real-time AFAS we propose a hardware/software design; it consists of a microprocessor with an associated specialised coprocessor, able to perform the most time-consuming functions of the ridge line following algorithm.

The use of fingerprint biometrics coprocessors is still a young field. A great majority of commercial fingerprint OEM modules [1][3][7][9][19] are based on embedded high performance 32-bit processors or DSPs (Digital Signal Processors), but feature extraction times are about 1 second or more. The IKENDI manufacturer offers the IKE-1 ASIC [12] based on an ARM7TDMI processor and a ConvTree-III coprocessor comprising less than 50K gates and it claims to encode 10 to 20 times faster than other DSP solutions at the same clock speed. The UCLA University group has developed the ThumbPod prototype [22], where the fingerprint recognition is based on a 32-bit LEONII microprocessor and a DFT (Discrete Fourier Transform) coprocessor mapped on a high-end Xilinx Virtex II FPGA. The DFT coprocessor is used to determine the dominant ridge flow direction in every block. Experimental results show that the coprocessor permits a 55% execution time reduction for the minutiae extraction, but 4 seconds of execution time is still quite high for many applications.

All of these solutions implement the AFAS system executing a set of software-implemented complex algorithms, and require the use of high feature processors or DSPs in order to perform a real time authentication. In our approach we solve the authentication problem using a low complexity algorithm, and designing a specific coprocessor for it.

A brief description of the ridge line following algorithm is presented in section 2, including our modifications in order to reduce the computational requirements needed to obtain an efficient hardware implementation. In section 3 the results of a software implementation are presented, showing the percentage of overall execution time spent in each one of the algorithm main functions. The critical tasks of the algorithm, located with this information, are mapped on the coprocessor, and section 4 describes its hardware. The obtained experimental results are presented in section 5. Finally section 6 presents the conclusions of our work.

2 The Ridge Line Following Algorithm

For minutia extraction we have analysed the ridge line following algorithm with the parameter values adopted in [14], and then we have modified some processes of the proposed methodology in order to minimise the computational requirements. These modifications are carried out bearing in mind the hardware implementation of critical tasks. We developed Matlab and C versions of the original algorithm, and then we started to simplify the used data types, changing floating point operations with integer versions, and trying to eliminate the computationally expensive operations like products and divisions.
From a mathematical point of view, a ridge line is defined as a set of points which are local maxima along one direction. The algorithm is based on the idea of tracking the fingerprint ridge lines on the gray scale image by “sailing” over these points, according to the fingerprint directional image. At each iteration of Maio’s algorithm [14], during the ridge following, a new section $\Omega$ is determined on the orthogonal direction to the ridge ($\Phi_c = \varphi + \pi/2$ where $\varphi$ denotes the ridge direction. Then the process locates a local maximum over a regularised silhouette of $\Omega$ in order to determine the next iteration point. As depicted in Fig. 1, given a starting point $(i_c,j_c)$ and a starting direction $\varphi_c$, the algorithm computes a new point $(i_t,j_t)$ by moving $\mu$ pixels from the starting point along direction $\varphi_c$. At this point the section $\Omega$ is computed as the set of $2\sigma+1$ points orthogonal to $\varphi_c$ and having $(i_t,j_t)$ as median point. A third step involves the computation of an averaged section of $\Omega$, denoted as $\text{avg}(\Omega)$, as the local average of each of the pixels belonging to section $\Omega$ with the two pixels belonging to the two parallel planes at a distance of $\pm 1$ pixel on the $\varphi_c$ direction, denoted as $\Omega^{-1}$ and $\Omega^+$. The next step computes a correlation $\text{corr}(\Omega)$ of the averaged section $\text{avg}(\Omega)$ against the gaussian silhouette mask showed in Fig. 3(a), and returns its weak local maximum $(i_n,j_n)$. An auxiliary binary image $T$ of the tracked points and its neighbours is updated in order to prevent a second tracking of the same ridge. Finally $(i_n,j_n)$ and its dominant ridge flow direction $\varphi_n$, become the new starting point and direction to start a new iteration. These steps are repeated during a ridge line following until a stop criterion is reached.

![Fig. 1. Ridge line following of a fingerprint ridge, from point $(i_c,j_c)$ to point $(i_n,j_n)$](image-url)

The most important change in the algorithm relates to the ridge direction computation, $\varphi_c$. This direction represents the ridge line local direction and can be computed as the tangent to the ridge at point $(i_c,j_c)$. The method used by Maio was proposed by Donahue and Rokhlin [8], and is computationally more expensive than the presented in [21] and used by Halici and Ongun in [11]. We have modified the proposed mask in order to obtain 24 different directions with angles varying in steps of 15°, which corresponds with $\mu = 3$ as used by Maio (see Fig 2).

An inspection of the extraction algorithm shows another computational expensive step. In the process of finding the local maximum in a section $\Omega$, the algorithm first calculates the local average of the gray levels of the pixels that belong to $\Omega$, $\Omega^+$ and $\Omega^-$, to finally compute a correlation with a gaussian silhouette mask.
The local average with two neighbors pixels involves a division by three, and the
correlation includes a division by 23; in our algorithm none of these divisions are
performed. These changes are equivalent to a change of scale, without loss of accu-
rance, and do not affect the position of the local maximum.

Moreover, a correlation involves products that can be substituted by bit-shift op-
erations, which can be implemented in hardware in a very efficient way, if performed
with values that are powers of 2. Fig. 3 presents the original gaussian mask (a) and
our proposed one (b), with changed weights in order to simplify the arithmetic opera-
tions involved in the correlation.

Our modified version of the algorithm has been validated using the same samples
used in [14] and available at [2]. It is composed of seven fingerprints taken from the
NIST fingerprint database [20], four fingerprints from an FBI sample set and three
fingerprints acquired through an opto-electronic device based on a prism. Table 1
reports the results in terms of undetected minutiae (dropped), non-existent minutiae
(false) and type-exchanged minutiae (exchanged).
There are minor differences in the comparison between the results reported in Maio’s original work with the ones obtained with our modified version of the algorithm. So, the algorithm modifications are validated, and it is demonstrated that is possible to construct a complete identification system based in our modified version of the ridge line following algorithm. In both cases most of errors of the ridge line following algorithm are minutiae exchanges, mainly due to some termination minutiae which are detected as bifurcation minutiae. If a termination minutia is very close to another ridge line the algorithm may skip the termination and intersect the adjacent ridge line. In [14] it is proposed to train a neural network to verify the type of minutiae detected, performing a local analysis of the gray scale image in each minutia neighbourhood. In our approach we are evaluating the method proposed by Methre [15]. We study the dominant direction by computing a histogram of directions in the neighbourhood of the minutia. If there is only one dominant direction the minutia is marked as a termination, but if there are two dominant directions it is marked as a bifurcation.

3 Hardware-Software Partitioning

The software implementation of the algorithm reflects the process of ‘sailing’ over a ridge as described in Section 2, and it consists of a set of loops executing the same iterative process, one time and another.

Code 1. Main functions of our integer version of the algorithm

```c
while (!end_condition) {
    next_point(angle_c,µ,&delta_i,&delta_j);
    i_t = i_c+delta_i;
    j_t = j_c+delta_j;
    createsecc(i_t,j_t,angle_c,seccX,seccY);
    filtersecc(seccX,seccY,angle_c,avg);
```
index = corr_weak_max(avg);
i_n = seccX[index];
j_n = seccY[index];
angle_n = tg_dir(j_n, i_n, angle_c);
...
updateT(j_c, i_c, j_n, i_n);
...

The C source lines listed above as Code 1, show the execution of the basic functions used in the inner loop of our version of the ridge-following algorithm. The rest of the minutiae extraction program checks for a stop criterion, and controls the correct processing of the whole fingerprint, but representing a few part of the total execution time.

We analysed the code, detecting the most critical tasks, and then we choose the hardware-software partitioning in order to accelerate the minutiae extraction process. The profiling of the program functions is detailed in Table 2. The function name is in the first column, the second one shows the total number of calls to it, and the last column represents the percentage of the overall time spent in the execution of the function.

We decided to implement a coprocessor that computes the first nine lines of Code 1, after the while sentence. So, the coprocessor will map the first five functions of the Table 2, which represent the 81.0 % of the overall ridge line following algorithm execution time. The updateT function represents 10.4 % of the total execution time but it is not included in the coprocessor because it is not executed in all the executions of the loop. Moreover, the high execution time of this function is due to the elevate number of memory write accesses it performs, but nor for its complexity neither for a great number of total calls.

**Table 2.** Percent of overall execution time spent in the main functions of the algorithm

<table>
<thead>
<tr>
<th>Function</th>
<th>#calls</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>next_point</td>
<td>29608</td>
<td>6.8</td>
</tr>
<tr>
<td>createsecc</td>
<td>3471</td>
<td>11.1</td>
</tr>
<tr>
<td>filtersecc</td>
<td>3471</td>
<td>16.5</td>
</tr>
<tr>
<td>corr_weak_max</td>
<td>3471</td>
<td>30.2</td>
</tr>
<tr>
<td>tg_dir</td>
<td>3526</td>
<td>16.4</td>
</tr>
<tr>
<td>updateT</td>
<td>1347</td>
<td>10.4</td>
</tr>
<tr>
<td>others…</td>
<td>-</td>
<td>8.6</td>
</tr>
</tbody>
</table>

On the other hand, the coprocessor includes the which, although it is not a complex function, collects the 6.8 % of the execution time due to its great number of executions. This great number of calls is because it is also called several times during the execution of the createsecc and filtersecc functions.

Table 3 shows experimental results of the program execution in two different prototyping platforms. The first column shows the execution time in a Xilinx MicroBlaze embedded soft core, a reduced instruction set computer (RISC) optimised for implementation in Xilinx FPGAs [16]. This soft core has been implemented in a Xilinx Virtex II XC2V1000-4FG456C device (part of a Celoxica RC200 prototyping plat-
form). The second column corresponds to an implementation in a Memec design Inc. Virtex-II Pro FF672 development board. It incorporates a Xilinx XC2VP4 device, with a PPC405, a 32-bit implementation of the PowerPC embedded-environment architecture [17].

<table>
<thead>
<tr>
<th>Execution time (s)</th>
<th>Mblaze (50Mhz)</th>
<th>PowerPC (100Mhz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>5.58</td>
<td>2.59</td>
</tr>
<tr>
<td>9 first lines of Code 1</td>
<td>5.01</td>
<td>2.37</td>
</tr>
</tbody>
</table>

These execution times validate the profiler results, and show that the first nine lines of code represent about 90% of the overall execution time. The implementation of these functions in hardware can improve significantly the identification process.

4 The Coprocessor

The coprocessor implements a hardware version of the critical functions, corresponding to the first 9 lines of the Code 1 after the while sentence. As depicted in Fig. 4, it has been divided in six stages plus a control unit (CU). The coprocessor works with a 256*256 pixels image of 8-bit gray levels stored on a SRAM, and with fixed parameters $\mu=3$ and $\sigma=7$ as used by the Maio-Maltoni algorithm [14]. The starting point $(i_c,j_c)$ and the discretised angle $\phi_c$ are read from the $x0,y0$ and $w0$ ports, then the $N_{rst}$ signal is asserted and the coprocessor negates $N_{stop}$ until the computation is completed, returning the weak local maximum point $(i_t,j_t)$ and the discretised angle $\phi_t$ of the dominant ridge flow direction in the ports $x1,y1$ and $w1$.

The first stage, named xynxt, computes the point $(i_t,j_t)$ from a starting point $(i_c,j_c)$ and angle $\phi_c$. It corresponds to the hardware implementation of the first three lines of Code 1, after the while sentence. The second stage, named secc, implements the fourth line of Code 1 to estimate the $2\sigma+1=15$ points of the section orthogonal to the starting angle $\phi_c$, read from port $w0$. The computed points of the section are stored in an internal RAM of the coprocessor, denoted as $\Omega$.

The next stage, named filt, implements the fifth line of the Code 1 and computes the average gray values of the pixels that belong to the section $\Omega$ with the pixels that belong to the two parallel planes at a distance of $\pm 1$ pixel on the $\phi_c$ direction, denoted as $\Omega^1, \Omega^{-1}$. It reads the internal coprocessor RAM that contains the pixels of the section $\Omega$, to determine the address of pixels at $\Omega, \Omega^1$ and $\Omega^{-1}$, in the external RAM, in order to compute the avg($\Omega$) of the $2\sigma+1=15$ points to store it into another internal coprocessor RAM.

The fourth and fifth stages, named corr and x1y1 respectively, implement the lines six to eight of the code. First the correlation of the avg($\Omega$) is computed as described in Section 2. And then, the fifth stage computes the pixel $(i_n,j_n)$ and writes it on the $x1,y1$ ports of the coprocessor.

Finally, the last stage, named tgdns, implements the ninth line of Code 1. Estimating the dominant ridge flow direction $\phi_n$ in the point $(x_n,y_n)$, as described on Section 2.
The number of clock cycles needed to complete all six stages is the sum of clock cycles of every stage plus a few due to the CU needs, resulting a total time of $720 \times T_{CLK}$. The details of the implementation, as well as the estimate of the number of clock cycles needed by the coprocessor to compute, can be found in some of our previous works [4][5].

The coprocessor has been described in VHDL, and synthesized and mapped on several low cost FPGAs and an ASIC technology.

In a XILINX SPARTAN-II 2S30 FPGA, of about 30Kgates equivalent, the synthesis reports 749 CLBs (configurable logic blocks) used, 87% of all available, and a maximum clock frequency of 76MHz. The synthesis circuit for an Altera ACEX EP1K30, also of about 30Kgates equivalent, occupies 1179 LCs (logic cells) (68%) and a maximum clock frequency of 83MHz. The synthesis results for the Atmel FPLSIC AT94K shows 1.200 LUTs (look up table) occupied and a maximum clock frequency of 23MHz. The synthesis result for an ASIC SCL05U technology of 0.5µm is 15.527 equivalent gates and 97MHz of maximum clock frequency.

In the used prototyping platforms, with a XILINX VIRTEXII XC2V1000fg456 device, the synthesis reports the use of 542 CLBs, and a maximum clock frequency of 111 MHz. The execution of the steps performed by the coprocessor with a 100 MHz system clock, is of 7.2 µs, from a starting point (x0,y0) and indexed angle w0, until the next point (x1,y1) and indexed angle w1 is completed. The execution of the software version of the code takes an average time of 784 µs to execute in the PowerPC version at 100 MHz and 1.69 ms if executed in a microBlaze processor running at 50MHz. Finally, the same code running in an ARM7TDMI processor, with a 100 MHz clock, takes an average time of 211 µs to execute. The coprocessor greatly improves the speed performance when compared with the 32-bit general-purpose microprocessor, due to the pipeline scheme of the coprocessor architecture and the parallel execution of computations into its stages. The time devoted to computations mapped on the coprocessor is reduced 92% when compared with the software running in the PowerPC processor at the same clock speed.

The overall execution time of the algorithm running in the PowerPC with the coprocessor is reduced from 2.59 s to 380 ms, that is a reduction of about 85 %. 
5 Conclusions and Future Work

The trend, in the prevention of unauthorised accesses or fraudulent uses, is the progressive replacement of current PIN or password-based identification methods by biometrics-based ones. Low-cost and low-response-time are the key parameters in an embedded fingerprint authentication system, as they will facilitate their integration in a wide range of new applications. Up to now, the biometrics algorithms have been developed using high level languages and complex arithmetic operations, and without thinking in the required computational resources. As a result, the biometrics identification must be performed in a high capacity platform, as the involved algorithms need a high capacity processor, not suitable to be embedded in low cost equipment.

Our work proofs that fingerprint biometrics algorithms can be implemented in low capacity microprocessors without performance loss. In a first stage, the algorithms must be optimised in order to reduce the number of floating point operations. It is demonstrated that the operations can be performed with integer numbers without significant changes in the obtained minutiae. In a second stage, a profiling of the software version locates the most critical tasks of the algorithm. These functions are implemented in hardware, with a significant improvement of the biometrics identification time (about 70 – 90 % of reduction).

In conclusion, we have demonstrated that, if the algorithms are optimised thinking in terms of hardware, an automatic fingerprint authentication system can be embedded in a low-performance-processor based system.

In the near future we will implement a complete embedded fingerprint-based-identification-system. The objective is the use of a 8 bit microprocessor with an associated dynamically reconfigurable FPGA. In this way, the associated hardware dimensions will be reduced, reconfiguring the FPGA to perform the concrete task needed at each moment.

References