A Dedicated Image Processor Exploiting Both Spatial and Instruction-Level Parallelism*

A.Broggi, M.Bertozzi, G.Conte

Dipartimento di Ingegneria dell'Informazione
Università di Parma
I-43100 Parma, Italy

F.Gregoretti, R.Passerone, C.Sansoè, L.M.Reyneri

Dipartimento di Elettronica
Politecnico di Torino
I-10129 Torino, Italy

Abstract

This paper presents the PAPRICA-3 massively parallel SIMD system, designed as a hardware accelerator for real-time image processing tasks. It is composed of a linear array of single-bit processing elements, including a fairly complex pipelined controller, thus allowing the system to take advantage also of the intrinsic parallelism in a program.

A programming environment has been developed to ease the prototyping of applications: a code generator converts C++ programs into assembly, and code optimization is performed directly at the assembly level, following a genetic approach.

The effectiveness of the processor, as well as of the code optimizer, are discussed with the aid of an application example for handwritten character recognition.

1 Introduction

This paper describes PAPRICA-3, a real-time hardware accelerator for low-level image processing applications. The system is designed to be interfaced to a conventional microprocessor to relieve it from the high computational load required by the first steps of image processing algorithms (filtering, noise removal, contrast enhancement, features extraction, etc.). The system is able to capture a serial input data stream coming from an external camera (or pair of stereo cameras), to process it in parallel on a line-by-line basis, and to store the resulting images either on a dedicated dual-port memory, accessible also from the conventional processor, or on a monitor.

One of the design goals was to build a low-cost, high-performance system, dedicated to embedded applications. PAPRICA-3 was initially formulated as the core of a high speed system for bank cheques processing called HACRE (HAndwritten Character REcognition). The main task of the system was to perform automatic recognition of the amount handwritten on the legal and courtesy fields of a bank cheque, at a rate of about ten cheques per second. A solution to the problem is to design a dedicated parallel processor with an instruction set tailored to low-level image processing tasks. To meet the cost and size constraints typical of embedded systems, the hardware accelerator has to be composed of the smallest possible amount of components, with a single-chip unit as the most desirable solution.

The heart of PAPRICA-3 consists of a linear array of processing elements (PEs), each one devoted to a single line pixel. At any time, all PEs execute the same instruction. The system operates on black & white, grey-scale or color bitmap images, so that a pixel is defined as a multi-bit value carrying information stored on different image bit-planes. A complete image bit-plane can be seen in different ways, for example as an independent black & white image, as part of one or more grey-scale or color images, or as an intermediate result of operations performed on any of the above objects.

Each PE can perform *logical* or *morphological* operations on an input pixel: logical operations compute a one-bit result depending on values stored on different image planes of the same pixel, while morphological operations evaluate a one-bit result based on the values stored in the neighboring pixels on a single image plane, using the *mathematical morphology* approach proposed in [16].

The execution unit, which is common to the whole processor array, controls program flow and can perform tests, branches, conditional execution of program segments, loops, and synchronization statements. To speed up program execution, a fairly complicated pipeline scheme has been adopted to exploit as much

*This work was partially supported by the CNR and the Italian Ministry for University and Scientific Research 40%.

instruction level parallelism as possible. The program is stored completely in an internal program memory (512 instructions wide) writable from the external processor, so there is no need for internal cache memory.

A Status Register File can be used to compute and store *global parameters* of the processed images, such as statistics, degrees of matching between an input and reference images, etc.. It is accessible to the external processor and is aimed to increase execution speed of a broad class of neural algorithms.

Two versions of the PAPRICA-3 system are under development, having the same architecture: the first one is dedicated to the above mentioned handwriting recognition application and in the following will be also referenced with the name HACRE, while the second one is of a general purpose aimed to build an image processing system based on a single PCI bus PC expansion board and will be called generically PAPRICA-3. Two different ASIC chips will therefore be built:

- HACRE is a single-chip system. The chip is composed of 64 PEs, an input video interface dedicated to a linear image scanner, a 1k×64 bits internal image memory and an internal 256 elements Status Register File. It interfaces to a 32 bit microcontroller with a minimum amount of external logic.

- PAPRICA-3 is a modular system. Each chip is composed of 32 PEs, a generic input/output video interface and does not include any internal image memory and the Status Register File. Instead, the chip has a fast interface to an external image memory and a Status Register File, to achieve two goals: connecting several chips together, it is possible to create a system with a varying number of PEs; an external but fast image memory will not decrease too much the performances compared with an internal memory, but the amount of memory can be much higher, to cope with memory demanding applications.

A programming environment has been developed to ease prototyping of applications on PAPRICA-3. Application programs are written in C++; a code generator first converts C++ statements into assembly code; subsequently, optimization is performed on the generated assembly code. The optimizer performs both deterministic optimizations following the standard well known techniques and stochastic techniques to improve performance of the pipeline structure and take advantage of the intrinsic instructions parallelism.

This paper presents the whole project which started about 3 years ago and involved two reserach units: Parma University and Torino Politechnic. Next section describes the hardware system and architecture; section 3 presents the programming environment and details the code optimization process, describing also its implementation on a parallel cluster of workstations; section 4 describes the main application that motivated the development of PAPRICA-3; and finally section 5 concludes the paper with some remarks.

2 Hardware System Description

As shown in figure 1, the kernel of PAPRICA-3 is a linear array of Q identical 1-bit PEs connected to an image memory via a bidirectional Q-bit wide bus. The image memory is organized in addressable words whose length matches that of the processor array; each word contains data relative to one binary pixel plane (also called *layer*) of one line of an image (Q bits wide), and a single cycle is needed to load an entire line of data into the PE's internal registers. When executing a program, the correspondence between the line number and the pixel plane of a given image and the absolute line address is computed in hardware by means of data structures, named *Image Descriptors*, stored in the Control Unit. An Image Descriptor is a memory pointer built up of three parts: a base address, a line counter and a page counter. Line and page counters can be reset or incremented by a specified amount under program control. Two counters, called M and N, are not directly related to image memory addressing, but can be used to modify program flow: instructions are provided to preset, increment and test them.

Data are transferred into internal registers of each PE, processed and explicitly stored back into memory according to a RISC-like processing paradigm.

Each PE processes one pixel of each line and is composed of a Register File and a 1-bit Execution Unit. The core of the instruction set is based on morphological operators [16]: the result of an operation depends, for each processor, on the value of pixels in a given neighborhood (reduced 5×5, as sketched by the grey squares in figure 1). Data from E, EE, W and WW directions (where EE and WW denote pixels two bits apart in E and W directions) may be obtained by direct connection with neighboring PEs while all other directions correspond to data of previous lines (N, NE, NW, NN) or of subsequent lines (S, SE, SW, SS). Operations on larger neighborhoods can be decomposed [2] in chains of basic operations.

To obtain the outlined neighborhood, a number of internal registers (16, at present), called *Morphological Registers (MOR)*, have a structure which is more com-

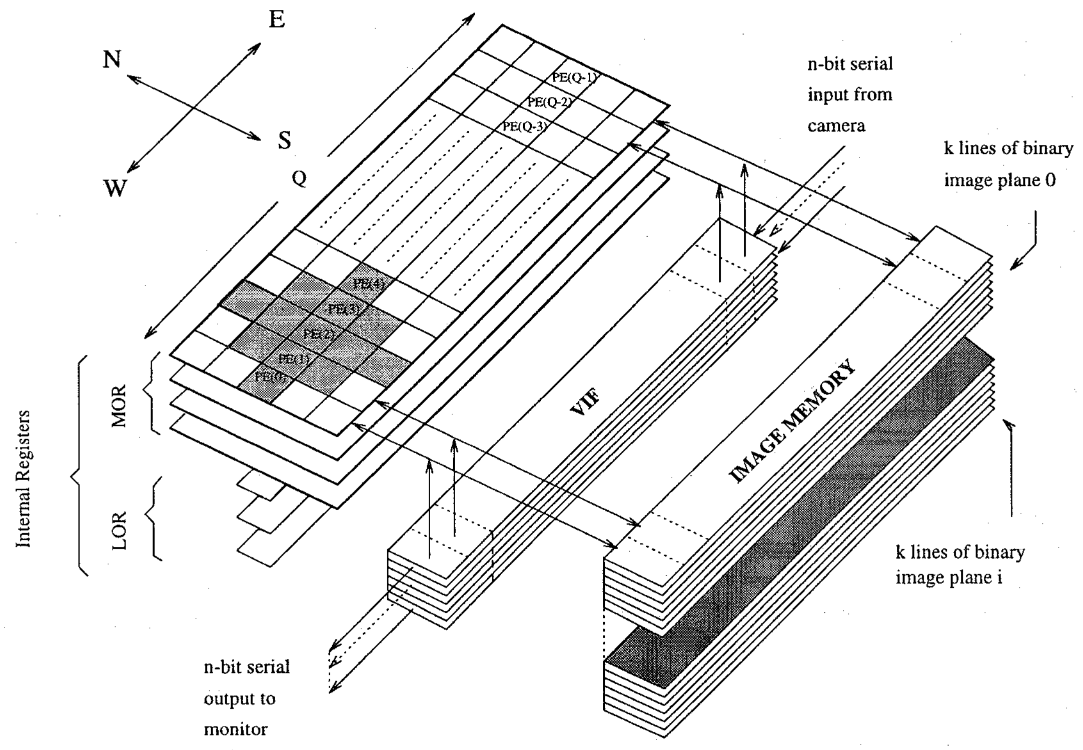
E
N
S
W
Q
PE(Q-1)
PE(Q-2)
PE(Q-3)
PE(4)
PE(3)
PE(2)
PE(1)
PE(0)
n-bit serial input from camera
k lines of binary image plane 0
Internal Registers
MOR
LOR
VIF
IMAGE MEMORY
k lines of binary image plane i
n-bit serial output to monitor

Figure 1: General architecture of the Processor Array

plex than that of a simple memory cell, and are actually composed of five 1-bit cells with a S→N shift register connection. When a load operation from memory is performed, all data are shifted northwards by one position and the south-most position is taken by the new line from memory. In this way, data from a 5×5 neighborhood are available inside the array for each PE, at the expense of a two lines latency. A second set of registers (48, at present), called *Logical Registers (LOR)*, is only 1-bit wide. On LORs and on the central bit of MORs it is possible to perform a different portion of the instruction set which comprises logical and algebraic operations.

An important characteristic of the system is the integration of a serial-to-parallel I/O device, called *Video Interface* (VIF), which can be connected to a conventional input imaging device (camera, linear array) and/or to a monitor. The interface is composed of two 8 bit, Q stages shift register which serially and asynchronously loads/stores a new line of the input/output image during the processing of the previous/next line. Two instructions enable the bidirectional transfer between the PE's internal registers and the VIF, ensuring also proper synchronization with the camera and the monitor.

Image processing algorithms consist of sequences of low level steps, such as filters, convolutions, etc., to be performed line by line over the whole image. This means that the same block of instructions has to be repeated many times, and instruction fetching from an external memory is an important source of overhead. Hence we chose to pre-load each block of instructions into an internal memory, named *Writable Control Store* (WCS, 512 words $\times$ 32 bits), and to fetch instructions from there. In this way it is possible to obtain the performance of a fast cache with a hit ratio close to 1, at a fraction of the cost and complexity.

Two inter-processor communication mechanisms are also available to exchange information among PEs which are not directly connected. The first one is the *Status Evaluation Network* (SEN), shown in figure 2, to which each processor sends the contents of one of its registers (selectable under program control) and which provides a *Status Word* divided into two subfields. The first one is composed of two global flags, named *SET* and *RESET*, which are true when the contents of the specified register are all 1s or all 0s, respectively. The second one is the *COUNT* field which is set equal to the number of processing elements in which the content of the specified register is 1. This global information may be accumulated and stored into the Status Register File and used to conditionally modify the program flow. As an alternative, it may be reloaded into the PE's Register File for further processing.

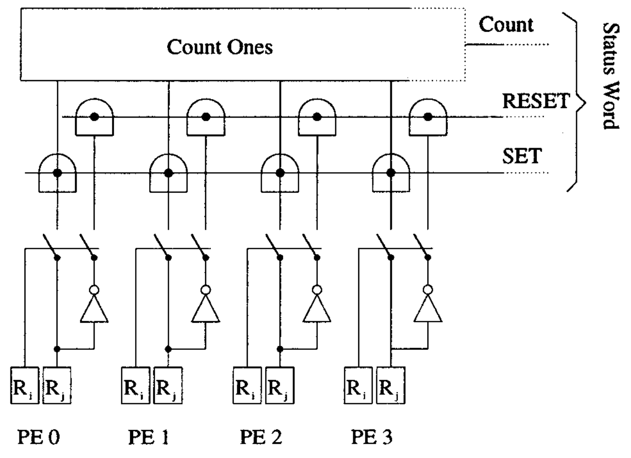

Figure 2: The Status Evaluation Network

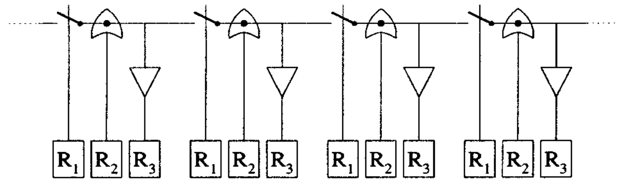

Figure 3: The Interprocessor Communication Network

The second communication mechanism is the *Interprocessor Communication Network* (ICN), shown in figure 3, which allows global and multiple communication among clusters of PEs. The topology of the communication network may be varied during program execution. In fact, each PE drives a switch that enables or disables the connection between pairs of adjacent processors. The PEs may thus be dynamically grouped into clusters, each PE can broadcast a register value to the whole cluster within a single instruction. This feature can be very useful in algorithms involving *seed-propagation* techniques, and in the emulation of pyramidal (hierarchical) processing.

The Image Memory and the Status Register File are external to the integrated circuit in PAPRICA-3, in order to provide an higher degree of modularity and flexibility at system level, while HACRE provides an internal 1K×64 bits Image Memory and 256 Status Registers. Besides being used to conditionally modify program flow, the Status Register File can be read by the external processor. This has been used in the HACRE system to implement a neural network, by computing the degree of matching between an image and a set of templates (weight matrices).

As mentioned earlier, PAPRICA-3 was conceived as a modular, high-performance, low-cost and small size image processing system. The first PAPRICA-3 board is currently under development. It will be a single PCI bus PC expansion board. It will incorporate: an array of 128 PEs (4 chips); a PCI 32 bit bus master controller implemented in FPGA; a 64K×128 bits, 20ns, Image Memory; a Status Register File (a FPGA and a memory chip); a simple PAL video camera interface (a single standard chip including an A/D converter, sync separators and pixel and line clock generators) connected to the input VIF.

3 The Programming Environment

3.1 High Level Programming Language

The intrinsic complexity of Assembly language, especially when combined with the SIMD computational model, produces a low programming efficiency; a high level programming language, together with a software tool for the generation of its corresponding Assembly code, were needed to ease the development of software applications.

Different solutions have been considered, ranging from the development of a completely new ad-hoc language to the use of a code generator. Some considerations [8] led to the choice of a code generator based on the C++ object oriented language: its only drawback is the low level of optimization achievable in a single pass (generally compilers use two passes).

The main disadvantage of a compiler-based solution lays in the need for a complete definition of a new language, which must be able to handle not only parallel statements, but also some other sequential functions (such as serial operations, data I/O, input of data from the user, memory management, etc.) which are of basic importance although not dealing with the specific SIMD extension. Conversely, a code generator may inherit these capabilities from the chosen high-level language and thus its implementation is considerably less complex than the former. Moreover, in this second case the efficiency in the development of applications is significantly increased, since the programmer does not have to learn a new language but only its parallel extensions. Moreover, the use of an object oriented language eases the programmer task since the operands acting on the new objects can be used intuitively by the application programmer thanks to the possibility of overloading known operators (such as '+', '-', '*',...).

As shown in figure 4, the complete sequence of operations required to build a PAPRICA-3 Assembly program is the following: first the extended C++ code is written and compiled; then the executable obtained is run and the PAPRICA-3 Assembly code is generated. The generation of *Assembly code* was preferred with respect to the generation of *binary machine code*, thus requiring a further processing step (PAPRICA-3 Assembler) to get the final binary code. This choice allows a faster and more efficient debugging of the code-generation functions, as well as a simpler implementation of the following global optimization step, since in the Assembly code also comments are automatically included to ease program comprehension. Moreover, since PAPRICA-3 has been conceived as a *real-time* image processor, a code optimizer can be used to further improve the generated code. Finally the PAPRICA-3 Assembler is used to produce the binary machine code.

3.2 Assembly Code Optimization

The optimization process is divided into two main steps: a deterministic one, performed on the initial assembly code, and a stochastic one, iterated on the result of the previous step. The first step is used to reduce both execution time and code size and exploits well known traditional optimization techniques [1]; on the other hand, the stochastic step tends to improve the efficiency of the Assembly code exploiting the characteristics of the pipeline structure, the specific instruction set, or, more generally, the intrinsic parallelism of each instruction [14]. This process is driven by the user choice about the cost function: execution time, code size, or a given combination of both.

3.2.1 Deterministic Optimization

This first step is aimed to the application of rules that will certainly lead to shorter and faster versions of the program. The techniques used are: Invariant Code Motion, and Removal of Jumps.

3.2.2 Stochastic Optimization

Since in the optimization phase no data are available, data-dependent branches cannot be solved (no *branch prediction* is here made). For this reason, the code is divided into *blocks*, which are defined as code segments without data-dependent conditional control-flow constructs. Each block is optimized independently of the others: a first module of the optimizer tool splits the code into blocks, and sends them to independent processes in charged of the optimization, as shown in figure 5. Finally the optimized blocks are re-assembled and the final program is generated.

Each block is optimized following an iterative stochastic approach (as opposed to traditional [9] polynomial solutions [3, 6]): at the i-th iteration, $N(i)$ new programs are generated by the consecutive application of $R(i)$ rules (drawn randomly from a given set) to each program of the $(i-1)$-th generation. The ones performing better than a given threshold $T(i)$ are kept and will form the i-th generation, while the other are discarded.

Since the population at the first iteration is composed by only the original block, at the beginning of the process (when i is small) the number of rules applied, $R(i)$, is high, thus allowing a large spreading of the initial population. As soon as i becomes large, the number of rules applied to the same block decreases to 1, for a fine tuning of the solution (see figure 6.a). A linear slope of $R(i)$ has been chosen to simplify the development of the tool, but more complex functions are under evaluation.

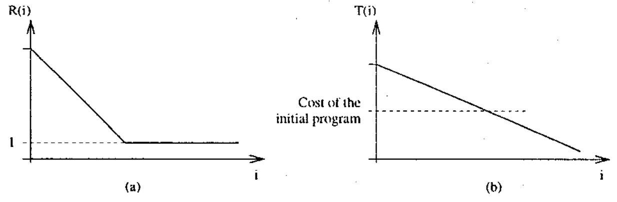

Figure 6: Qualitative behavior of *(a)* the number of rules applied consecutively to a single block, *(b)* the threshold $T(i)$ as a function of the iteration number i

Figure 6.b shows a qualitative behavior of threshold $T(i)$, which, starting from a value twice as large as the cost featured by the initial program, decreases linearly with i (the slope of $T(i)$ determines the convergence speed). This implies that at the beginning of the iterative process (i small), solutions with a cost higher than the initial can be kept (this is of basic importance to escape from local minima), while in the following iterations a threshold limits the number of candidates and keeps only the best solutions. At each iteration the block used to generate the new population is also included in the evaluation process, in order to avoid jittering from the optimal solution that may have been eventually detected. The number of iterations of the stochastic process and the slope of $T(i)$ is a function of the length of the block to be optimized: in the first version its dependence is inversely proportional to the iteration number.

The performance of each new program is deter-

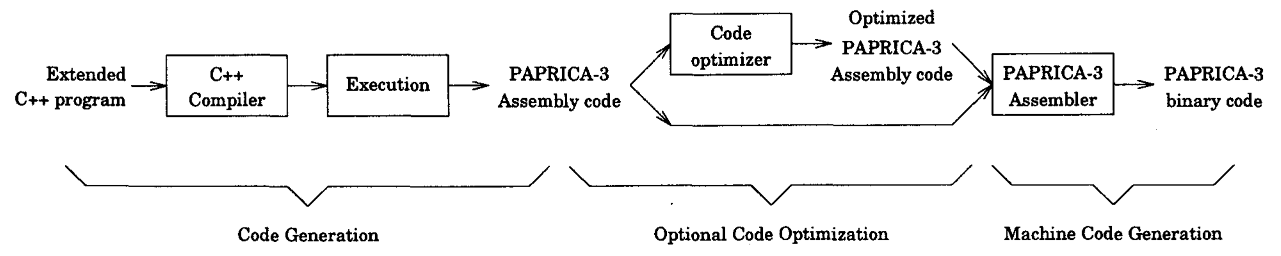

Figure 4: The complete sequence of operations required to build a PAPRICA-3 executable

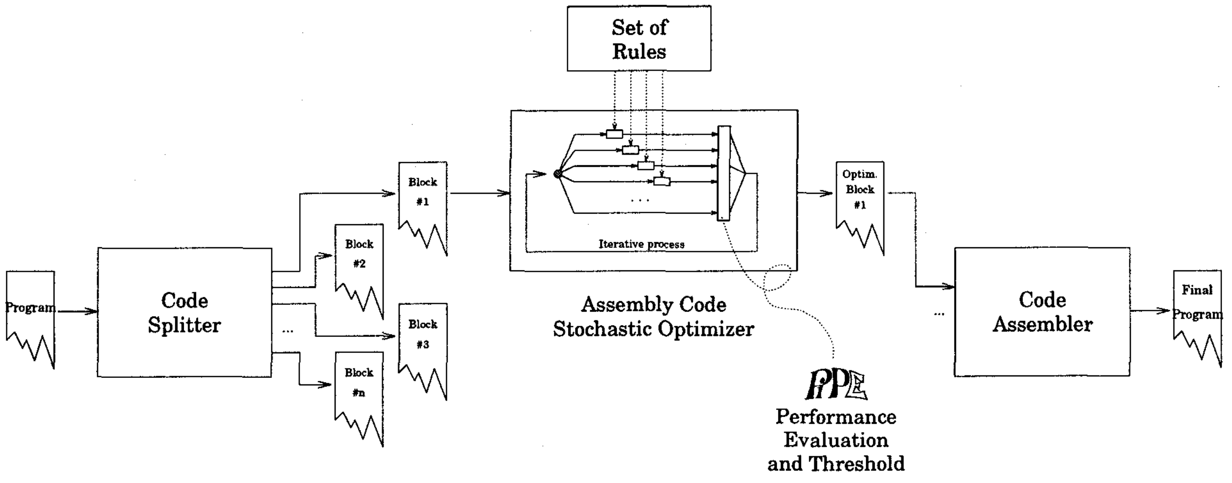

Figure 5: Block diagram of the stochastic code optimization

mined using PiPE, a tool that allows to determine the performance of a code segment (i.e. the number of clock cycles required for a complete run). Upon initialization, PiPE reads a description of the internal architecture of the processor, such as pipeline structure, structural, data, and control hazards [12], and the instruction set encoding. Then the list of instructions of a given block is sent to the PiPE tool, which computes and returns the total number of clock cycles required to run that segment of code.

Each rule can move, modify, add, or remove instructions. Every action is fully reversible, thus allowing to return to the originating state. The elementary rules used are: Instruction Scheduling, Instruction Alignment, Loop Fusion and Breaking, Loop Unrolling and Rolling, Instruction Fusion and Breaking, Multiple Assignment Removal, Common Subexpression Elimination, Registers Reallocation.

3.2.3 Parallel Implementation of the Code Optimizer

Since the major drawback of a stochastic approach to code optimization is the high computational power (in terms of both memory and CPU time), a parallel version of the optimizer has been developed. The development of the parallel version started with the analysis of the system requirements and performance:

- the CPU time spent in the evaluation of the performance of a given segment of assembly code is up to 90% of the overall CPU time;
- the larger the number of iterations, the better the quality of the result, but also the larger the amount of memory required.

Different approaches for the development of the parallel version of the optimizer were considered (code splitting and assembling is anyway a serial step which is performed by a single master processor). The approach followed in the development of the current parallel version (using standard MPI libraries) is based on the distribution of the performance evaluation step only: a master processor splits the code into blocks and generates the new blocks to be evaluated. The new blocks are queued and sent to different processors for the evaluation of their performance. When a processor completes the evaluation phase, it returns the number of clock cycles to the master node and signals its availability to evaluate a new code segment. Due to the different computational power required by (*i*) the evaluation phase (90%) and (*ii*) the rest of the processing, this approach balances the load even in the case of a non-homogeneous cluster of nodes.

The rules that are currently operative in the first version of the tool and that have been used in the examples described in this work are: Invariant Code Motion, Removal of Jumps, Instruction Scheduling, Instruction Alignment, Loop Fusion and Breaking, Loop Unrolling and Rolling, while Instruction Fusion and Breaking, Multiple Assignment Removal, Common Subexpression Substitution, Registers Reallocation, are currently under test. The future integration of these rules will lead to a higher degree of optimization.

4 Applications and Performance

This section describes a practical application of PAPRICA-3 and compares its performance with and without pipeline and with commercial processors. The application proposed is part of an automatic handwriting recognizer for bank cheques called HACRE [10], which is made of two major blocks strictly interacting: an *image preprocessor and neural recognizer* and a *context analysis subsystem.* The former is implemented on PAPRICA-3, while the latter, which uses the redundancy contained in the amount written twice to reduce error probability, runs on a commercial PC but is not described here. Refer to [10] for a complete and more detailed description.

The proposed system and its component blocks are representative of the algorithms used in a wide range of applications of image processing, such as: handwriting recognition [7, 13], mailing address interpretation, document analysis, signature verification, banking documents processing, quality control, etc.

The problem of handwriting recognition is a strategic, although quite challenging problem. Besides the classical problems encountered in reading machine-printed text, such as word and character segmentation and recognition, the domain of handwritten text recognition has to deal with other problems such as, for instance, the evident similarity of some characters with each other, the unlimited variety of writing styles and habits of different writers, and also the high variability of character shapes issued from the same writer over time. This are few of the reasons why handwriting recognition requires very powerful computing systems, mainly to deal with real-time requirements.

The image preprocessor and neural recognizer are made of three cascaded subsystems: a *mechanic-optical scanner*, to acquire a bit-map image of the cheque; an *image preprocessor* for preliminary filtering, scaling, and thresholding of the image; a *neural subsystem*, based on an ensemble of neural networks, which detect character centers and provide hypotheses of recognition, for each detected character.

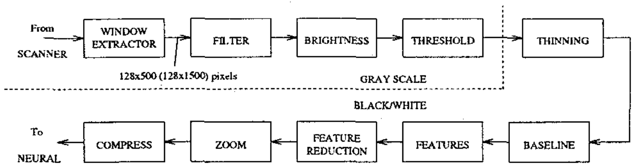

Figure 7: Block diagram of the image preprocessor and neural recognizer.

The second and third blocks are both executed on a number of PAPRICA-3 chips (two, in the prototype) to cope with the tight real-time requirements of the application (target speed is more than 10 cheques per second). As shown in fig. 7, they are made of the cascade of the blocks listed below.

1. the `WINDOW EXTRACTOR` acquires the input image from the `SCANNER`, at a resolution of 200dpi, gray scale, 16 levels. The scanner is an 876-pixel linear CCD scanned mechanically over the image, at a speed of 2m/s (equivalent to about 1,200 characters/s). Character recognition takes place in two smaller areas of known coordinates, containing the *legal* and the *courtesy amounts*, respectively. These areas, 128×500 and 128×1500 pixels in size, respectively, are extracted from the scanned image by means of an ad-hoc CCD controller. The system has to process about 2.8 Mpixel/s of data.
2. the `FILTER` block computes a simple low-pass filter with a 3×3 pixel kernel.
3. the `BRIGHTNESS` block compensates for the non-uniform detector sensitivity and paper color. A pixel-wise adaptive algorithm shifts the level of white to a pre-defined value and computes the contrast, while the image is scanned. White level and contrast are periodically damped, to reduce the influence of very bright and very black small spots.
4. the `THRESHOLD` block converts the gray-scale image, after compensation, into a B/W image; the image is compared against an adaptive threshold which is function of both the white level and the contrast.
5. the `BASELINE` block detects the *baseline* of the handwritten text, which is an horizontal stripe intersecting the text in a known position, as shown

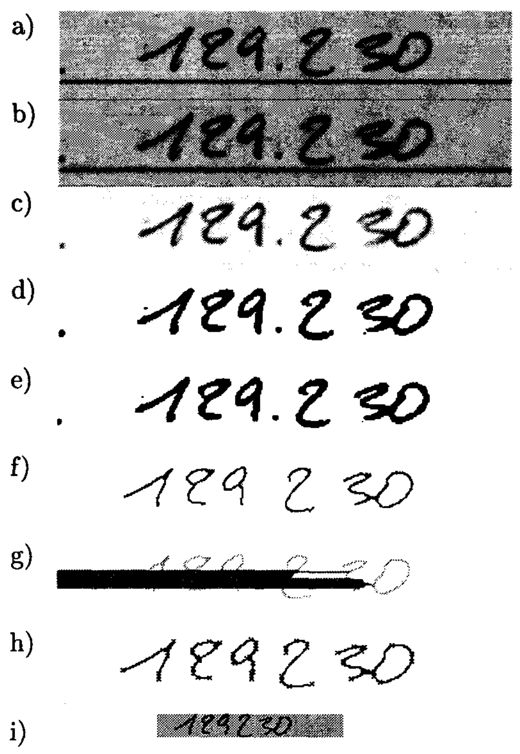

Figure 8: Preprocessing steps of handwritten images: a) original image, 200 dpi, 16 gray levels; b) low-pass filtered image; c) brightness compensation; d) thresholded image; e) spot noise removal; f) thinning, after 6 steps; g) finding baseline (at the left side of the image). h) features detection (features are tagged by small crosses). i) centering detection: neural (solid lines) and feature-based (dashed lines).

in fig. 8.g (left side). Unlike the other steps of preprocessing, baseline cannot be detected only by means of *local* algorithms (namely, algorithms with limited neighborhood), as it is a *global* parameter of the entire image, therefore it is available only at the end of the image, as shown in fig. 8.g.

6. the THINNING block reduces the width of strokes to 1 pixel. Thinning is a morphological operator [16] which reduces line width, while preserving stroke connectivity. Thinning is required to properly identify character *features* (see next block).
7. the FEATURES block detects and extracts from the image a set of 12 useful *stroke features*, which are helpful for further character recognition. Features are not used alone, but together with the neural recognizer to improve recognition reliability, as described in [10].
8. the FEATURE REDUCTION block reduces the number of features, by removing both redundant and useless features. Examples of redundant features are evident in the vertical stroke of digit "9" shown in fig. 8.h.
9. the ZOOM block evaluates the *vertical size* of the manuscript and scales it in size to approximately 25-30 pixels. Scaling the size eases the task of the neural recognizer, as the number of *weights* required for proper recognition is reduced.
10. the COMPRESS block reduces image size further by a factor 2, approximately, by means of an ad-hoc topological transformation which does not preserve image shape, although preserves its connectivity. At this point, after all preprocessing steps, the B/W image is ready for the following neural recognition steps. Each character is compressed into 252 bits, which are then reorganized into four adjacent processor memory words (4×64 bits), to optimize the performance of the neural detector.
11. the CENTERING DETECTOR takes care of detecting the location of character centers. Most handwriting recognizers [7, 13, 15] require an independent character segmenter which segments each word into individual characters. Unfortunately segmentation is a quite difficult task [7, 13], especially for handwritten texts, as there is no clear separation between consecutive characters. For this reason, we have decided to use an *integrated segmentation and recognition* (**ISR**) approach [10], in which character segmentation is tightly integrated with character recognition, and no preliminary segmentation is required. This approach has one main advantage: localization of character centers can be partially implemented by means of a neural network, therefore the system can *self-learn* to detect centers from a set of appropriate *training examples* (*training set*). This further reduces development time.

   In details, the CENTERING DETECTOR is made of two independent blocks: a *neural detector*: a two-layer ($252 \times 30 \times 3 \times 1$) WRBF+MLP network [15, 11], trained to detect character centers, and a *feature-based detector*, which attempts to locate character centers according to the distribution and occurrence frequency of character features. Fig. 8.i shows the results of the two center detectors. It is clear that each character center is localized at more than one position, but the context analysis subsystem [10] easily filters them.
12. the CHARACTER RECOGNIZER recognizes each individual character, using a neural network trained from an appropriate training set. At this stage

	PAPRICA-3		Pentium 90 MHz		Sparc 10
	worst case		morphol.	ad-hoc	morphol.
Image preprocessor	μs/line	ms/check	μs/line	μs/line	μs/line
WINDOW EXTRACTOR + FILTER	1.38	2.76	1,700	705	1,370
BRIGHTNESS	2.95	5.90	1,970	320	1,660
THRESHOLD	0.91	1.82	830	255	570
THINNING	8.34	16.7	7,390	-	7,850
BASELINE	4.24	8.28	4,320	-	3,890
FEATURES	3.05	6.10	9,490	-	10,430
ZOOM	2.24	4.48	820	160	760
COMPRESS†	30.8	61.6	21,350	-	24,950
OTHER (VARIOUS)	3.25	6.50	3,330	-	5,020
TOTAL PREPROCESSING	57.2	114	51,200	-	56,500

of processing, context is not yet considered, as it is used by the context based subsystem described in [10]. The neural network is a two-layer (252 × 75 × 1) WRBF network [15, 11], which is "triggered" for each character center detected by the CENTERING DETECTOR.

As shown in table 10, the CHARACTER RECOGNIZER is the slowest piece of code, due to the large number of synaptic weights involved. Fortunately it is computed at a relatively slow rate, namely every 15 lines, in the average, therefore its effects on computing time are limited.

Table 10 lists execution times of the various blocks and compares the performance of HACRE with and without pipelining, with and without optimization, and with a Pentium 90 and a Sparcstation 10 running the same algorithms. Figures for PAPRICA-3 are given for a system with 64 PEs, with and without using the pipeline, considering a 15ns cycle time plus 30ns memory access time, and a 50ns cycle time plus 50ns memory access time, respectively.

All programs have also been tested on both a Pentium and a Sparc, using the same algorithms based on mathematical morphology, which are well suited to the specific problem of bitmap processing and character recognition. Some of them (FILTER, BRIGHTNESS, THRESHOLD, NEURAL RECOGNIZER) could be implemented more efficiently on sequential computer using more traditional methods; these have been implemented on the Pentium and their performance listed in table 10 for comparison.

The performance of PAPRICA-3 are two to three orders of magnitude faster than that of Pentium and Sparc, for almost all the programs considered; this difference reduces slightly when the system has to crunch numbers with many bits (typically, 32 bits fixed point, or 64 bits floating points).

As the first eight programs are relatively short ones, they do not affect the performance of the whole application very much, but their performance is presented as a comparison between systems. Note that running the small programs individually exposes them to the pipeline fill up overhead (the time it takes to reach a steady state), which somewhat penalizes the pipeline solution (up to 5-10% overhead). The real speedups of those pieces of code inserted in a larger program are higher than what shown in the table.

5 Conclusions

Thanks to the specific solutions adopted, the PAPRICA-3 system fully exploits the spatial parallelism of the processor array and the intrinsic parallelism found in the sequence of assembly instructions. The code generator allows the user to develop applications without worrying about the spatial parallelism of the machine, while the assembly code optimizer is used to improve the final code exploiting instruction-level parallelism. Due to the high computational power required for code optimization, the optimizer is useful only in the final optimization of programs whose execution time is a key parameter.

Recently not only the application to bank checks has been developed but the use of PAPRICA-3 to speed-up the real-time processing of images for the automatic guidance of a passenger car [5] has been investigated as well. Images are acquired by cameras installed on board of ARGO, a Lancia Thema passenger car, and are processed in order to detect the lane position and to localize possible obstacles [4]. Within this framework, the PAPRICA-3 system is now being compared to general-purpose systems, and in particular with a Pentium 200 MHz with MMX technology. From the preliminary results obtained by the first experiments on low-level image processing, the PAPRICA-3 system is from 2 to 3 times faster than the Pentium processor. The experiments deal mainly with low-level processing of images, where both systems can take advantage of their internal structure; more precisely, the MMX pentium is being programmed directly at assembly level since up to now no public domain C compiler (GCC ver. 2.7.2) does not support MMX-based instructions.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] G. Anelli, A. Broggi, and G. Destri. Decomposition of Arbitrarily Shaped Binary Morphological Structuring Elements using Genetic Algorithms. *IEEE Transactions on PAMI*, 1997. In press.

[3] S. J. Beaty, S. Colcord, and P. H. Sweany. Using Genetic Algorithms to Fine-Tune Instruction-Scheduling Heuristics. In *Proceedings MPCS-96 - IEEE International Conference on Massively Parallel Computing Systems.* IEEE Computer Society - EuroMicro, May 6-9 1996.

[4] M. Bertozzi and A. Broggi. GOLD: a Parallel Real-Time Stereo Vision System for Generic Obstacle and Lane Detection. *IEEE Transactions on Image Processing*, 1997. In press.

[5] M. Bertozzi and A. Broggi. Vision-based Vehicle Guidance. *IEEE Computer*, 30(7):49–55, July 1997.

[6] M. J. Bourke III, P. H. Sweany, and S. J. Beaty. Extending List Scheduling to Consider Extension Frequency. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, pages 193–202. IEEE Computer Society, 1996.

[7] R. Bozinovic and S. Srihari. Off-line cursive script word recognition. *IEEE Trans. on PAMI*, 11:68–83, Jan. 1989.

[8] A. Broggi. The PAPRICA-3 Programming and Simulation Environment. In *Proceedings MPCS-96 - IEEE International Conference on Massively Parallel Computing Systems*, Ischia, Italy, May 6-9 1996. IEEE Computer Society - EuroMicro.

[9] D. J. DeWitt. *A Machine Independent Approach to the Production of Optimal Horizontal Microcode.* PhD thesis, Dept of Computer and Communication Sciences, University of Michigan, Ann Arbor, 1976.

[10] F. Gregoretti, B. Lazzerini, A. Mariani, and L. Reyneri. Mixing neural networks and context based techniques in a high-speed handwriting recognizer. *submitted to IEEE Trans. on Pattern Analysis and Machine Intelligence.*

[11] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Mc Millan College Publishing Company, 1994.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[13] F. Kunihito and W. Nobuaki. Handwritten alphanumeric character recognition by the neurocognitron. *IEEE Trans. on Neural Networks*, 2(3):355–365.

[14] B. Ramakrishna Rau and J. A. Fisher. Instruction Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, pages 9–50, 1993.

[15] L. Reyneri. Weighted radial basis functions for improved pattern recognition and signal processing. *Neural Processing Letters*, May 1995.

[16] J. Serra. *Image Analysis and Mathematical Morphology.* Academic Press, London, 1982.