Image Processing on High Performance RISC Systems

PIERPAOLO BAGLIETTO, MASSIMO MARESCA, MAURO MIGLIARDI, AND NICOLA ZINGIRIAN, ASSOCIATE MEMBER, IEEE

Invited Paper

The recent progress of RISC technology has led to the feeling that a significant percentage of image processing applications, which in the past required the use of special purpose computer architectures or of "ad hoc" hardware, can now be implemented in software on low cost general purpose platforms. We decided to undertake the study described in this paper to understand the extent to which this feeling corresponds to reality . We selected a set of reference RISC based systems to represent RISC technology, and identified a set of basic image processing tasks to represent the image processing domain. We measured the performance and studied the behaviour of the reference systems in the execution of the basic image processing tasks by running a number of experiments based on different program organizations. The results of these experiments are summarized in a table, which can be used by image processing application designers to evaluate whether RISC based platforms are able to deliver the computing power required for a specific application.

The study of the reference system behaviour led us to draw the following conclusions. First, unless special programming solutions are adopted, image processing programs turn out to be extremely inefficient on RISC based systems. This is due to the fact that present generation optimizing compilers are not able to compile image processing programs into efficient machine code.

Second, while computer architecture has evolved from the original flat organization towards a more complex organization, based, for example, on memory hierarchy and instruction level parallelism, the programming model upon which high level languages (e.g., C, Pascal) are based has not evolved accordingly. As a consequence programmers are forced to adopt special programming solutions and tricks to bridge the gap between architecture and programming model to improve efficiency.

Third, although processing speed has grown up much faster than memory access speed, in current generation single processor RISC systems image processing can still be considered compute-bound. As a consequence, improvements in processing speed (originated for example by a higher degree of parallelism) will yield improvements of an equal factor in applications.

Manuscript received xxxx, 1995; revised February 1996. This work was supported in part by research grants from the European Community under the ESPRIT BRA Project 8849-SM-IMP.

P. Baglietto, M. Migliardi are with DIST - University of Genova, Genova, Italy.

M. Maresca and N. Zingirian are with DEI - University of Padova, Padova, Italy.

Publisher Item Identifier S 0018-9219(96)04995-X.

1 INTRODUCTION

The rapid progress of RISC technology [13][41][33][46] has recently given a new impulse to image processing and pattern recognition (IPPR). The availability of a computing power of the same order of magnitude as that previously delivered by massively parallel computers on low cost RISC systems presently makes software based IPPR effective and convenient in a number of applications which were not even targeted in the past because of the prohibitive cost of the hardware required. As a consequence of this evolution, traditional high end applications of IPPR, for example in the area of automatic document recognition and classification, have been ported to RISC based platforms [1] and successfully installed in a number of small and medium size document processing centers, and novel applications of IPPR, based for example on digital video processing [23], have been introduced and have quickly found wide acceptance.

IPPR is one of the traditional application domains of massively parallel computers [24][45] because of the size and of the regularity of the data structures involved, typically consisting of large two-dimensional arrays of picture elements (pixels), and because of the characteristics of the algorithms used, typically consisting of the execution of the same set of operations on all the pixels and of the combination of each pixel with a small set of pixels located at a short distance.

It was indeed the characteristics of image processing that stimulated most of theoretical research and experiments in massively parallel computer architectures. Such research and experiments, which largely developed and progressed in the seventies and in the eighties, produced a variety of scientific results in the form of parallel algorithms [20], parallel architectures [42] and prototypes of parallel computers (e.g., CLIP [10], MPP [27], CM[16], IUA[44]).

Unfortunately the cost and the programming complexity of massively parallel computers did not favour the diffusion of IPPR in real applications, as only a limited number of companies and institutions were able to afford the cost of dedicated parallel hardware and the cost of developing machine dependent algorithms and programs. It was in the

second half of the eighties that the technology of RISC microprocessors and systems, driven by the huge market of desktop workstations and servers, reached a point at which the performance of IPPR programs, coded in a standard sequential language such as C, became high enough to allow for the implementation of cost effective RISC based IPPR applications.

The possibility of using a general purpose RISC system instead of an expensive special purpose massively parallel computer for IPPR soon cooled down the interest in special purpose parallel architectures. The feeling that any IPPR application could be run on a RISC, or at worst on a few RISCs, gradually pervaded the scientific community and led to the conclusion that special purpose parallel architectures were not going to be necessary any more for IPPR applications.

Such a conclusion has not been substantiated so far by an analytical investigation of the performance and efficiency of RISC systems in IPPR applications, as it was done, on the contrary, in the application domain of scientific computing [7][19][8]. This is exactly the subject of this paper: the specific objectives of the study presented in this paper are *i)* the characterization of the level of performance of IPPR applications on RISC systems, *ii)* the analysis of the behaviour of the main components of RISC systems (i.e., CPU, primary memory, cache) in the execution of IPPR programs, and *iii)* the investigation of program organization techniques aimed at improving the performance of IPPR applications on RISC systems.

The study was carried out as follows. We selected a set of reference families of RISC based desktop systems, among those commercially available, namely from Sun Microsystems, Digital Equipment Corp., Silicon Graphics Inc., International Business Machines, and Hewlett Packard and chose the top level models of these families as reference systems. We classified the reference systems in terms of their main architectural characteristics, which include general information on the CPU (for example clock speed and presence/absence of dedicated hardware units), specific information on the pipeline organization (for example instruction latency and throughput), and information on the memory hierarchy organization (for example cache size and throughput).

We then selected four basic tasks to represent the IPPR application domain, namely Discrete Cosine Transform [28] and Full Search Block Matching [18][29], used in video compression [23], and Convolution [31] and Hough Transform [3], used in vision and pattern recognition. We programmed each basic task on each reference architecture starting from a plain derivation of the code from the task definition, and gradually adopting more and more sophisticated programming solutions to improve its performance. We studied the effect of such programming solutions on the behaviour and on the performance of the reference systems by measuring the throughput improvements and by inspecting the assembly code produced by the compiler in order to identify the sources of inefficiency.

The experiments led to two results, which are the main contributions of this paper, namely *i)* the measurement of the performance of the basic IPPR tasks on the reference systems and *ii)* the analysis of the behaviour of such tasks, and by induction of IPPR in general, in RISC systems. The measurement of the performances of the basic IPPR tasks on the reference systems allows us to identify the class of IPPR applications that can be actually supported by RISC systems and the level of performance that can be achieved using current technology. The analysis of the behaviour of IPPR programs in RISC systems allows us to locate the main processing bottle-necks and leads to the identification of a number of source level program optimization techniques to improve efficiency.

The paper is organized as follows. In Section 2 we present, analyze and compare the reference systems by means of a set of parameters which capture their architectural characteristics. In Section 3 we introduce the set of IPPR basic tasks. In Section 4 we describe the experiments and present the results of the performance analysis of the basic tasks on the reference systems. In Section 5 we discuss the results of the experiments, and in Section 6 we provide some concluding remarks.

2 THE REFERENCE SYSTEMS

We selected some of the top level models of high performance desktop workstations to represent RISC architecture and technology. We restricted our analysis to single processor systems as we are only interested in studying the behaviour of RISC architectures in IPPR and not how IPPR can be parallelized on RISC based multiprocessors. The reference systems selected are SUN Microsystems' SPARCstation 20 model 61 [40], Hewlett Packard's HP 9000 Series 700 model 735/125 [14], Digital Equipment Corporation's DEC 3000 model 800S [5], Silicon Graphics' Indigo2 Impact [34], and International Business Machines' RISC System/6000 model 43P [17]. In the rest of the paper we will refer to these systems by means of the name of their manufacturers, namely SUN, HP, DEC, SGI and IBM.

The reference systems are based on five different CPUs, namely SuperSPARC [37][39], HPPA 7150 [15][2], DECchip 21064 (Alpha) [6][26], MIPS R4400 [12] and PowerPC 604
[36], which cover the two directions of instruction level parallelism, i.e. pipelining and scalarity, and feature different memory hierarchy organization.

Table 1 summarizes the characteristics of the reference systems. The *System Software* Section includes the identification of the operating system under which we ran our experiments and of the compilers that we used to build the experimental programs. The *General CPU Parameters* Section includes the main CPU features, such as the CPU clock frequency, the number of general purpose and floating point registers available, the number of instructions that can be issued concurrently, the way floating point to integer and integer to floating point conversions are carried

System Name		HP 735/125	DEC AXP 3000/800S	IBM RS6000 43P	SGI Indigo2 Impact	SUN Sparc 20/61
System Software						
1	Operating system	HP UX 9.05	OSF1 v. 3.2	AIX 4.1.1	IRIX 5.3	SunOS 4.1.4
2	C Compilers	gcc 2.6/HP cc	gcc 2.6/DEC cc	gcc 2.6/IBM xlc	gcc 2.6/MIPS cc	gcc 2.6/SUN acc
General CPU Parameters						
3	CPU name	HPPA 7150	DECchip 21064	PowerPC 604	MIPS R4400	SuperSparc
4	Year of availability	1994	1992	1994	1994	1992
5	Clock frequency	125 MHz	200 MHz	100MHz	250 MHz	60 MHz
6	CPU cycle time	8 ns	5 ns	10 ns	4 ns	16.6 ns
7	Data path width	32 bits	64 bits	32 bits	64 bits	32 bits
8	Number of general purpose registers	32×32 bits	32×32 bits	32×64 bits	32×32 bits	32×32 bits
9	Number of floating point registers	64×32 or 32×64 bits	32×64 bits	32×64 bits	32×64 bits	32×32 or 16×64 bits
10	Number of instructions issued	2	2	4	1	3
11	Integer-floating point unit interaction	Through memory	Through memory	Through memory	Direct	Through memory
12	Time for an int to float conversion	65 ns	56 ns	174 ns	24 ns	67 ns
Instruction execution latency						
13	Integer (32 bits) sum	1 cycle	1 cycle	1 cycle	1 cycle	1 cycle
14	Integer (32 bits) multiply	10 cycle	21 cycles	4 cycles	13 cycles	5 cycles
15	Double (64 bits) sum	2 cycles	6 cycles	4 cycles	4 cycles	3 cycles
16	Double (64 bits) multiply	2 cycles	6 cycles	5 cycles	8 cycles	3 cycles
Instruction Throughput						
17	Integer (32 bits) sum	1 cycle	1 cycle	½ cycle	1 cycle	½ cycle
18	Integer (32 bits) multiply	10 cycles	19 cycles	2 cycles	13 cycles	5 cycles
19	Double (64 bits) sum	1 cycle	1 cycle	1 cycle	3 cycles	1 cycle
20	Double (64 bits) multiply	1 cycle	1 cycle	1 cycle	4 cycles	1 cycle
Memory system						
21	Primary memory	64 Mbytes	64 Mbytes	64 Mbytes	64 Mbytes	128 Mbytes
22	RAM access time (read line)	360 ns	200 ns	366 ns	1162 ns	365 ns
Instruction cache						
23	Size	256 Kbytes	8 Kbytes	16 Kbytes	16 Kbytes	20 Kbytes
24	Organization	Direct mapped	Direct mapped	Four way set associative	Direct mapped	Five way set associative
25	Access Time	8 ns	5 ns	10 ns	4 ns	16.6 ns
Data cache						
26	Size	256 Kbytes	8 Kbytes	16 Kbytes	16 Kbytes	16 Kbytes
27	Line size	32 bytes	32 bytes	32 bytes	16 bytes	32 bytes
28	Organization	Direct mapped	Direct mapped	Four way set associative	Direct mapped	Four way set associative
29	Access Time (read int)	16 ns	5 ns	10 ns	4 ns	16.6 ns
30	Access Time (write int)	16 ns	5 ns	10 ns	4 ns	16.6 ns
31	Access Time (read floating point)	24 ns	5 ns	10 ns	4 ns	16.6 ns
32	Access Time (write floating point)	16 ns	5 ns	10 ns	4 ns	16.6 ns
33	Data write protocol	Copy back	Write through	Copy back	Write through	Copy back
Second Level cache						
34	Size	NA	2 Mbytes	256 Kbytes	2 Mbytes	1 Mbyte
35	Line size	NA	32 bytes	32 bytes	128 bytes	128 bytes
36	Access Time	NA	70 ns	153 ns	85 ns	151 ns

Table 1 Reference systems feature summary.

out and the time such conversions require[1]. The *Instruction Latency* Section and the *Instruction Throughput* Section report the values of the parameters related to instruction processing such as the number of cycles necessary to complete a single operation and the maximum operation throughput. The *Memory System* Section reports the values of the parameters related to the organization and to the performance of memory.

3 THE IPPR BASIC TASKS

The most typical tasks of IPPR are those which require the processing of images[2] and of lists of pixels. Based on

[1] We decided to include this parameter, considering that in C programs, in which type conversions are implicit and not immediately visible, floating point to integer and integer to floating point conversions take a not negligible percentage of time.

[2] In this paper the term image is referred to a two-dimensional array of pixels.

	Low level	Intermediate level
Local processing	Convolution [31]	Full Search Block Matching [18]
Global processing	Discrete Cosine Transform [22]	Hough Transform [3]

Table 2 - Basic IPPR tasks.

TASK	Definition	Complexity	Operations
Convolution	$im^{out}_{x+\frac{m}{2},y+\frac{m}{2}} = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} mask_{i,j}\, im^{in}_{x+i,y+j}$	$O(n^2m^2)$ *m = size of conv. mask*	int mul int add
Discrete Cosine Transform	$im^{out}_{x,y} = C_x C_y \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} im^{in}_{i,j}\, f(i,j,x,y)$ $f(i,j,x,y) = \cos(c_2 ix+c_3)\cos(c_2 jx+c_3)$	$O(n^4)$	float mul float add cos, sin
Hough Transform.	$H_{r,k} = \sum_{(x,y)\in L(r,k)} 1$ $L(r,k) = \{(x,y) \mid x\cos k\theta + y\sin k\theta = r$ and $im^{in}_{x,y} = \text{object\_pixel}\}$	$O(ml)$ *m = number of angles* *l = number of edge pixels*	float mul float add cos sin float-to-int conversion
Full Search Block Matching	$mad_{i,j} = \sum_{x=0}^{m-1}\sum_{y=0}^{m-1} \lvert im^{reference}_{x+i,y+j} - im^{current}_{x,y} \rvert$	$O(n^2d^2)$ *d = maximum displacement*	int sub abs int sum

Table 3 - Basic IPPR tasks: definition, complexity and type of operations required.

such an assumption, we decided to restrict our analysis to *low level* and *intermediate level* image processing [3]: l*ow level* image processing tasks receive images at their input, process them, and produce images at their output whereas *intermediate level* image processing tasks receive images or lists of pixels at their input, process them, and produce a set of features, not necessarily in the form of an image, at their output.

The operations that can be performed on images can be classified as *point operations*, if the value of every element of the output data set is a function only of the value of one pixel of the input image, *local operations*, if the value of every element of the output data set is a function only of the values of a limited number of pixels located inside a well defined area in the input image, or *global operations*, if the value of every element of the output data set is a function of all the pixels of the input image. We decided to neglect *point operations*, which are less computing demanding than the other two categories, and to focus on *local operations* and *global operations*. According to these guidelines, we selected the basic tasks shown in Table 2 for our analysis.

Table 3 summarizes the characteristics, the complexity and the type of operations of the basics IPPR tasks. For each of them we now give a short description, show the

basic sequential algorithm[3], and report the computational complexity.

3.1 Low Level Image Processing Tasks

Two-dimensional convolution (hereafter CONV) and two-dimensional Discrete Cosine Transform (hereafter DCT) were selected to represent *low level* image processing. CONV only requires local processing while DCT requires global processing.

3.1.1 Two-dimensional Convolution (CONV)

CONV performs the spatial linear filtering of an image. The effect of filtering is to reduce or to emphasize the amplitude of certain spatial frequencies. Applications of CONV include image enhancement in general as well as, depending on the convolution mask, edge detection [31], regularization [4], morphological operations [25]. In the implementation of CONV a matrix of coefficients or weights, called mask, slides over the input image, covering a different image region at each shift. At each shift, the coefficients of the mask are combined with the image to produce a weighted average value of the pixels currently covered: the result is assigned to the output pixel located in the center of the covered region. The basic sequential algorithm for CONV is the following[4]:

```
for each mask position (x,y)
    begin
    for each mask element (i,j)
        acc:=acc+input_image[x+i][y+j]*mask[i][j];
    output_image[x+mask_dim div 2]
                                [y+mask_dim div 2]:=acc;
    end
```

Indexes *x* and *y* control the shifting of the mask over the input image, whereas indexes *i* and *j* scan the portion of the input image which overlaps with the convolution mask. During such a scan the weighted average value of the input image pixels is computed and accumulated in *acc*: the result is then moved to *output_image*. The computational complexity of CONV is $O(n^2m^2)$, where $n \times n$ is the image size and $m \times m$ is the mask size.

3.1.2 Discrete Cosine Transform (DCT)

The Cosine Transform performs the computation of the coefficients of the different spatial frequency components of an image. Its advantage, with respect to the Fourier Transform, is that its discretization (i.e. the DCT) does not cause periodical discontinuities on the spatial domain as it happens in the Discrete Fourier Transform.

DCT is used in image coding, both for still image compression, applied to the source image, and for full motion video compression, applied to the difference between a video frame and its prediction. The basic sequential algorithm for DCT is shown below.

```
for each DCT element (i, j)
    begin
    for each image pixel (x, y)
        acc:=acc+c[i]*c[j]*in_image[x][y]*cos[c2*j*x+c3]
                                          *cos[c2*y*j+c3];
    out_image[i][j]:=acc;
    end
```

The computational complexity of DCT is $O(n^4)$, where *n* is the size of the image or of the image block to be transformed.

3.2 Intermediate Level Image Processing Tasks

Full Search Block Matching (in the following denoted by FSBM) and Hough Transform (in the following denoted by HT) were selected to represent *intermediate level* image processing. FSBM only requires local processing while HT requires global processing.

3.2.1 Full Search Block Matching (FSBM)

FSBM is the most straightforward way to implement motion estimation, which is the most time consuming part of interframe video compression. Motion estimation in a stream of video frames is carried out by partitioning the current frame in blocks of pixels of fixed size and, for each of these blocks, by looking for the most similar block of pixels in another frame, called reference frame. Once this search is completed , each block of the current frame can be coded by means of a vector associated to the difference between its position in the current frame and the position of the corresponding block in the reference frame[5]. This vector is named motion vector.

FSBM is the basic technique used to compute the block most similar to a reference block of the current frame within a search area of the reference frame. It considers all the blocks, called candidate blocks, contained within a

---

[3] In our terminology the term *task* is referred to a specific piece of work that must be performed, while the term *algorithm* is referred to the way a task is carried out. There are many algorithms that carry out a given task: the "basic sequential algorithm" is the algorithm which is most directly derivable from the task definition.

[4] We ignore the issues related to the processing of image borders, which require special processing, as we are interested in the computational issues and not in the completeness of the results of the algorithm.

[5] To be precise the code includes the motion vector and the difference between the actual block and its estimation obtained applying the motion vector to the reference frame block.

search area of the reference frame for comparison[6], as is shown in the following pseudo-code.

```
for each block in the current frame
    begin
    mad := MAX_INT      /* mad stands for Mean Absolute
    Difference */
    for each candidate block in the reference frame
                                    within given displacement
        for each pixel
            curmad := curmad + |reference pixel
                                              - candidate pixel|
        if (curmad < mad)
            begin
            mad := curmad
            update the motion vector
            end
    end
```

For each reference block, all the surrounding blocks within a given search area are considered, and the position of the one that best matches the reference block is used to compute the motion vector. The matching criterion used to measure how well a candidate block matches the reference block is the Mean Absolute Difference (MAD), which requires the execution of simple operations (subtraction and absolute value). The computational complexity of FSBM is $O(n^2d^2)$, where n is the size of the image and d is the maximum displacement considered.

3.2.2 Hough Transform (HT)

HT is a typical task of pattern recognition aimed at detecting lines and curves in binary images. In our formulation HT processes a list of edge pixels extracted from an image (for example by means of convolution) and produces a matrix H, each element of which corresponds to a straight line in the input image, as a result. In particular, element $H_{r,k}$ corresponds to a straight line L identified by parameters r and k, where r is the distance between L and the origin and k is proportional to the angle between the x-axis and a straight line orthogonal to L. $H_{r,k}$ contains the number of edge pixels aligned along straight line $L(r, k)$. The basic sequential algorithm for HT is shown below:

```
for each angle kΔθ
    begin
    for each edge pixel (x,y)
        begin
        r:= y * sin (kΔθ) + x * cos (kΔθ);
        inc(H[r][k]);
        end
    end
```

For each angle $k\Delta\theta$ all the edge pixels are processed. For each edge pixel (x,y) the value of r such that straight line $L(r,k\Delta\theta)$ intersects the pixel is found and the value of output matrix H at position (r,k) is incremented. The computational complexity of HT is $O(ml)$, where m is the number of angles considered and l is the number of edge pixels.

4 Performance of the IPPR Basic Tasks on the Reference Systems

We ran a number of experiments to measure the performance of the IPPR basic tasks on the reference systems and to observe their behaviour. A preliminary set of experiments led us to identify the following sources of inefficiency:

1) The computing intensive parts of IPPR programs tend to be organized as processing loops of limited size: in such pieces of code the loop control processing overhead takes a significant percentage of the total processing time.
2) The iterative organization of IPPR programs tends to generate a large number of data hazards[7] in pipelined architectures.
3) Because of the presence/absence of specific functional units (e.g., integer/floating point units) and data paths (e.g., from integer to floating point registers and viceversa), the use of the data types which naturally match the task characteristics in the source programs may turn out to be not the most convenient solution.
4) The use of arrays to keep large tables of pre-computed functions (e.g., trigonometric functions) in memory for fast on line access may introduce unnecessary load/store instructions, which slow down program execution; the most effective mechanism to maintain these tables (e.g., arrays, variables or constants) depends both on the characteristics of the host architecture and on the compiler used.
5) The possibility to plan the order in which the input data structures (e.g., the images), are accessed allows defining program and data partitioning strategies which depend on the size of the cache memory, aimed at reducing the average latency of memory accesses.

We demonstrated, through a second set of experiments, that the native compilers of the reference systems and the GNU C compiler for the reference systems are not able to eliminate these sources of inefficiency. On the contrary, we demonstrated that the sources of inefficiency can be eliminated only through manual optimization of source programs. More specifically, we experimented with the following source level optimizations:

1) *Loop Unrolling* - LU consists of transforming a loop in

[6] Since FSBM is highly computing intensive, other less computing demanding (and of course sub-optimal) algorithms have been proposed and are used in software implementations of compression algorithms on workstations. We focus on FSBM as we are interested only in the computational aspects of IPPR and not in innovative algorithmic solutions.

[7] Data hazards take place when an instruction needs the result of an immediately previous instruction as an input.

such a way to increase the loop body size and to decrease the number of iterations. LU reduces both the number of load/store instructions in the programs, thanks to a better utilization of the CPU registers, and the effect of the data hazards, by increasing the number of instructions in each loop iteration, thus allowing the compiler optimizer to schedule the instructions more efficiently.

2) *Data Type Optimization* - DTO consists of choosing the data types for the variables in the program critical path using as a criterion the performance of the different functional units instead of using the data types naturally deriving from the task definition. DTO improves efficiency by forcing the data to flow through the fastest CPU data paths.
3) *Table Access Optimization* - TAO consists of selecting the most convenient mechanism for keeping tables of pre-computed data in memory, thus reducing the time required to access such tables inside the program critical path.
4) *Cache Access Optimization* - CAO consists of partitioning the programs in such a way to reduce the traffic between primary memory and cache memory.

In the following subsections we describe the rationale of our experiments and the methodology adopted. We then describe the source level program optimizations in details and present the performance results obtained.

4.1 Rationale

First of all it is important to remark that the goal of our experiments is not to rank the reference systems in IPPR applications. The main reason why a ranking of that kind would make no sense is that the evolution of RISC technology is so fast and, as a consequence, the rate at which the families of reference systems are enriched with new models is so high that any ranking would be subject to a complete revision every few months (see [11] for an example of a new CPU which was announced during the development of our experiments). A second reason why ranking the reference systems in IPPR would make no sense is that the architecture of desktop workstations is presently migrating toward a more composite organization, in which dedicated devices take care of the processing of video and images instead of loading the CPU.

Our investigation and experiments aim instead at observing the behaviour and evaluating the performance of the basic RISC architecture, which includes CPU and memory, with no support of hardware accelerators, in IPPR. Running IPPR applications on RISC systems not equipped with hardware accelerators allows the evaluation of the extent to which the RISC architectural concepts suit IPPR. The results of such an evaluation can be used, for example, to guide the design of RISC based hardware accelerators for IPPR.

4.2 Methodology

For each basic task and for each reference system we performed the following steps:

1) we identified the algorithm most directly derivable from the basic task definition and wrote the corresponding C program;
2) we compiled such a program using the most advanced compilers provided by the manufacturers as well as the GNU C compiler (see Table 1), activating the most aggressive optimization options [38][35].
3) we measured the performance of the object code on test input data set of fixed size: the performance measurements were based on the *gettimeofday()* system call provided by the Unix operating system, for each test repeating the measurements several times and selecting the minimum time obtained[8];
4) we analyzed the performance measurements, the source code and the assembly code generated by the compiler jointly, in order to identify the percentages of time spent in the different sections of the source program;
5) we manipulated the source program in a variety of different ways which appeared to be convenient, to improve its performance;
6) we repeated steps 3), 4) and 5) until we found the best performance for that basic task;
7) we interpreted the results of our measurements and actions.

In the next Section we present the four source level optimizations using the four basic tasks as case studies. In particular LU is presented using FSBM as a case study, DTO is presented using CONV as a case study, TAO is presented using DCT as a case study, and CAO is presented using HT as a case study.

4.3 Experiments

4.3.1 Full Search Block Matching (FSBM)

Fig. 1 shows the C program directly derived from the definition given in 3.2.1, which computes the mean absolute difference between a block of an image and the surrounding blocks, within displacement `DISP`. The code matches a block of size `BLOCK_SIZE` × `BLOCK_SIZE` of the current picture (variable `cur_pic`) with `2` × `DISP+1` blocks of the same size in a reference picture (variable `ref_pic`) and produces the components of the motion vector (variables `min_u` and `min_v`) as a result. The code shown in Fig. 1 is supposed to be part of a loop which scans all the blocks of the current image. Table 4 (row 1) shows the performance of the reference systems in the execution of such a code.

4.3.1.1 Optimization of FSBM

The code shown in Fig. 1 exhibits poor performance in RISC systems mainly because of the limited size of the internal loop body, which consists of only one statement and is executed a large number of times. Such a

[8] Times larger than minimum include terms due to operating system overhead.

characteristic leads to the following two negative effects:

1. it is not possible to take advantage of sophisticated instruction scheduling algorithms to generate the code corresponding to the loop body, considering the low number of machine instructions to be scheduled: as a consequence some of the data hazards cannot be eliminated and the utilization factor of the hardware functional units remains low.
2. it is possible to take advantage only of a limited number of CPU registers to allocate variables or results of expressions repeatedly used: as a consequence most registers are not even accessed inside the loop and a large number of unnecessary memory accesses are performed.

The first negative effect can be eliminated by Internal Loop Unrolling (ILU). ILU consists of collapsing some iterations of the most internal loop in one statement (see Figure 4.b for an example). This new statement is larger and more complex than the original loop body and requires a higher number of machine instructions to be executed. The higher number of machine instructions in the new loop body allows the optimizer to take advantage of its sophisticated scheduling algorithms and, as a consequence, to come up with a more efficient schedule.

```
1:   min_v = 0;
2:   min_u = 0;
3:   mad = MAXINT;                          /* Mean Absolute Difference */
4:   for (u = -DISP; u <= DISP; u++)
5:      for (v = -DISP; v <= DISP; v++) {     /*for each candidate block*/
6:       cur_mad = 0;
7:       for (i = 0; i < BLOCK_SIZE; i++)
8:          for (j = 0; j < BLOCK_SIZE; j++)     /* for each pixel */
9:           cur_mad+=abs(cur_pic[y+i][x+j]-ref_pic[y+DISP+i+u][x+DISP+j+v]);
10:      if (cur_mad < mad ) {
11:         mad = cur_mad;
12:         min_v = v;
13:         min_u = u;
14:         }
15:      }
```

Figure 1 - FSBM: code directly derived from task definition.

```
10:  min_v = 0;
11:  min_u = 0;
12:  mad = MAXINT;
13:  for (u = -DISP; u <= DISP; u++) {      /* for each candidate
                                                 out of 2*DISP+1 */
14:     cmad0 = .........= cmad16 = 0;      /*2*DISP+1 accumulators */
15:     for (i = 0; i < BLOCK_SIZE; i++)
16:        for (j = 0; j < BLOCK_SIZE; j+=4) { /* for each pixel out of 4 */
17:           r0 = cur_pic[y+i][x+j]; /* 4 pixels per iteration*/
              ..........
18:           r3 = cur_pic[y+i][x+j+3];
19:           cur_mad+=abs(cur_pic[y+i][x+j]-ref_pic[y+DISP+i+u][x+  +j]);

20:           cmad0+=     (     abs(r0-ref_pic[y+DISP+i+u][x+  j])
21:                       +  abs(r1-ref_pic[y+DISP+i+u][x+1+j]))
22:                   +  (     abs(r2-ref_pic[y+DISP+i+u][x+2+j])
23:                       +  abs(r3-ref_pic[y+DISP+i+u][x+3+j]));
              ...............
24:           cmad16+=    (  abs(r0-ref_pic[y+DISP+i+u][x+16+j])
25:                       +  abs(r1-ref_pic[y+DISP+i+u][x+17+j]))
26:                   +  (     abs(r2-ref_pic[y+DISP+i+u][x+18+j])
27:                       +  abs(r3-ref_pic[y+DISP+i+u][x+19+j]));
28:           }
29:     if (cmad0 < mad) {   /*selects the best new mad and motion vector*/
30:        mad = cmad0;
31:        min_v = -8;
32:        min_u = u;
33:        }
           ............
34:     if (cmad16 < mad) {
35:        mad = cmad16;
36:        min_v = 8;
37:        min_u = u;
38:        }
39:     }
```

Figure 2 - LU applied to FSBM.

	HP	**DEC**	**IBM**	**SGI**	**SUN**
Code derived from task definition:	7,467[g] ms	8,521[c] ms	3,631[c] ms	4,177[g] ms	8,287[g] ms
Best performance of optimized code:	2,854[g] ms	2,205[g] ms	1,764[g] ms	1,202[g] ms	3,586[g] ms
	LU: ELU 17 ILU 8	LU: ELU 17 ILU 16	LU: ELU 17 ILU 8	LU: ELU 17 ILU 8	LU: ELU 17 ILU 8

Operating conditions: image size: 512×512
block size: 16×16
displacement: 8
compilers:: c = native compiler, g = GNU compiler.

Table 4 - Performance of FSBM.

The second negative effect can be eliminated by External Loop Unrolling (ELU). ELU consists of moving iterations from outer loops to inner loops (see Figure 4.a for an example). Using ELU, several threads of computations are created and simultaneously progress at every iteration of the internal loop: as long as there are registers available to be assigned to these threads of computation, the global efficiency improves.

Fig. 2 shows the organization of the FSBM code resulting from the application of both ILU and ELU. The presence of more than one statement in the internal loop (`cmad0+= ..., cmad1+=..., ...`) is due to ELU, while the size of each of such statements is due to ILU.

Table 4 (row 2) shows the best performance obtained in each reference system along with the loop unrolling factor. DTO, TAO and CAO are not applicable in FSBM.

4.3.2 Convolution (CONV)

Fig. 3 shows how the C program for two-dimensional convolution directly derived from the definition given in 3.2. Table 5 (row 1) presents the performance of such a program on the reference systems.

4.3.2.1 Optimization of CONV

In order to improve the performance of convolution, LU can be effectively applied. The application of LU in CONV is not different from its application in FSBM, as presented in Section 4.3.1.1, and leads to a code which follows the general scheme shown in Fig. 4. The choice of different factors for ELU and ILU allows the derivation of specific programs from such a general scheme.

In addition to LU, DTO can be used to speed up CONV. From Table 1 it appears that, in general, the reference systems are faster in floating point processing than in integer processing. This fact suggests to explore whether it may be convenient to carry out CONV in floating point rather than in integer, as it would seem natural.

In an intrinsically integer task, such as CONV, the advantages of using integer types (`int` and `unsigned char`) derive from the faster execution of sums (in IBM and SUN) and the absence of type conversions[9]. On the contrary, the advantages of using floating point types (`float` or `double`) derive from the faster execution of products and from the availability of a higher number of registers. The availability of a higher number of registers is due to the fact that some of the general purpose integer registers cannot be assigned to program variables, as they are needed for housekeeping operations, such as array element address computation, run time stack handling, control variables and others. Using more registers allows the increase of the ELU factor, thus leading to a significant performance improvement: such a performance improvement can be considered an indirect effect of DTO.

The question to be answered, to understand whether migrating to floating point processing is convenient, is whether the slow down due to the introduction of the `unsigned char` to `float` conversion, for the input image, and of the `float` to `unsigned char` conversion, for the convolution result, as well as to the execution of sums in floating point rather than integer, is so high to overcome the acceleration due both to the execution of products in floating point rather than in integer and to the increased ELU factor. In order to analyze these contributions we have run some experiments using programs derived from the scheme of Fig. 4. Table 5 shows the best performance measured for CONV in the reference systems along with the loop unrolling factor and the data types used[10]. TAO and CAO are not applicable in CONV.

4.3.3 Discrete Cosine Transform (DCT)

We investigated DCT as a basic building block of image compression standards, such as JPEG [43] and MPEG [18]. In such a context the following facts have to be considered:

1. the DCT is not applied directly to images but to blocks of 8×8 pixels;
2. the two-dimension DCT of an 8×8 pixel block can be computed by means of a sequence of two one-dimension DCTs, namely a row-wise one-dimension DCT over each row of the input block and a column-wise one-dimension DCT over each column of the result of the row-wise DCT;
3. in each one-dimension DCT of a row of a 8×8 pixel block it is possible to identify a number of partial combinations of the inputs which are repeatedly used to compute the results [30]. The identification of such combinations allows to reduce the number of operations required to carry out the DCT from 56 sums and 64 products, which corresponds to the case of a direct implementation of the DCT definition over eight elements, to 26 sums and 16 products (see Figure 5). The code in Figure 5 corresponds to the body of a loop which must be executed eight times, one for each row of the block, during the first one-dimensional convolution, and one for each column of the block during the second one-dimensional convolution. Two sections can be identified in the program, namely a Section in which the intermediate results to be used repeatedly are computed (statements 1-18), and a Section in which such intermediate results are combined to produce the final result (statements 19-26).

[9] We assume that both the original input image and the output image are declared `unsigned char` arrays. As a consequence, type conversions would be needed to convert them to floating point, in case of floating point processing.

[10] In the case of floating point, the performance measurements include the time required to carry out the input conversion from `unsigned char` to `float` and the output conversion from `float` to `unsigned char`

```
1. for(x=0;x<IM_SIZE;x++)
2. for (y=0; y<IM_SIZE;y++){
3.    temp=0;
4.    for(i=0; i<MASK_SIZE; i++)
5.       for(j=0; j<MASK_SIZE; j++)
6.          temp+= source [x+i][y+j]*mask[i][j];
7.    dest[x+MASK_SIZE/2][y+MASK_SIZE/2]=temp/norm;
8.    }
```

Figure 3 - CONV: code directly derived from task definition

```
1. for(x=0;x<IM_SIZE;x++)
2.    for (y=0; y<IM_SIZE;y+=N){
3.       temp1=temp2=....=tempN=0;
4.       for(i=0; i<MASK_SIZE; i++)
5.          for(j=0; j<MASK_SIZE; j++){
6.          m=mask[i][j];
7.          temp0+= in_image[x+i][y+j]*m;
8.          temp1+= in_image[x+i][y+j+1]*m;
            .....
9.          tempN+= in_image[x+i][y+j+N]*m;
10.            }
11.      out_image[x+MASK_DIM/2][y+MASK_DIM/2]=temp0;
12.      out_image[x+MASK_DIM/2][y+MASK_DIM/2+1]=temp1;
         .....
13.      out_image[x+MASK_DIM/2][y+MASK_DIM/2+N]=tempN;
      }
```

a) ELU

```
1. for(x=0;x<IM_DIM;x++)
2. for (y=0; y<IM_DIM;y++){
3.    out_im[x+MASK_DIM/2][y+MASK_DIM/2]=in_image[x][y]*mask[0][0]+
      .....+in_image[x][y+MASK_DIM]*mask[0][MASK_DIM]...
    .....+in_image[x+MASK_DIM][y+MASK_DIM]*mask[MASK_DIM][MASK_DIM];
5.    }
```

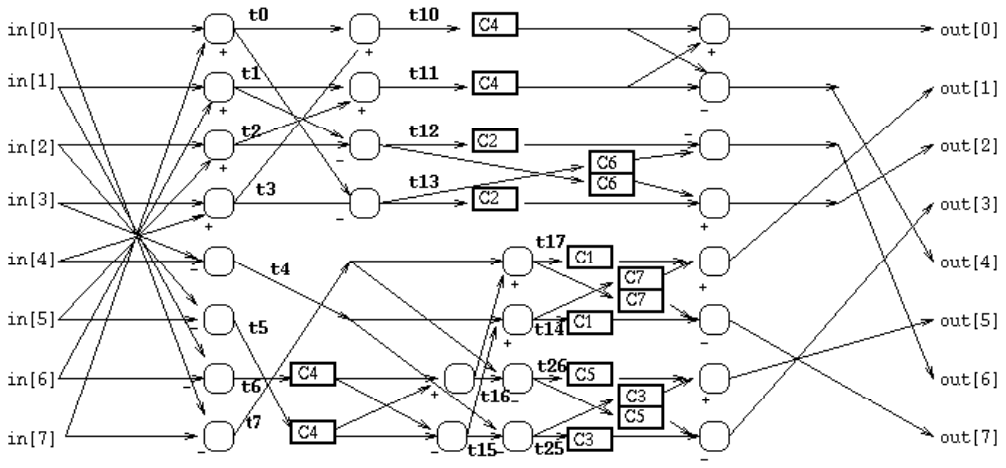
b) ILU

Figure 4 - LU applied to CONV.

	HP	**DEC**	**IBM**	**SGI**	**SUN**
Code derived from task definition:	1,430[g] ms	1,610[c] ms	330[c] ms	888[g] ms	1,360[c] ms
Best performance of optimized code:	230[g] ms	220[c] ms	190[c] ms	225[g] ms	430[g] ms
	LU: ILU 25 DTO: float	LU: ELU 24 DTO: float	LU: ILU 25 DTO: int	LU: ILU 25 DTO: float	LU: ILU 25 DTO: float

Operating conditions: image size: 512×512
mask size: 5×5
compilers: c = native compiler, g = GNU compiler.

Table 5 - Performance of CONV.




```
1.   t0 = in[7] + in[0];
2.   t1 = in[6] + in[1];
3.   t2 = in[5] + in[2];
4.   t3 = in[4] + in[3];
5.   t4 = in[3] - in[4];
6.   t5 = in[2] - in[5];
7.   t6 = in[1] - in[6];
8.   t7 = in[0] - in[7];
9.   t10 = t3 + t0;
10.  t11 = t2 + t1;
11.  t12 = t1 - t2;
12.  t13 = t0 - t3;
13.  t14 = t4 + t15;
14.  t15 = (t6 - t5) * COS_4;
15.  t16 = (t6 + t5) * COS_4;
16.  t17 = t7 + t16;
17.  t25 = t4 - t15;
18.  t26 = t7 - t16;
19.  out[0] = (t10 + t11) *COS_4;
20.  out[2] = t13 * COS_2+ t12 * COS_6;
21.  out[4] = (t10 - t11) * COS_4;
22.  out[6] = t13 * COS_6 - t12 * COS_2;
23.  out[1] = t17 * COS_1 + t14 * COS_7;
24.  out[3] = t26 * COS_3 - t25 * COS_5;
25.  out[5] = t26 * COS_5 + t25 * COS_3;
26.  out[7] = t17 * COS_7 - t14 * COS_1;
```

Figure 5 - DCT: fast data flow diagram and code [30].

	HP	DEC	IBM	SGI	SUN
Best performance of optimized code:	77[g] ms	86[c] ms	45[c] ms	73[g] ms	110[c] ms
	LU: NA DTO: float or double TAO: array or const or variables	LU: NA DTO: float or double TAO: const or variables	LU: NA DTO: int TAO: array or const or variables	LU: NA DTO: int TAO: const or variables	LU: NA DTO: float TAO: array or variables

Operating conditions: image size 512×512
block size 8×8
compilers: c = native compiler, g = GNU compiler.

Table 6 - Performance of DCT.

```
1.  for ( k=0; k<MAX_ANGLES; k++ ){
2.     for ( i=0; i<EDGE_PIX_NUM; i++ )
3.        H[k][OFFS+(int)(edge[i].x*cosine[k]+edge[i].y*sine[k])]++;
4.     }
```

Figure 6 - The HT code derived from its definition (arrays cosine[] and sine[] are assumed to be computed off-line)

```
#define COS_1 c[1]
#define COS_2 c[2]
...
#define COS_7 c[7]
float c[8];
for(i=1;i<8;i++)
   c[i]=cos(PI*i/16);
```

a)

```
#define COS_1 0.980785
#define COS_2 0.923879
....
#define COS_7 0.195090
```

b)

```
register int
COS_1=0.980785,
COS_2=0.923879,
...,
COS_7=0.195090;
```

c)

Figure 7 - TAO techniques: a) array, b) constants, c) variables.

It is important to remark that, due to the symmetry of cosine, only seven cosines, namely *cos (kπ/16), k=1, .., 7*, need to be computed for a 8×8 DCT. As it can be expected, these values are computed off line and repeatedly used in the computation loop.

4.3.3.1 Optimization of DCT

While explicit LU is not necessary, as a sort of ILU is implicit in the code transformation mentioned above under item 3, DTO does affect the performance of DCT. Our experiments have shown that the best performance of DCT in DEC and HP is achieved using the `float` or `double` type. On SUN and SGI the best performance is obtained using the `float` type because the number of available single precision floating point registers (32) turns out to correspond to only 16 double precision floating point registers, which are not enough to hold all the temporary values required. On IBM, finally, the best performance results are obtained using the `int` data type (see Table 6).

A significant performance improvement in DCT can be obtained by moving the computation of the cosines off the main computation loop and keeping the pre-computed values in a fast access Table. We examined three ways in which the cosine Table can be kept in memory, namely using an array, using constants, and using variables. Fig. 7 shows how these three methods of Table access optimization (TAO) are implemented. The performance delivered using an array was shown to depend on the compiler: in particular while the SUN and HP native compilers are able to detect the absence of aliases and assign the array values to registers, the other compilers keep the array values in memory and thus require a load/store operation for each array access. Constants and variables turn out to be almost equivalent and deliver the best performance results. The cosine values are loaded from primary memory to registers the first time they are referenced and are never removed because the number of values necessary to complete the task is smaller than the number of available registers (in all the reference systems but SUN and SGI in double precision). When the number of values used is larger than the number of available registers (i.e., on SUN in double precision) the `register` compiler directive can be issued, when using variables, to suggest the compiler which variables are to be kept in registers and which variables can be removed. As a consequence, using variables is more efficient than using constants. Table 6 presents the best performance measured for DCT in the reference systems along with the data types used and the Table access methods adopted. CAO is not applicable to DCT.

4.3.4 Hough Transform (HT)

Fig. 6 shows the code derived from the Hough Transform definition presented in 3.2.2. Tab. 7 shows its execution time in each reference system, assuming `MAX_ANGLES=180` and `EDGE_PIX_NUM=26214`[11].

4.3.4.1 Optimization of HT

LU, DTO and TAO, definitely improve the performance of the HT code. LU improves the HT performance thanks to the explicit assignment of the CPU registers to the intermediate results of the computation. If R is the number of available registers, the best performance result is achieved by using (R-2) registers to hold the values of sin() and cos() of (R-2)/2 adjacent angles, and using the remaining 2 registers to hold the coordinates of the edge pixel that has to be processed at each iteration. After initialization, only two memory accesses (i.e. the coordinates of the next edge pixel) are necessary at each iteration to perform the computation of (R-2)/2 partial results, which have to be stored in memory[12]. In Fig. 8 is reported, as an example, the case of R=20.

TAO improves the HT performance by reducing the

[11] We consider the case of an image of size 512 x 512 in which 10% of the pixels are edge pixels.

[12] Note that, in general, optimizing compilers automatically map a variables on a CPU register only if such a variable is local, is not an array element, and is never referenced through a & operator.

```
1.  for (k=0;k<MAX_ANGLES; k+=9) {
2.     r_1 =cosine[k  ]; r_2=sine[k  ];
3.     r_3 =cosine[k+1]; r_4=sine[k+1];              /* TAO */
       ....
4.     r_17=cosine[k+8]; r_18=sine[k+8];
5.     for (i=0; i<EDGE_PIX_NUM; i++){
6.        r_19=edge[i].x; r_20=edge[i].y;            /* ELU */
7.        H[k  ][(int)(r_19*r_1 + r_20*r_2 )]++;
8.        H[k+1][(int)(r_19*r_3 + r_20*r_4 )]++;
9.        .....
10.       H[k+8][(int)(r_19*r_17+ r_20*r_18)]++;
11.       }
12    }
```

a)

```
7.        H[k  ][(r_19*r_1 +  r_19*r_2 )>>GRANE]++;        /* DTO */
8.        H[k+1][(r_19*r_3 +  r_19*r_4 )>>GRANE]++;
9.        .....
10.       H[k+8][(r_19*r_17 + r_19*r_18)>>GRANE]++;
11.       }
```

b)

Figure 8 - LU, DTO and TAO applied to HT: a) `all_float` solution and b) `all_int` solution.

```
(int) (((double) int_var)*double_var + ((double) int_var)*double_var)
```

Figure 9 - Type conversions hidden in statement 3 of Fig. 6.

```
1.  while(subset_start != EDGE_PIX_NUM){
2.     subset_start = subset_end; /* Delimit next pixel subset */
3.     subset_end = max ( EDGE_PIX_NUM, (subset_end + SUBSET_SIZE) );
4.     for ( k = 0; k< MAX_ANGLES; k++ ){
5.        c=cosine[k];
6.        s=sine[k];
7.        for ( i=subset_start; i<subset_end; i++ ) /*HT on current subset*/
8.           H[k][OFFS+(int)(edge[i].x*c+edge[i].y*s)]++;
9.     }
10. }
```

Figure 10 - CAO applied to HT.

	HP	**DEC**	**IBM**	**SGI**	**SUN**
Code derived from task definition:	860[c] ms	850[c] ms	2,610[c] ms	960[g] ms	2,340[c] ms
Best performance of optimized code:	340[g] ms	520[g] ms	320[c] ms	460[g] ms	600[g] ms
	LU: ELU 10 DTO: float TAO: variables	LU: ELU 5 DTO: float TAO: variables	LU: ELU 6 DTO: int TAO: variables	LU: ELU 3 DTO: float TAO: variables	LU: ELU 6 DTO: float TAO: variables

Operating conditions:	image size:	512×512
	number of angles:	180
	number of edge pixels:	10% of image pixels
	compilers:	c = native compiler, g = GNU compiler.

Table 7 - Performance of HT.

	HP	DEC	IBM	SGI	SUN
FSBM	2,854 ms	2,205 ms	1,764 ms	1.202 ms	3,586 ms
CONV	230 ms	220 ms	190 ms	225 ms	430 ms
DCT	77 ms	86 ms	45 ms	73 ms	110 ms
HT	340 ms	520 ms	320 ms	460 ms	600 ms

Operating conditions:
image size: 512×512 one byte pixels.
CONV mask size: 5×5
FSBM displacement: 8
DCT block size: 8×8
HT number of edge pixels: 10% of image pixels

Table 8 - Summary of the best performances.

time required to access the tables containing the pre-computed values of sin() and cos(). Before entering the program internal loop the array values to be used within the loop body are loaded into variables (see Figure 8 statements 2-4).

DTO improves the HT performance thanks to the elimination of some type conversions (see Fig. 9) and to the use of a larger number of registers. Two alternative DTO solutions, named all_`float` and all_`int`, are shown in the optimized code of Fig. 8. The all_`int` solution performs all the computations in fixed point and thus requires fixed point tabulation of sin() and cos(), while the all_`float` solution performs all the computations in floating point and thus requires a preliminary conversion of the edge point coordinates from integer to floating point before starting the HT computation. The most evident advantage of the all_`int` solution over the all_`float` solution is the elimination of the final `float-to-int` conversion (see Fig. 9), which is quite expensive, considering the absence of a direct path from the floating point registers to the integer registers (see Table 1). On the contrary the most evident advantages of the all_`float` solutions over the all_`int` solution are the execution of the multiplications in floating point (which are faster than in integer) and the elimination of the right shift implementing fixed point data alignment (see statements in Fig. 8(b)).

In addition to LU, DTO and TAO, the structure of the HT code suggests to explore cache access optimization techniques (CAO) to improve efficiency. In particular, it would seem possible to plan the sequences of memory accesses to maximize locality: the set of edge pixels that have to be scanned repeatedly can be partitioned into subsets of size equal to that of cache memory, which are then processed one at a time. The resulting flow is presented in Figure 10. After initial loading, subset processing consists of looping over the edge pixels inside the cache with no access to primary memory. Unfortunately, this optimization is effective only under particular operating conditions (i.e., cache size, image size): in most cases, on the contrary, a set of negative effects (e.g., related to matrix H access) compensate the benefit of increased edge pixel access locality. In particular, in none of the reference systems the cache size and the image size allow taking advantage of CAO[13]. Table 7 presents the best performance measured for HT in the reference systems along with the unrolling factor, the data types used and the Table access method adopted.

4.4 SUMMARY OF THE RESULTS OF THE EXPERIMENTS

Table 8 summarizes the best performances of the basic IPPR tasks on the reference systems. A rough estimate of the level of performance that RISC systems can achieve in IPPR applications can be based on the observation that, under the operating conditions selected, the chain CONV-HT, which might be regarded as a straight line recognition task, typical of pattern recognition, exhibits a throughput of 0.6 to 2 recognitions per second, whereas the chain FSBM-DCT, which might be regarded as a video compression task, typical of distributed multimedia, exhibits a throughput of 0.2 to 0.3 frames per second, depending on the reference system[14].

Table 9 shows the speed up factor due to the separate application of each source level optimization to the code derived from the task definition for each IPPR basic task.

[13] On the contrary, we experimented the benefit of CAO in two other platforms, namely HP 725 (128Kbytes cache, direct mapped) and Alpha 600 (96Kbytes cache three-way set associative).

[14] Using suboptimal techniques such as Logarithmic Search [18], the speed of block matching improves of a factor between 5 and 10.

	LU	DTO	TAO	CAO	Global
FSBM	2.1 ÷ 3.9	NA	NA	NA	2.1 ÷ 3.9
CONV	1.7 ÷ 5.0	1 ÷ 1.7	NA	NA	1.7 ÷ 6.2
DCT	1	1 ÷ 1.7	1 ÷ 1.4	NA	1 ÷ 2.4
HT	1.4 ÷3.0	1.2 ÷ 2.5	1 ÷ 1.2	1	1.6 ÷ 8.2

Table 9 - Ranges of the speed up factors due to the optimizations.

T_{IC}	T_{CC}	T_{MA}	T_{OH}	T_{OL}
103 ms	37 ms	$T_{CACHE;MA}$: 54 ms $T_{PRIMARY;MA}$: 15 ms	24 ms	28 ms

Table 10 - CONV on HP: times spent in the different activities

The last column reports the speed up factors due to the best combination of the source level optimizations. A "NA" sign denotes that an optimization technique is not applicable to a task, while a "1" denotes the fact that an optimization technique has no effect on a task. It is worth noticing that:

1) when no source level optimization is adopted, RISC systems run at about one third of the speed that they are actually able to achieve;
2) loop unrolling (LU) is by far the most effective optimization technique. It has no effect on DCT because a sort of internal loop unrolling (ILU) is implicit in the adopted algorithm;
3) data type optimization delivers no benefit on IBM for CONV and on HP and ALPHA for DCT because in these architectures the data types naturally deriving from the task definition (respectively `int` and `float`) already deliver the best performance;
4) table access optimization (TAO) is beneficial only when the compiler does not optimally allocate table elements in registers;
5) cache access optimization (CAO) has no effect on the basic tasks in the reference systems under the operating conditions selected. In FSBM, CONV and DCT CAO is not applicable due to the intrinsically local nature of the algorithms[15], which matches the intrinsically local nature of caching. On the contrary in HT, which is not a local task, CAO delivers no benefit as it turns out to be in conflict with LU, which, as mentioned above, is more convenient.

[15] Although DCT is in principle not local, when used in image compression standards, it can be considered local because of the small size of the data sets.

5 DISCUSSION

The discussion of the results of our study and of our experiments is organized as follows. In Section 5.1 we analyze the behaviour of the IPPR basic tasks on the RISC reference systems in order to identify the architecture bottlenecks. In Section 5.2 we discuss the limitations of the current generation programming models, which force programmers to adopt source level program optimization techniques to obtain an acceptable level of efficiency.

5.1 Bottleneck identification

The execution time of an IPPR task, on a RISC machine, can be computed by means of the following expression:

$$T_{EX} = T_{IC} + T_{CC} + T_{MA} + T_{OH} - T_{OL} \quad (1)$$

in which, T_{IC} (Image Computation) denotes the time spent in executing operations on image pixels, T_{CC} (Control Computation) denotes the time spent in executing operations on loop control variables, array addresses and, more in general, for housekeeping, T_{MA} (Memory Access) denotes the time for memory access, T_{OH} (Overhead) denotes the time wasted due to inefficiencies, and T_{OL} (Overlap) denotes the time saved due to superscalar execution of operations and instructions.

The terms in expression (1) are affected by the optimizations presented in Section 4 as follows. T_{IC} is affected by DTO, which adapts the algorithm data types to those supported most efficiently by the CPU. T_{CC} is affected by TAO, which reduces the number of computations needed to compute the array addresses, and

by LU, which reduces the amount of loop control computations, due to the lower number of iterations, as well as the amount of array address computations, due to the allocation of a larger number of variables in CPU registers. T_{MA} is affected by LU and TAO, both of which improve the use of CPU registers. T_{OH} is affected by LU, which enlarges the size of the loops and thus allows to take advantage of more sophisticated instruction scheduling techniques. T_{OL} is affected by DTO, which may lead to the use of different functional units for T_{IC} and T_{CC} (e.g., floating point and integer), and by LU, which increases the number of parallel activities inside the program most internal loop.

A quantitative analysis of the terms of expression 1 was carried out for the specific case of CONV on HP to identify the main processing bottle-necks (see Table 10). T_{IC} was obtained by dividing the number of image computations needed to perform CONV (multiplies and add) by the peak machine throughput for these computations (see Table 1). T_{MA}, T_{CC} and T_{OL} were obtained respectively by counting the memory accesses (evaluating both $T_{CACHE;MA}$ and $T_{PRIMARY;MA}$) and the housekeeping instructions and by considering the overlapping among the different terms in the CONV assembly code corresponding to the best performance obtained. These values were validated by experiments. The value of T_{OH} was obtained by $T_{OH} = T_{EX} - T_{IC} - T_{CC} - T_{MA} + T_{OL}$.

The experiments have shown that T_{OL} is almost completely due to the overlapping between T_{IC} and T_{MA}, in particular $T_{CACHE;MA}$, while only a very limited overlapping was observed between T_{CC} and T_{IC} and between T_{CC} and T_{MA}.This is due to a combination of two reasons, namely the intrinsic data dependencies appearing in the CONV loop body, which require that some of the machine instructions be executed in sequence, and the limited number of functional units available (structural hazards), which do not allow the parallel execution of all the potentially parallel activities.

We modified the order of instructions manually to further improve the performance of the assembly code produced by the compiler/optimizer. In the best case we achieved an almost complete overlap between cache memory accesses and image computations. This leads to the conclusion that, unlike expected, CONV on HP is a compute bound task, as memory accesses can be totally executed in parallel with computation. We observed the same behaviour, qualitatively, in all the basic tasks and in all the reference systems. As a consequence, CPU speed-ups due, for example, to an increase of instruction level parallelism in the next generation CPUs, are expected to yield task speed-ups of the same factor, provided that $T_{IC} \geq T_{CACHE;MA}$.

As far as primary memory access is concerned, no overlapping was observed and, as a consequence, a term of 15 ms, mainly due to the scanning of the input image, is to be added to the other terms to compute the global task execution time. The reason why no overlap can take place between primary memory accesses and other activities is that primary memory accesses are activated only at the occurrence of cache misses, that is exactly at the moment at which the missing data items are needed and therefore the CPU remains idle waiting for memory access completion. Prefetching techniques were proposed to reduce such a latency [33]. In IPPR, where memory accesses are known in advance, it would be possible to plan the loading of cache lines in advance to eliminate the latency of primary memory access, thus overlapping primary memory accesses with other system activities. This behaviour should be supported by proper architectural features, such as for example independent access to cache memory from the CPU side and from the primary memory side, as well as by special language directives.

5.2 Programming model limitations

Although it is true that present generation compilers use sophisticated techniques to optimize the use of hardware resources, to reduce the number of data dependencies, and more in general to schedule machine instructions in order to maximize efficiency, unfortunately in data intensive programs such as those belonging to the IPPR domain, the action of compilers is not sufficient to generate efficient code and must be complemented by a careful organization of the source code.

As a consequence, writing an efficient program implementing an IPPR task on a RISC based architecture requires to take into account the characteristics of the specific RISC architecture for which the program is written and a careful analysis of the type of processing performed by the IPPR task, in order to identify the most convenient coding solutions. If, on the contrary, the program is coded as a plain transposition of the task definition, leaving the task of finding a convenient mapping on the RISC architecture to the compiler, then efficiency turns out to be unacceptable[16] (see Section 4.4).

The reason why manual source code optimization is necessary is that in compilers code generation and optimization take place after the analysis of the source program and, as a consequence, after the generation of the compiler internal representation. At that point several of the most relevant task characteristics, in particular some which might be used to improve the efficiency of the object code, are lost and cannot be exploited by the optimizer. In order to take advantage of the task characteristics, or at least of part of them, the optimizer should process the compiler internal representation to infer as much information as possible on the task structure, which was clear in the mind of the programmer, not so evident any more in the source program, and almost completely hidden in the program internal representation inside the compiler.

The point is that unfortunately while current generation CPUs have moved apart from the flat execution model based on the fundamental Von Neumann abstraction

[16] This is usually the case also in numerical analysis applications, which share with IPPR the characteristic of being based on the processing of large data sets, most of the times organized as two-dimension arrays [9][21].

(which does not include, for example, memory hierarchy and data dependencies), on the contrary the programming models upon which current generation high level languages (e.g., C, Pascal and their descendants) are based have not evolved along the same direction. While it can be argued that, generally speaking, programmers should not be aware of the specific characteristics of the architecture of the system in which programs are supposed to run, which implies that efficiency is not to be regarded as a goal of programming models, there are some domains, and IPPR is among them, in which, on the contrary, efficiency is a primary goal and in which, as a consequence, programmers cannot restrict their action to the development of high level, correct and portable code. In these domains the flatness of the available programming model forces programmers to optimize the code manually.

6 CONCLUDING REMARKS

In this paper we presented the results of a study on the performance and on the behaviour of RISC systems in image processing and pattern recognition (IPPR). The investigation was conducted experimentally as follows. We selected a set of high performance RISC based systems, from those presently commercially available, to represent RISC technology, and identified a set of IPPR tasks to represent the image processing domain. We studied the behaviour of the RISC systems selected in the execution of the IPPR tasks identified by running a set of experiments based on different program organizations.

The main contributions of the paper are i) the performance evaluation of current generation RISC systems in the domain of IPPR tasks, ii) the analysis of the behaviour of RISC systems in the execution of IPPR tasks and iii) the identification of a set of source level program optimizations delivering a significant speed up over regular program implementation in the domain of IPPR.

BIBLIOGRAPHY

[1] Amano T. et al., DRS: a Workstation Based Document Recognition System for Text Entry, IEEE Computer, vol. 25, n. 7, pp. 67-71, July 1992.

[2] Asprey T., Averill G. S., DeLano E., Mason R., Weiner B. and Yetter J., Performance Features of the PA7100 Microprocessor, IEEE Micro, pp. 22-35, June 1993.

[3] Ballard D. H. and Brown C. M., Computer Vision, Prentice Hall, 1982.

[4] Bertero M., Poggio T. A. and V. Torre, Ill-posed problems in Early Vision, Proceedings of the IEEE, vol. 76, n. 8, pp. 869-889, 1988.

[5] Digital Equipment Corp., Alpha Workstation Summary, September 1995.

[6] Digital Equipment Corp., DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors - Hardware Reference Manual, 1994.

[7] Dixit D., The SPEC Benchmarks, Parallel Computing, vol. 17, n. 1, pp. 195-209, 1991.

[8] Dongarra J. J. and Gentzcsh, eds., Computer Benchmarks, North-Holland, Amsterdam, 1993.

[9] Dowd K., High Performance Computing, O'Reilly Associates Inc., 1993.

[10] Fountain T. J., Matthew K. N. and Duff M. J. B., The CLIP7A image Processor, IEEE Transaction on Pattern Analysis and Machine Intelligence, vol. 10, no. 3, pp. 310-319, 1988.

[11] Greenley D. et al., UltraSPARC: The next Generation Superscalar 64-bit SPARC, CompCon Spring 95, March 1995.

[12] Heinrich J., MIPS R4000 Microprocessor Users's Manual, MIPS Technologies Inc., 1994.

[13] Hennessy J. L. and Patterson D. A., Computer Architecture: a Quantitative Approach, Morgan-Kauffman, 1990.

[14] Hewlett Packard, HP 9000 series 700 models 735/125, September 1995.

[15] Hewlett Packard, PA-RISC 1.1 Architecture and Instruction Set Reference Manual, February 1994.

[16] Hillis W. D., The Connection Machine, The MIT Press, 1985

[17] IBM Corp., RISC System/6000 Overview, September 1995.

[18] ISO/IEC JTC1/SC29/WG11, Coding of Moving Pictures and Associated audio for Digital Storage Media at up to about 1.5 Mbits/s, ISO/IEC CD11172-2, 1993.

[19] Jain R., The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling, Jonh Wiley & Sons, New York, 1991.

[20] JàJà J., Introduction to Parallel Algorithms, Addison-Wesley, 1992.

[21] Kacmarcik G., Optimizing PowerPC Code, Addison-Wesley, 1995.

[22] Lee B. G. L., A new algorithm to Compute the Discrete Cosine Transform, IEEE Trans. on Acoustic, Speech, Signal Processing, vol. ASSP-32, n. 6, pp. 1243-1245, December 1984.

[23] Legall D., MPEG - A video Compression Standard for Multimedia Applications, Communications of the ACM, vol. 34, no. 4, pp. 47-58, April 1991.

[24] Maresca M. and Fountain T. J., eds., Special Issue on Massively Parallel Computers, Proceedings of the IEEE, 1991.

[25] Maresca M. and Li H., Morphological Operations on Mesh Connected Architectures: a generalized convolution algorithm, Proc. IEEE Conference on Computer Vision and Pattern Recognition, Miami Beach (FL), pp. 199-304, June 1986.

[26] McLellan E., The Alpha AXP Architecture and 21064 Processor, IEEE Micro, pp. 36-47, June 1993.

[27] Potter J. L., ed., The Massively Parallel Processor, The MIT Press, 1980.

[28] Pratt W. K., Digital Image Processing, John Wiley & Sons, 1991.

[29] Puri A. and Aravind R., Motion-Compensated Video Coding, IEEE Trans. on Circuits and Systems for Video Technology, vol. 1, n. 4, pp. 351-361, December 1991.

[30] Rao K. R. and YipP., Discrete Cosine Transform - Algorithms, Advantages, Applications, Academic Press, London, 1990.

[31] Rosenfeld A. and Kak A. C., Digital Picture Processing, Academic Press, 1982.

[32] Saavedra R. H. and. Smith A. J, Performance Charcterization of Optimizing Compilers, IEEE Transactions on Software Engineering, vol.21, no. 7, pp. 615-628, July 1995.

[33] Saavedra R. H. and. Smith A. J, Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes, IEEE Transactions on Computers, vol.44, no. 10, pp. 1223-1235, Oct. 1995.

[34] Silicon Graphics Inc., Product Overview, September 1995.

[35] Silicon Graphics Inc., Indigo2 and Power Indigo Technical Report, 1994.

[36] Song S. P., Denman M. and Chang J., The PowerPC 604 RISC Microprocessor, IEEE Micro pp. 8-17, October 1994.

[37] SPARC International Inc., The SPARC Architecture Manual Version 8, - Prentice Hall, 1992.

[38] Stallman R. M., Using and Porting GNU CC, GNU Software Foundation, September 1994.

[39] SUN Microsystems, The SuperSPARC Microprocessor - White Paper, 1992.

[40] SUN Microsystems, Workstation Overview, September 1995.

[41] Tremblay M., P. Tirumalai, Partners in Platform Design, IEEE Spectrum, vol.32, no. 4, pp. 20-26, April 1995.

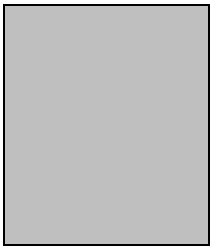
[42] Uhr L., ed., Multicomputer Vision, Academic Press, 1988.

[43] Wallace G. K., The JPEG Still Picture Compression Standard, Communication of the ACM, vol 34, n. 4, pp. 30-44, April 1991.

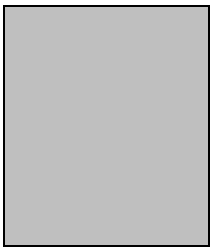
[44] Weems C. C., Architectural Requirements of Image Understanding with respect to parallel processing, Proceedings of the IEEE, vol. 79, no. 4, pp. 537-547, April 1991.

[45] Weems C. C., Riseman E., Hanson A. and Rosenfeld A., The DARPA Image Understanding Benchmark for Parallel Computers, J.

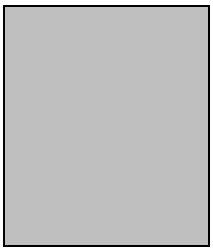
of Parallel and Distributed Computing, vol. 11, pp. 1-24, 1991.
[46] White S. W., Hester P. D., Kemp J. W. and McWilliams G. J., How Does Processor Performance MHz Relate to End-User Performance?, IEEE Micro, vol. 13, n. 4, pp. 8-16, August 1993.



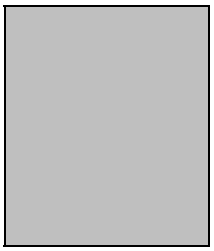
Pierpaolo Baglietto was born in Varazze, Italy, in 1963. He received a Laurea Degree in electrical engineering in 1990 from the University of Genoa, Italy and a Ph.D. in computer engineering in 1994 from the same University. He was a Visiting Scientist at the MasPar Computers Corp., Sunnyvale, CA, in 1991 and at the NTT Communication Science Laboratories, Kyoto, Japan in 1992. He is currently a Researcher at the University of Genoa, Italy. His research interests include computer architecture and performance evaluation, operating systems and computer networks.



Massimo Maresca was born in Genoa, Italy, in 1956. He received a Laurea Degree in electrical engineering in 1980 and a Ph.D. in computer engineering in 1986, all from the University of Genoa, Italy. He was a Research Staff Member at Elsag SpA, Genova, from 1980 to 1982, a Visiting Scientist at the IBM T.J. Watson Research Center, Yorktown Heights, NY, in 1985-1986, a Visiting Scientist at the International Computer Science Institute, Berkeley, CA, in 1990-91, a Researcher at the University of Genoa in 1990-92, and an Associate Professor at the University of Genoa in 1993. Currently he is a Full Professor at the University of Padova, Italy. His research interests are in the area of Computer architecture, operating systems and networking.



Mauro Migliardi is born in Genova (Italy) on April the 19th 1966. In June 1991 he took a Laurea degree in electrical engineering from University of Genoa. In November 1995 he took a PhD in computer engineering from the University of Genova. Currently he is a Post Doctoral Fellow at the University of Genova where he collaborates with Prof. M. Maresca in EU funded research projects dedicated to the investigation of mixed mode (SIMD-MIMD) image processing. His main research interests are computing architectures for image and signal processing and coding, architectural support for multimedia systems and parallel and distributed high performance computers.



Nicola Zingirian received the Laurea degree in electrical engineering summa cum laude in 1994 from the Universtity of Genoa, Italy. He is currently a PhD student in computer engineering at the University of Padova, Italy. His research interests include performance evaluation, computer architectures and networking. He is associate member of the IEEE Computer society.