Modeling Instruction Level Parallel Architectures Efficiency in Image Processing Applications♣

M. Migliardi, DIST University of Genova, Via Opera Pia 13, 16145, Genova, Italy
M. Maresca, DEI University of Padova, Via Gradenigo 6/A, 35131, Padova, Italy

Abstract. *Image Processing and Pattern Recognition (IPPR) is receiving new impulse from the progress of Instruction Level Parallel (ILP) architectures which in general exhibit a level of performance comparable with that of the previous decade supercomputers. However, in spite of the huge computing power in principle available, it is a common experience that ILP efficiency in IPPR turns out to be low.*
In this paper we describe the sources of inefficiency of ILP in IPPR and define a set of indices that allows analyzing them quantitatively. The quantitative analysis of the sources of inefficiency can be used by applications software developers to identify the most convenient coding solutions for IPPR algorithms (e.g. loop unrolling, loop permutation, register assignment) as well as to assess the advantages of such solutions over the natural and straightforward transposition of the algorithms in programs.
Keywords: *Performance Indices, Instruction Level Parallel Architectures, Coding Solutions, Image Processing Applications.*

1 Introduction

Image Processing and Pattern Recognition (IPPR) have received a new impulse from the progress of Instruction Level Parallel (ILP) architectures and in particular from the advances of RISC technology [5][10][11]. As a consequence of the fact that low cost RISC systems presently deliver a computing power of the same order of magnitude as that delivered by supercomputers a few years ago, many applicative fields that in the past were not even targeted because of the prohibitive cost of the hardware required, and in particular IPPR, have now become convenient for software based solutions.
Unfortunately, it is a common experience that software implementations of IPPR tasks on ILP architectures exhibit a level of performance by far lower than expected, up to one order of magnitude lower than the peak performance [2]. The inefficiency of ILP RISC architectures in IPPR derives both from the structure of the CPU and from the hierarchical structure of the memory system [9]. In this paper we focus our attention on the first category of inefficiencies.

♣ This work was supported by European Community through the ESPRIT BRA Project 8849-SM-IMP and by MURST.

We model the inefficiency of IPPR applications due to the structure of ILP CPU as the ratio between the ideal (calculated) performance and the real (measured) performance. We define efficiency as the inverse of inefficiency; thus an application has an efficiency value ranging from 1 (ideal efficiency) to 0. In our model efficiency is the product of three factors, namely a factor related to the number of unnecessary load/store instructions, a factor related to the number of unnecessary non load/store instructions and a factor related to the utilization of the CPU functional units.
In order to maximize the efficiency of a task a programmer can use source level coding techniques [2][4] as well as optimizing compilers; unfortunately source level coding techniques and optimizing compilers very often interact and produce unexpected negative effects. On the contrary, the capability of measuring the three factors the product of which gives the global efficiency factor allows both to identify the most appropriate source level coding solutions or transformations[1] in advance and to measure the results of their application.
The paper is structured as follows: in section 2 we define the set of indices that allows to express the efficiency of ILP CPU in IPPR; in section 3 we compute the values of the indices in a case study, namely two-dimensional convolution, and use them to identify the most convenient way to transform the source code in order to improve efficiency; in section 4 we compute the values of the indices on the code transformed in order to evaluate the effectiveness of the source level program transformation identified in section 3; in section 5 we discuss the results of the experiments and provide some concluding remarks.

2 A Quantitative Model for ILP Architectures Efficiency

Current generation high end workstations adopt many different architectural solutions to achieve high performance. Nevertheless some common traits can be found, namely a large degree of Instruction Level Parallelism through deep pipeline structures and wide superscalar structures, a reduced instruction set, and a Load-Store architecture.
Considering these common traits it is possible to identify two main categories of inefficiencies:

I. inefficiencies due to the addition of unnecessary machine instructions;
II. inefficiencies due to the waste of machine cycles in the execution of the machine instructions.

Thus we can say that the efficiency of an ILP architecture can be divided in *Instruction Efficiency* (I) and *Execution Efficiency* (II).
A more in depth analysis allows us to identify two different sources of inefficiency falling under the definition of *Instruction Efficiency* of a task, namely:

A. the number of unnecessary load/store instructions due to the limited number of CPU registers;
B. the number of bookkeeping instructions, i.e. instructions that are not inherently due to the task, but that are necessary to control the correct execution of the task (e.g. control flow, addresses calculations, etc.).

[1] We define a source level code transformation a rewriting of the source level code implementing a same algorithm in a different way.

Thus we can say that the *Instruction Efficiency* of a task can be further on divided in *Load/Store Efficiency* (A) and *Bookkeeping Efficiency* (B).
The main source of inefficiency that lowers the *Execution Efficiency* of a task is the idleness of the CPU functional units which is normally due to hazards. These inconveniences prevent ILP architectures from reaching their maximum instruction execution throughput.

2.1 Instruction Efficiency

A program is a sequence of machine instructions. Each machine instruction belongs to one of the three following categories:

- load/store operation instructions; these instructions may be due to the need to load input data into registers and to store output data to primary memory, or they may be due to the spilling of temporary values;
- arithmetical/logical operation instructions; these instructions may be due to the operations intrinsically necessary to carry out the task or they may be due to bookkeeping;
- branch operation instructions; these may be due to ramification intrinsic to the flow of the task or they may be due to programming techniques such as loops.

In this paper we consider the latter two categories as contributing together to lower the Bookkeeping Efficiency of a task. Thus we distinguish only between Load/Store Operations (LSO) instructions and Arithmetical/Logical and Branch Operations (ALBO) instructions.

Definition 1.
We define the minimum number of LSO instructions of a task (L_{min}) as the number of LSO instructions that a load-store architecture with an infinite number of register would execute to perform the same task.

L_{min} is easily calculated: it is equal to the number of input data-items of the task plus the number or output data-item of the task. In fact every input data item needs to be loaded just once at the beginning of the task as register pool cannot be exhausted (infinite register pool), intermediate results never need to be stored in primary memory for the same reason, and every output data-item needs to be stored just once at the end of the task.

Definition 2.
We define the actual number of LSO instructions of a task (L_{ACT}) as the sum of all LSO instructions of the code actually executed.

L_{ACT} can be measured by inspecting the machine code of the task and counting the LSO instructions actually executed.

Definition 3.
We define the number of additional LSO instructions of a task (L_{add}) as the actual number of LSO instructions of a task (L_{ACT}) minus the minimum number of LSO instructions (L_{min}) of that same task.

Thus, for a given task:

$$\mathbf{L_{ACT} = L_{min} + L_{add}} \quad \textbf{(1)}$$

L_{add} takes into accounts the LSO instructions due to register spilling.

Definition 4.
We define the minimum number of processing instructions (P_{min}) of a task as the minimum number of ALBO instructions that must be executed on the data-items to complete the task.

P_{min} corresponds to the number of ALBO instructions required by the algorithm[2]. That number can be calculated by counting the ALBO instructions that a human operator would need to perform to execute the task manually[3].

Definition 5.
We define the number of actual processing instructions of a Task (P_{ACT}) as the number of ALBO instructions of a task actually executed.

P_{ACT} can be measured by inspecting the machine code of the task and counting all the ALBO instructions actually executed.

Definition 6.
We define the number of additional processing instructions of a task (P_{add}) as the actual number of processing instructions of a task (P_{ACT}) minus the minimum number of processing instructions (P_{min}) of the same task.

Thus for a given task:

$$\mathbf{P_{ACT} = P_{min} + P_{add}} \quad \textbf{(2)}$$

P_{add} takes into account the ALBO instructions due to bookkeeping operations and programming artifacts.

Definition 7.
We define the actual number of instructions of a task (I_{ACT}) as the sum of the actual number of load and store instructions (L_{ACT}) plus the actual number of processing instructions (P_{ACT}).

Thus for a given task:

$$\mathbf{I_{ACT} = P_{min} + P_{add} + L_{min} + L_{add}} \quad \textbf{(3)}$$

Definition 8.
We define the minimum number of instructions of a task (I_{min}) as the sum of the minimum number of LSO instructions of the task (L_{min}) plus the minimum number of processing instructions of the task (P_{min})

A task with no inefficiency due to unnecessary instructions would need to execute exactly I_{min} machine instructions.

---

[2] This does not include any branch due to programming artifacts such as loops.

[3] Let's clarify this definition with an example. To compute the bidimensional convolution of a 128x128 image with 3x3 mask we need to perform 128x128x3x3 multiplications and sums, thus P_{min} for this convolution task is equal to 128*128*3*3 = 147456.

Definition 9.
*We define the Load Efficiency Index (**L**) of a task as*

$$\mathrm{L} = 1 - \frac{L_{add}}{I_{ACT}} \qquad \textbf{(4)}$$

From a qualitative point of view ***L*** measures how far a piece of code is from the ideal result of a single load for each data item and a single store for each result. ***L*** is equal to 1 in the case of an architecture with an infinite register set and decreases toward zero as the number of register spilled during the execution grows up (register trashing).
In real programs it is possible, though difficult [2], to reach high values of ***L***. Tasks with a very high level of data locality can approach such a value.

Definition 10.
We define the reduced actual number of instructions of a task (I_{ACTR}) as the number of actual instructions of the task (I_{ACT}) minus the number of additional Load and Stores of the task (L_{add}).

$$I_{ACTR} = I_{ACT} - L_{add} \qquad \textbf{(5)}$$

The subtraction of L_{add} from I_{ACT} eliminates the contribution of the additional loads from the measurements of *Bookkeeping Efficiency*.

Definition 11.
*We define the Bookkeeping Efficiency Index (**P**) of a task as*

$$P = 1 - \frac{P_{add}}{I_{ACTR}} \qquad \textbf{(6)}$$

P measures how many of the ALBO instructions of the task are intrinsically due to the task execution and how many of the ALBO instructions are due to Bookkeeping. ***P*** is equal to 1 when all the ALBO instructions are intrinsically due to task computations and decreases toward 0 as the number of ALBO instructions due to bookkeeping becomes relevant with respect to the ALBO instructions due to processing.
So far we have identified two sources of inefficiency, namely register spilling and bookkeeping, and have measured them by means of the indices ***L*** and ***P***. Considering that both ***L*** and ***P*** measure a number of unnecessary instructions added to the task, we can say that they capture different aspects of *Instruction Efficiency*.

Definition 12.
*We define the Instruction Efficiency Index (**I**) of a task as the ratio*

$$I = \frac{I_{\min}}{I_{ACT}} \qquad \textbf{(7)}$$

I measures the *Instruction Efficiency* of the task, that is how far from the ideal result the task is in terms of number of machine instructions to be executed. This index takes into account both *Load Efficiency* and *Bookkeeping Efficiency*.

Proposition
The effects of *Load Efficiency* and *Bookkeeping Efficiency* contribute orthogonally to I. As a consequence

$$I = PL \qquad \textbf{(8)}$$

Proof
from **(5)** and **(6)**

$$P \bullet L = \frac{I_{ACT} - L_{add}}{I_{ACT}} \bullet \frac{I_{ACTR} - P_{add}}{I_{ACTR}}$$

that gives, from the definition of I_{ACTR} **(5)**

$$P \bullet L = \frac{I_{ACT} - L_{add}}{I_{ACT}} \bullet \frac{I_{ACT} - L_{add} - P_{add}}{I_{ACT} - L_{add}}$$

that gives

$$P \bullet L = \frac{I_{\min}}{I_{ACT}}$$

that gives, from the definition of I **(7)**

$PL = I$ **QED**

Proposition **(8)** allows us to visualize *Instruction Efficiency* as a plane in which the X axis corresponds to *Load Efficiency* and the Y axis corresponds to *Bookkeeping Efficiency*. In the *Instruction Efficiency Plane* the level of *Instruction Efficiency* of a task is represented by the area of the rectangle of length L and width P. We call the rectangle that represents a task on the *Instruction Efficiency Plane* the *Instruction Efficiency Rectangle* of that task. A task with an ideal level of *Instruction Efficiency* (I = 1) is represented as a unitary side square. We call this unitary side square the ideal square. As the unitary value is the upper bound for a task, every task is represented by a rectangle contained in the *Ideal Square*.
The difference between the area of the ideal square and the area of the *Instruction Efficiency Rectangle* of a task represents the level of *Instruction Inefficiency* of the task.

2.2 Execution Efficiency

In this section we analyze the inefficiency of ILP architectures with regard to the number of wasted execution cycles.
We call the Maximum Number of Instructions issued (MaxIssue) the number of instructions that the CPU issues in a single clock cycle in the best case.

Definition 13.
We define the Minimum Execution Time (T_{min}) of a task as the ratio

$$T_{\min} = \frac{I_{\min}}{MaxIssue} Tcycle \quad (9)$$

T_{min} is the ideal execution time of a task both from the *Instruction Efficiency* point of view and from the *Execution Efficiency* point of view. In fact it implies the optimal instruction number and the optimal usage of CPU resources. For every task T_{min} represents a lower bound for the execution time.

Definition 14.
We define the Minimum Actual Execution Time (T^{ACT}_{min}) of a task as the ratio

$$T^{ACT}_{\min} = \frac{I_{ACT}}{MaxIssue} Tcycle \quad (10)$$

T^{ACT}_{min} represents the ideal execution time of a task given the actual instruction sequence. Thus it implies an ideal *Execution Efficiency* but only the actual level of *Instruction Efficiency* of the task. The goal of achieving an execution time equal to T^{ACT}_{min} is seldom reached as it implies both a perfect instruction scheduling (no hazards), and a perfect instruction mix (MaxIssue instructions issued at each clock cycle).

Definition 15.
We define the Execution Efficiency Index (E) of a task as the ratio

$$E = \frac{T^{ACT}_{\min}}{T^{ACT}} \quad (11)$$

where T^{ACT} is the measured (actual) execution time of the task.

E measures how efficient the instruction execution of a task is, in fact it measures the ratio between the ideal execution time and the actual execution time of the actual instruction sequence of the task. This index includes the effects deriving both from pipeline stages idleness and from an instruction sequence not able to feed the functional units[4].

[4] Consider as an example a CPU with an integer unit and a FP unit and an instruction sequence composed solely of integer instructions. This instruction sequence can be able to keep all the pipeline stages busy all the time, nevertheless it is unable to feed the FP functional unit that is left idle cycle after cycle. Thus the instruction sequence cannot have a high value of E.

Definition 16.
*We define the Global Efficiency Index (**G**) of a task as the ratio*

$$G = \frac{T_{\min}}{T^{ACT}} \quad \textbf{(12)}$$

where T^{ACT} is the measured (actual) execution time of the task.

G measures the level of efficiency of the task both from the *Instruction Efficiency* point of view and from the *Execution Efficiency* point of view, i.e. it gives a comprehensive measure of all the sources of efficiency of a ILP architectures. Thus ***G***. gives a global measure of the effects previously individually captured by ***E***, ***L*** and ***P***.

Proposition

The three components of ***G***. namely ***E***, ***L*** and ***P***, measure orthogonal sources of inefficiency. As a consequence

$$G = E \bullet L \bullet P \quad \textbf{(13)}$$

Proof

Applying the definition of ***E*** **(11)** to **(13)** we obtain

$$E \bullet L \bullet P = \frac{T_{\min}^{ACT}}{T^{ACT}} \bullet L \bullet P \quad \textbf{(14)}$$

Applying the definition of $T^{ACT}_{\min}$ **(10)** to **(14)** we obtain

$$E \bullet L \bullet P = \frac{I_{ACT}}{MaxIssue} \bullet Tcycle \bullet \frac{1}{T^{ACT}} \bullet L \bullet P \quad \textbf{(15)}$$

Applying proposition **(8)** to **(15)** we obtain

$$E \bullet L \bullet P = I \bullet \frac{1}{L} \bullet \frac{1}{P} \bullet \frac{I_{ACT}}{MaxIssue} \bullet Tcycle \bullet \frac{1}{T^{ACT}} \bullet L \bullet P \quad \textbf{(16)}$$

Simplifying and applying the definition of I **(7)** to **(16)** we obtain

$$E \bullet L \bullet P = \frac{I_{\min}}{I_{ACT}} \bullet \frac{I_{ACT}}{MaxIssue} \bullet Tcycle \bullet \frac{1}{T^{ACT}} \quad \textbf{(17)}$$

Simplifying and applying the definition of $T_{\min}$ **(9)** to **(17)** we obtain

$$E \bullet L \bullet P = \frac{T_{\min}}{T^{ACT}} \quad \textbf{(18)}$$

The second member of equality **(18)** is the definition of ***G*** as shown by **(12)**, thus

$G = E \bullet L \bullet P$ QED

Proposition **(13)** allows us to define the *Global Efficiency* as a space in which the X axis corresponds to *Load Efficiency*, the Y axis corresponds to *Bookkeeping Efficiency* and the Z axis corresponds to the *Execution Efficiency* of a task. The level

of *Global Efficiency* is represented by the volume of the parallelepiped of length L, width P and heigth E. We call the parallelepiped that represents a task in the *Global Efficiency Space* the *Global Efficiency Parallelepiped* of that task. A task with an ideal level of *Global Efficiency*, ($G = 1$) is represented as a unitary side cube. We call this unitary side cube the ideal cube. As the unitary value is the upper bound for a task, every task would be represented in the *Global Efficiency Space* by a parallelepiped contained inside the ideal cube.
The difference between the volume of the ideal cube and the volume of the *Global Efficiency Parallelepiped* of a task represents the level of *Global Inefficiency* of the task.

3 Applying the Indices to a Case Study: Two-dimensional Convolution

We selected two-dimensional convolution (CONV) to represent a typical image processing task.
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 [8], regularization [3], morphological operations [7]. 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 pixels 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 sketched in figure 1[5]:
Indices *x* and *y* control the shifting of the mask over the input image, whereas indices *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.

Definition 18
A loop is a level n loop if and only if it contains n more nested loops[6].

We ran our experiments on an HP9000 725/100 [1] workstation using the following picture size and mask size in pixel:
X mask size = 5; Y mask size = 5; X picture size = 128; Y picture size = 128;

[5] 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.
[6] *In case of loop unrolling, if and only if it would contain n more nested loops rerolling the unrolled loops.*

```
integer x,y,i,j,mask elements, input image, output image;
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
```

Figure 1 Pseudo Code for CONV task

Because of the nested structure of the task, level 0 loops program statements are executed a much larger number of times than program statements of loops of any other level. In particular, adopting the parameters above described, level 1 loop statements are executed 5 times less than level 0 loop statements; the contribution of a program statement located in a loop of level greater or equal than 2 is less than 1/25

```
L$0053                                          Start level 1 loop
       comb,<,n %r2,%r21,L$0043
       addl %r3,%r22,%r12
       stw %r12,-16(0,%r30)                     Integer to stack
       addl %r22,%r23,%r12
       fldws -16(0,%r30),%fr9L                  Stack to FP
       stw %r12,-16(0,%r30)                     Integer to stack
       xmpyu %fr9L,%fr10R,%fr9
       fldws -16(0,%r30),%fr8L                  Stack to FP
       fstws %fr9R,-16(0,%r30)                  FP to stack
       xmpyu %fr8L,%fr10L,%fr8
       ldw -16(0,%r30),%r12                     Stack to Integer
       fstws %fr8R,-16(0,%r30)                  FP to stack
       addl %r12,%r28,%r25
       ldw -16(0,%r30),%r12                     Stack to Integer
       copy %r12,%r24
L$0048                                          Start level 0 loop
       addl %r25,%r21,%r20
       addl %r9,%r20,%r20
       ldb 0(0,%r20),%r20
       stw %r20,-16(0,%r30)                     Integer to stack
       addl %r24,%r21,%r19
       addl %r19,%r2,%r19
       ldwx,s %r19(0,%r7),%r19
       fldws -16(0,%r30),%fr8L                  Stack to FP
       stw %r19,-16(0,%r30)                     Integer to stack
       fldws -16(0,%r30),%fr11L                 Stack to FP
       xmpyu %fr8L,%fr11L,%fr8                  Multiplication
       fstws %fr8R,-16(0,%r30)                  FP to stack
       ldw -16(0,%r30),%r12                     Stack to Integer
       ldo 1(%r21),%r21
       comb,>= %r2,%r21,L$0048
       addl %r12,%r26,%r26                      Accumulation, End
                                                level 0 loop
L$0043
       ldo 1(%r22),%r22
       comb,>=,n %r23,%r22,L$0053
       sub 0,%r2,%r21                           End level 1 loop
```

Figure 2 Assembly code for CONV direct implementation level 0 and level 1 loops

```
L$0043                                                  Start of Level 1
                                                        loop
        comb,<,n %r29,%r21,L$0033
        addl %r2,%r22,%r7
        stw %r7,-16(0,%r30)                             Integer to stack
        addl %r22,%r31,%r7
        fldws -16(0,%r30),%fr9L                         Stack to FP
        stw %r7,-16(0,%r30)                             Integer to Stack
        xmpyu %fr9L,%fr11L,%fr9
        fldws -16(0,%r30),%fr8L                         Stack to FP
        fstws %fr9R,-16(0,%r30)                         FP to stack
        xmpyu %fr8L,%fr10R,%fr8
        ldw -16(0,%r30),%r7                             Stack to integer
        fstws %fr8R,-16(0,%r30)                         FP to stack
        addl %r7,%r24,%r28
        ldw -16(0,%r30),%r7                             Stack to integer
        copy %r7,%r23
L$0038                                                  Start of level 0
                                                        loop
        addl %r28,%r21,%r20
        addl %r23,%r21,%r19
        addl %r19,%r29,%r19
        fldwx,s %r20(0,%r4),%fr8R
        fldwx,s %r19(0,%r3),%fr8L
        fmpy,sgl %fr8R,%fr8L,%fr8R                      Multiplication
        ldo 1(%r21),%r21
        comb,>= %r29,%r21,L$0038
        fadd,sgl %fr10L,%fr8R,%fr10L                    Accumulation, End
                                                        of level 0 loop
L$0033
        ldo 1(%r22),%r22
        comb,>=,n %r31,%r22,L$0043
        sub 0,%r29,%r21                                 End of level 1 loop
```

Figure 3 Assembly code for CONV implementation with the application of DTO.

of the contribution of a program statement located within level 0 loop. Thus in our analysis we focus on level 0 and level 1 and neglect the effects of program statements of higher levels; this introduces an approximation error of about 4%.

The value of L_{min} can be easily calculated: the task needs to load a complete frame and the mask at the beginning and to store a complete frame at the end.

Level 0 loop body executes:

1. the multiplication between the current pixel value and the current mask value:
2. the accumulation of the previous result.

To complete CONV we need to perform DIMX • DIMY • XMASK • YMASK level 0 loop bodies.

Assuming that both the load/store operations and the processing operations are evenly distributed in the whole task we can calculate the values of L_{min}, P_{min} and I_{min} normalized for level 0 and level 1 loop bodies alone. If we accept the above mentioned approximation errors we can use these values to calculate the metric parameters of the task.

We obtain:

$L_{min} = 0.48$ $P_{min} = 12.00$ $I_{min} = 12.48$

Looking at the assembly code of figure 2 we can measure I_{ACT} and L_{ACT}. We obtain:

$I_{ACT} = 98$ $L_{ACT} = 48$ $P_{ACT} = 50$
Thus:
$L_{add} = 47.52$ $I_{ACTR} = 50.48$ $P_{add} = 38$
and
$\lambda = 0.52$ $\pi = 0.25$
Given a value of MaxIssue equal to 2 and a CycleTime equal to ten nanoseconds [1][6] substituting in **(9)** we obtain:
$T_{min} = 851993/2*0.00001 = 4.26$ ms.
For this implementation of CONV T^{ACT} is equal to 121 ms, thus
$\gamma = 0.035$
From **(13)** we can calculate the value of ε:
$\varepsilon = 0.27$
The efficiency parameters of the direct CONV implementation are quite low, as they vary from 0.25 to 0.5. This fact is mainly due to the absence of an integer multiplier in the HPPA 7100LC processor. This lack forces the processor to transfer the values of pixels and mask elements from the integer data path to the FP data path to execute a fixed point multiplication and then to transfer the results back to the integer registers. Beside, the only way to transfer data from the integer data path to the FP data path and back is to execute a store and load in memory which causes a large increment of L_{add}. Notice that the load/store steps through memory slow down the processing both because of the intrinsic latency of FP load/store operations and because of the fact that they prevent the CPU from exploiting superscalarity. In fact they prevent the CPU from overlapping the small FP part of the task with the large integer part of the task. Thus the task has a low level of execution efficiency.
For all these reasons a very beneficial source level transformation would be a Data Type Optimization [2] (DTO) converting the whole problem from integer to floating point.

4 Quantitative Analysis of the Effects of Data Type Optimization on Two-dimensional Convolution

As the algorithm has not changed the values of P_{min}, L_{min} and I_{min} are the same we calculated for the base implementation.
Figure 3 shows the assembly code for level 0 and level 1 loops of CONV implemented with the application of DTO.
Looking at the assembly code of figure 3 we can calculate I_{ACT} and L_{ACT}. We obtain:

	λ	π	ε	γ
Direct Implementation	0.52	0.25	0.27	0.035
DTO	0.72	0.27	0.38	0.073

Table 1 Performance Metrics of the CONV task with and without DTO.

$I_{ACT} = 63$ $L_{ACT} = 18$
Thus:
$L_{add} = 17.52$ $P_{add} = 33$ $I_{ACTR} = 45.48$
These yield:
$\lambda = 0.72$ $\pi = 0.27$
The value of T_{min} is the same we calculated in the previous section, namely 4.26 ms.
For this implementation of CONV T^{ACT} is equal to 58.5 ms, thus
$\gamma = 0.073$
From **(13)** we can calculate the value of ε:
$\varepsilon = 0.38$
In table 1 we see the metric parameters of CONV with and without the application of DTO.
Although the value of γ is still very low, the application of DTO doubled the global efficiency of CONV; nevertheless this increment is not uniform in all the components of the global efficiency. The main benefit of the application of DTO is in the load efficiency, in fact the value of λ increased by 0.20. This is mainly due to the fact that the application of DTO allowed the processor to perform all the data calculations directly in the FP data path and there is no need to perform integer to FP and back data conversions.
On the other side, we can see in table 1 that the application of DTO allows only a very slight increment of the level of bookkeeping efficiency of the task: 0.02. This fact is mainly due to the fact that the DTO source level transformation has no effect on the number of ALBO instructions but, on the contrary, allows the usage of a more efficient instruction mix. The increased efficiency of the instruction mix is shown by the increment of π, in fact the application of DTO allows an increment of π of 0.11.

5 Concluding Remarks

In this paper we have identified and analyzed the main sources of inefficiency that cause a poor performance of ILP architectures in IPPR applications, namely unnecessary instructions and idle CPU cycles, and have defined a set of indices to measure them. We have used these indices to analyze the efficiency of a typical IPPR task, namely two-dimensional convolution (CONV), on an ILP HP 725/100 workstation. We have used the performance indices obtained through the aforementioned analysis to plan a very beneficial source level program transformation, namely Data Type Optimization (DTO), and to assess the impact of that transformation on the task performance. The final performance assessment allowed us to evaluate how DTO contributed to reduce the effect of the three sources of inefficiency.

Bibliography

[1] 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.

[2] Baglietto P., Maresca M., Migliardi M. and Zingirian N., Image Processing on High Performance RISC Systems, Proc. of IEEE, Vol. 84 n. 7, pp 917-930, July 1996

[3] 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.

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

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

[6] Hewlett Packard, HP9000 Series 700 Models 725/100.

[7] 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.

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

[9] 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.

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

[11] 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.