On Applying Graph Theory to ILP Analysis

Technical Note TN-UAH-AUT-GAP-2005-01-en

Raúl Durán, Rafael Rico

Department of Computer Engineering, Universidad de Alcalá, Spain

January 2005

Versión en español:

Aplicación de la teoría de grafos al análisis del paralelismo a nivel de instrucción

Nota técnica TN-UAH-AUT-GAP-2005-01-es

Raúl Durán, Rafael Rico

Departamento de Automática, Universidad de Alcalá, España

Enero 2005

Abstract:

The differences found between the superscalar performance in x86 and non-x86 processors and the peculiar characteristics of the x86 instruction set architecture recommend to carry out a thorough analysis of the available parallelism at the machine language layer. However, computer architecture evaluation requires new tools that complement the customary simulations and, in this sense, the traditional graph theory can help to create a new frame for fine-grain parallelism analysis.

Starting off from graph theory basic foundations, new concepts are introduced from *reduced valence* to *data dependence matrix D*, both characterizing a code sequence in a mathematical manner. This matrix fulfills a number of properties and restrictions and provides information about the ability of the code to be processed concurrently. Among other details, a relation between the *critical path length* and the *parallelism degree* along with techniques to calculate it from the matrix *D*, are presented.

Finally, it is explained how different data dependence sources can be composed, thus providing a mechanism to analyze their final influence on the parallelism degree. These techniques are applied to an example from which some conclusions are derived.

Index words: Evaluation of computer architectures, instruction level parallelism, instruction set architecture, graph theory, DDG-based quantification.

1. Introduction: new challenges in computer architecture evaluation.

Quantitative evaluation is a crucial point in the computer architecture research. As Kevin Skadron *et al.* explain the simulation has become the first evaluation tool both in industry and research [35]. Unfortunately, the construction of good simulators and the selection of appropriate workloads have become appalling tasks. These difficulties have led to a work focused just on fields where the tools have a tested quality while setting aside really interesting research topics as, for example, the multiprocessing.

Almost contemporary with the work referred to in the previous paragraph and also presented in *IEEE Computer*, Dror G. Feitelson indicates that the performance measures are usually employed to compare the quality of several systems without taking into account that differences can arise from the evaluation methodology itself [14]. Specifically, he concludes that metric and workload can affect the results since they are liable to interact.

Other aspects to consider in the quantitative evaluation could be the following [35]:

- the sequential programming language used to program the simulators does not contribute to the correct description of the targeted parallel systems
- the most used benchmark programs (SPEC) are not properly characterized
- the statistical analysis commonly used (mean, standard deviation, etc.) is not rigorous because the bursty behavior of many computation processes does not follow a Gaussian distribution
- usually the published results are not verified independently

As to the scanty independent verification, Skadron suggests that it could be caused by the fact of not being properly rewarded. We would like to add that it is often impossible to reproduce the experiments because no sufficient information about the simulator, the workload, assumptions or parameter settings is provided. Reaching, at least, the same degree of reproductibility that we observe in other research fields would be desirable.

Consequently, the moment has arrived for shifting gears and facing the new challenges arisen in computer architecture evaluation. Skadron's work supplies some recommendations enumerated by a group of researchers who met in December of 2001 with the aim of discussing these subjects. Among them, the quest for analytical methods has seemed to us particularly interesting.

We agree with Skadron that certain hostility is detected towards the analytical methods in congresses and journals. This should not be so, since the analytical model contributes to understand those aspects not uncovered by simulation. Moreover, the analytical model can be useful to validate the simulator and to predict the behavior of some architectural proposals.

As is common in other research fields, the mathematical formalization facilitates the description of phenomena, allows to predict behaviors, supports reproductibility and simplifies the knowledge transference.

a. Applying graph theory to fine-grain parallelism analysis.

We have found it convenient to take the first steps for the application of graph theory to fine-grain parallelism analysis. Graph theory provides an efficient mathematical formalization that promises to be very useful in the analytical modelling we are aiming at, encouraged by the aforementioned works. Moreover, graphs have already been successfully applied to the study of other aspects of computation like, for example, the medium- and coarse-grain parallelism extracted by compilers [3, 45, 46]. Padua and Wolfe claim that the parallel code will be as good as its corresponding data dependence graph [27]. Graphs are usefully employed in other fields: data structures [2, 3, 8, 20], operating systems design [33], software description [10, 11], automata logic [28], electronic design [9], and so on.

Indeed, there are more reasons to approach this matter. First of all, the study of the fine-grain parallelism has been and is a significant field, where the simulation is the most frequently used evaluation technique. Perhaps it is for that reason that it needs, more than other fields, analytical modelling. Secondly, it is very common that, once the factors that contribute to the fine-grain parallelism are identified, the impact of the instruction set architecture is forgotten, and the stress goes directly to bare physical aspects. Maybe the cause is, again, the simulator: modelling physical aspects is easier than modelling the behavior of different instruction set characteristics. We have spotted a significant difference between the fine-grain parallelism degree reported in the literature for x86 and for non-x86 processors. This has led us to think that, perhaps, the impact of the instruction set architecture on fine-grain parallelism availability has been underrated.

b. ILP difference between x86 and non-x86 processors.

Indeed, as previously mentioned, the quantification of parallelism at the instruction level is one of the most popular subjects in computer architecture. Some papers describe the hardware solution space as, for example, the one of Jouppi and Wall [19] or the one of Smith and Sohi [37]. In the literature, numerous studies can be found identifying limiting factors, quantifying their effects, providing possible solutions and evaluating the results. Among the evaluated factors, one can find: inhibition of parallelism due to conditional jumps [31], branch prediction with an ideal fetch unit and a 32-entry instruction window [36], instruction window size under renaming, branch prediction and several sizes of

the register file [42, 43], precise interruptions [7], control flow limitations [22], memory disambiguation [40], performance of hybrid branch predictors [12], use of multithreading for increasing physical resource usage [41], pressure on the register file under the SPEC92 programs for a 256-entry reorder buffer [13], number of memory ports in combination with the reorder buffer and the register file sizes [15], data prediction [23, 44], impact of the SPEC95 programs [29], code reordering effects on branch prediction [30], or early register release [24], just to mention some of the most important ones.

All these works have in common that they present non-x86processors, the evaluation has been made by means of simulators and the identified limiting factors always are related to the physical layer. The reported IPC average results are in the range 2.5-15, peaking around 30 IPC (for example, in the case of memory disambiguation [40]) and with a higher limit of 50 IPC (case presented in [15]) under ideal conditions.

The works where x86 instruction set processors have been used are less frequent. In those cases, the reported levels of parallelism are not so good as the reviewed ones in the previous paragraph. Y. N. Patt group has proposed techniques such as pipelining scheduling [38] or instruction lookahead fetch [25] on x86 processors, thus obtaining average IPC values in the range between 0.5 and 3.5 in the best situations, the value being slightly over 1 for most cases. Huang and Xie measure parallelism at the microoperation level (MLP) [18]. MLP average is 1.32 without renaming and 2.20 with renaming. Bhandarkar and Ding characterize the Pentium Pro performance by means of hardware counters included in the processor itself [5]. The CPI for SPECint95 is in the range 0.75 to 1.6.

To summarize, it seems clear that the available parallelism in x86 processors is lower than the one obtained in non-x86 processors, according to the available articles on the topic. This led us to conjecture that the instruction set architecture (ISA) itself may impose an important limiting factor on the available fine-grain parallelism.

c. The x86 instruction set and the superscalar model.

For the sake of binary compatibility with previous processors, which has provided so far unquestionable benefits, the x86 instruction set architecture has inherited design characteristic suitable to older requirements but clearly harmful in the scope of superscalar processing. Among other undesirable characteristics from the point of view of parallelism explotation, we could mention the following: the dedicated register use, the implicit operands (associated to the operation and not specified by the programmer), the use of state register and the large number of registers involved in the address arithmetic.

The effect of these undesirable characteristics becomes apparent in the over-ordering of the code, imposed by the machine language layer through data dependencies, and not strictly necessary to preserve the computational meaning of the compiled task. As a result, the instructions appear more coupled at the physical layer than one should expect just observing the corresponding high level program [32].

The instruction set architecture has a significant impact in the availability of fine-grain parallelism before reaching the physical layer, which can reduce exploitable parallelism degree at run time. Fig. 1 schematically illustrates the factors that affect the available parallelism at each layer of the computation process.

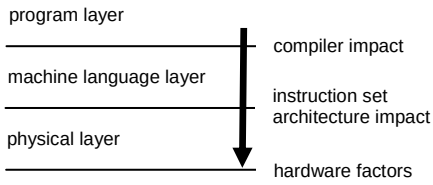


Fig. 1. Factors affecting the available parallelism in the different layers of the computation process.

Nowadays, the works on instruction set architectures are scarce and rather oriented to study specific machines like, for example, DSP processors [34], to offer simulation tools [26] or to describe other collateral aspects.

Regarding the works focused on the x86 instruction set, the analyses at the machine language layer are limited to counting and to calculate usage frequencies. Adams and Zimmerman made a study on the usage frequency of x86 instructions under DOS applications [1], but this work does not include the use of operands. More recently, Huang and Peng have made countings of the instruction usage with different operands [17], though they do not analyze the dependencies among operands. Finally, Huang and Xie present a study evaluating parallelism at microoperations level (RISC kernel of Intel CISC processors) where the operation and the address modes are considered [18].

d. Metrics.

IPC (Instructions Per Cycle) is by far the metric most often employed in parallelism quantification at the instruction level. It consists of finding the ratio of the number of instructions *vs.* the run time in cycles (simulated). This method demands a complex simulator, if the measurements are to be precise; moreover, the assumptions and simplifications have a significant effect on the final result. The measurements strongly depend on the characteristics of the physical floorplanning and, therefore, this metric is amenable to the study of the different architectural proposals at the physical layer level.

There exists another metric, slightly different from the previous one. The parallelism degree is now the ratio of the total number of executed instructions

vs. the scheduled time grids. It is also oriented to the study of the physical resources organization .

When measuring events other than the instruction execution, such as, for example, cache misses, branch prediction misses, etc., we use a similar metric: the ratio with respect to the total.

It is necessary to point out that the metric based on the IPC is commonly associated with a statistical treatment that has its own impact on the results, as recalled above. Most often, values, such as the average or the standard deviation, are supplied disregarding an intrinsic fact related to the computational process: parallelism appears to come in bursts, as indicated by Kumar [21]. Therefore, these statistical results are flawed since the measured events seldom adjust to a Gaussian distribution, as Skadron's work points out [35].

A much less used metric consists of measuring the critical path length of a code sequence. The critical path is the longest chain of data dependencies that can be found among the instructions of a sequence. Thus, the least number of computation steps necessary to process the sequence will be equal to that length, should enough physical resources be available.

Then, the greatest parallelism degree will be equal to the total number of processed instructions divided by this length, which gives an indication as to the amount of instructions that can be processed concurrently in each computation step.

The algorithm used to measure the dependence chains annotates which instruction produces each writing and reading in each data location and then counts the number of correlated writing-reading "links".

Parallelism quantification by means of the critical path length has been used previously by Kumar. In this case, the study was performed on source code written in FORTRAN, taking sentences instead of instructions, and variables instead of physical data locations [21]. It is, therefore, a work that we can locate at the program layer. It is very interesting because it reports a much greater range of parallelism than the one found in the literature for the works related to the physical layer. Undoubtfully, the own computation process impairs parallelism availability.

The critical path has also been used sometimes to evaluate characteristics of the physical layer. Specifically, the works of Austin and Sohi [4], Postiff, Greene, Tyson and Mudge [29] and Stefanovic and Martonosi [39] obtain their results by means of the critical path length. Really these studies start from execution traces on which the measurement is performed by applying certain rules that model the characteristics of the targeted hardware. For this reason, the results thus generated assume the specifications of the physical layer.

Although the critical path measurement method permits to quantify the parallelism degree at the language layer machine, irrespective of the restrictions at the physical layer, no study has been found in the literature. It would be interesting to have some information about the possible parallelism degradation at this layer.

An alternate measurement method is based on the data dependence graphs (DDG). It consists of building the DDG of a real code sequence. In such case, the characterization is independent of the physical implementation, since it is located in a previous step of the computation process, namely, in the machine language layer. Nevertheless, it includes the impact of the compilation process. These measures suggest possible hardware architectures suitable to take advantage of the partial order specified by the graph without imposing further restrictions to the out-of-order execution process.

The DDG-based quantification is a powerful tool of analysis when the matrix representation is used because it permits a mathematical processing. Thus we can determine not only the critical path length and, consequently, the parallelism degree of a instruction window, but also the life span of operands, data sharing reuse, the most important sources of dependencies, the parallelism distribution and other significant parameters. Moreover, the mathematical formalization will also permit to compose different data dependence sources with the purpose of finding the possible origin of the code coupling.

Finally, the DDG can also include the specifications of the physical layer using formal hardware descriptions.

2. Representation of instructions sequences by graphs. A revision.

The data dependencies in an instruction sequence can be represented by a graph *G(V, E)*, where *V* is the set of vertices and *E* is the set of edges. Each vertex in *V* represents an instruction and each edge in *E* a data dependence. Any two vertices related by an edge are said to be adjacent. The graph topology can be represented by the so-called *adjacency matrix A*:

$$a_{ij} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ vertices are adjacent;} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

A is a symmetric matrix of dimension $n \times n$ where n is the number of instructions in the graph, with null diagonal and $a_{ij} \in \{0, 1\}$ [6, 16].

We define the *incidence matrix B* as:

$$b_{ij} = \begin{cases} 1, & \text{if } e_i \text{ is incident with vertex } v_j; \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

If the graph has n vertices and m edges, the dimension of *B* is $n \times m$.

This graph bears two possible orientations: either "instruction i produces data for instruction j" (orientation σ) or "instruction j consumes data from (depends on) instruction i" (orientation $\overline{\sigma}$). In each case, the edges point in opposite directions with a

complementary meaning: the first orientation shows the data flow whereas the second one shows the data dependencies. Figure 2 shows a simple x86 code sequence (16 bits subset) and illustrates both orientations of the graph.

```
0: XOR BX, BX
1: MOV SI, 07AB
2: MOV BL, ES:[SI]
3: INC SI
4: CMP BL, DL
5: JNE/JNZ IP+F7
```

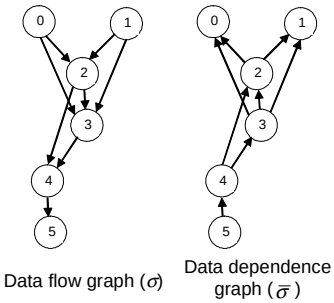


Fig. 2. Example of a code sequence and the two possible orientations of the associated graph.

We can define the incidence matrix B^{σ} with respect to orientation σ, as the matrix n x m:

$$b_{ij}^{\sigma} = \begin{cases} +1, & \text{if } v_i \text{ is the incoming end of } e_j; \\ -1, & \text{if } v_i \text{ is the outgoing end of } e_j; \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

We define the *valence* of a vertex as the total number of edges that are incident with this vertex. The *valence matrix* Δ is an n x n diagonal matrix where the *(i, i)* component is the valence of vertex i. The adjacency matrix and the incidence matrix for the orientation σ are related as follows:

$$Q = B^{\sigma} \cdot (B^{\sigma})^t = \Delta - A \tag{4}$$

The $B^{\sigma} \cdot (B^{\sigma})^t$ product is known as the *Laplacian matrix* Q and is independent of the orientation.

Moreover, the graph representation based on the adjacency matrix A enjoys the properties of the characteristic polynomial $\det(\lambda I - A)$, a central aspect in graph theory [6, 16].

The adjacency matrix A, the incidence matrices for both orientations B^{σ} and $B^{\bar{\sigma}}$ and the valence matrix Δ corresponding to the proposed example are shown in Fig. 3.

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$B^{\sigma} = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$B^{\bar{\sigma}} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

$$\Delta = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 3. A, B^{σ}, $B^{\bar{\sigma}}$ and Δ matrices corresponding to the proposed example.

Notice that adjacency and valence matrices are orientation invariant, whereas the incidence matrix depends on it. The first two arise from the incidence relation while the third depends on the incidence relation and the direction of the edges.

a. Reduced valence.

We define the *reduced valence* of a vertex as the total number of edges that enter this vertex. The reduced valence depends, therefore, on the orientation selected.

The σ-oriented *reduced valence matrix* V^{σ}, is an n x n diagonal matrix where the component *(i, i)* is the σ-oriented reduced valence of vertex i.

Considering only one orientation, it is possible to give a special definition for the incidence matrix. Thus, we can define the *reduced incidence matrix* I^{σ} with respect to orientation σ, as:

$$i_{ij}^{\sigma} = \begin{cases} +1, & \text{if } v_i \text{ is the incoming end of } e_j; \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

If the graph has n vertex and m edges, the dimension of I^{σ} is n x m.

Proposition 1: The $I^{\sigma} \cdot (I^{\sigma})^t$ product generates the reduced valence matrix V^{σ} for the selected orientation.

Proof: If we compute the *(i, j)* product component:

$$\left[I^{\sigma} \cdot \left(I^{\sigma} \right)^t \right]_{ij} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{kj}^{\sigma^t} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\sigma} \tag{6}$$

However, $i_{ik}^{\sigma} \cdot i_{jk}^{\sigma} \neq 0$ if and only if $i = j$, because each edge has just one incoming end. Since $i_{ik}^{\sigma} \in \{0, 1\}$, then $(i_{ij}^{\sigma})^2 = i_{ij}^{\sigma}$ and so

$$\left[I^{\sigma} \cdot \left(I^{\sigma} \right)^t \right]_{ij} = \begin{cases} \sum_{k=0}^{m-1} i_{ik}^{\sigma} = \begin{cases} \text{number of incoming} \\ \text{ends if } i = j; \end{cases} \\ 0, \text{ if } i \neq j \end{cases} \tag{7}$$

This result agrees with the definition of the oriented reduced valence matrix.

$$V^{\sigma} = I^{\sigma} \cdot (I^{\sigma})^t \tag{8}$$

■

Figure 4 presents the reduced incidence and the reduced valence matrices for both orientations corresponding to the example.

$$I^{\sigma}=\begin{pmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\1&1&0&0&0&0&0&0\\0&0&1&1&1&0&0&0\\0&0&0&0&0&1&1&0\\0&0&0&0&0&0&0&1\end{pmatrix}$$

$$I^{\bar{\sigma}}=\begin{pmatrix}1&0&1&0&0&0&0&0\\0&1&0&1&0&0&0&0\\0&0&0&0&1&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&0\end{pmatrix}$$

$$V^{\sigma}=\begin{pmatrix}0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&2&0&0&0\\0&0&0&3&0&0\\0&0&0&0&2&0\\0&0&0&0&0&1\end{pmatrix}$$

$$V^{\bar{\sigma}}=\begin{pmatrix}2&0&0&0&0&0\\0&2&0&0&0&0\\0&0&2&0&0&0\\0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&0\end{pmatrix}$$

Fig. 4. Reduced incidence and reduced valence matrices for both orientations: I^{σ}, $I^{\bar{\sigma}}$, V^{σ} and $V^{\bar{\sigma}}$.

The inspection of Fig. 3 and Fig. 4 allows us to verify that the incidence matrix can be derived from the reduced incidence matrices and the valence matrix from the reduced valence matrices. We have:

$$B^{\sigma}=I^{\sigma}-I^{\bar{\sigma}} \tag{9}$$

$$\Delta=V^{\sigma}+V^{\bar{\sigma}} \tag{10}$$

The previous equations formalize a quite intuitive relation that we will not prove here.

Moreover, we can observe that the values of the reduced valence matrix V^{σ} correspond to the I^{σ} components summed by rows, as expressed in equation (7). The meaning is obvious: since the reduced valence is the total number of incoming ends to each vertex, it will suffice to count the number of entries different from 0 in each row of the I^{σ} matrix. The same holds for the opposite orientation $\bar{\sigma}$.

b. Data dependence matrix *D*.

We defined the data dependence matrix *D* as:

$$d_{ij}=\begin{cases}1, & \text{if } i \text{ instruction depends on } j;\\ 0, & \text{otherwise}\end{cases} \tag{11}$$

Therefore, i instruction is associated to a data dependence vector $\vec{d}_i$ whose j-th component is 1 if a direct dependence on instruction j through any data exists and 0 otherwise. Thus, the matrix D is the set of all data dependence vectors $\vec{d}_i$ of a code sequence. We want to emphasize that the matrix D represents the direct data dependence path or data dependence path of length one, that is, instruction i consumes a data processed directly by instruction j with no interveners.

Figure 5 shows the data dependence matrix for the example of a code sequence.

$$D=\begin{pmatrix}0&0&0&0&0&0\\0&0&0&0&0&0\\1&1&0&0&0&0\\1&1&1&0&0&0\\0&0&1&1&0&0\\0&0&0&0&1&0\end{pmatrix}$$

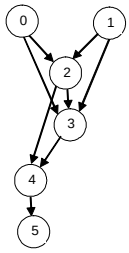


Fig. 5. Data dependence matrix *D* and the data flow graph.

The matrix *D* considers just the incoming edges whereas the matrix *A* considers both incoming and outgoing edges. In this sense, the graph orientation that the matrix *D* shows is the one that corresponds to the data flow.

Although the matrix *D* is used commonly in compiler theory (for example, in loop transformations [45, 46]) it has been used neither in the machine language analysis nor in the parallelism characterization at instruction level. Authors consider the reason for this may be that the analyses of instruction set architectures are considered overridden now or that it is assumed that their influence in the superscalar execution is minor, thus trusting too much the physical layer.

c. Topological properties and ILP restrictions for the data dependence matrix.

One of the aims of the graph theory algebra is to precisely determine how the graphs properties are exposed in the algebraic properties of their associated matrices. We try to extract, in addition, consequences in the scope of parallel instruction processing.

- **The vertex labelling should not affect the properties of the data dependence matrix**. The matrix *D* can be associated to a graph that consists of a vertex set *V*, an edge set *E* and an incidence relation, that is, a subset of *V* x *E*. Since the set $\{v_0, v_1, v_2, \ldots, v_{n-1}\}$ corresponds to an arbitrary vertex labelling, the properties of matrix *D* should be invariant under arbitrary permutations of rows and columns. We are interested in those properties that remain invariant under these permutations.

As a starting point, in the scope of instruction level parallelism, we must consider a sequence of instructions that keeps the natural sequentiality of Von Neumann programming model fixed by the strictly precedence order in which they are written in the program. The original vertex labelling of the graph (instructions) is induced by the own program counter. We will term this labelling *programmatic labelling*.

The programmatic labelling will facilitate the discovering of properties invariant under arbitrary permutations.

• **There exists a precedence relation among the data dependence graph vertices**. Any computable task entails some precedence relation or partial ordering among the tasks (instructions) to perform, since it is a process developed in an ordered and finite succession of steps. A certain ordering is essential to the algorithm and, often, it is stressed when passing from a layer to the next in the computation process as Fig 1 illustrates (from algorithm to program, from program to compiled image in machine code, from machine code to physical layer).

• **An instruction does not depend on itself**. A data cannot have the same instruction as source and as destination. Consequently, matrix *D* has null diagonal. That is, $d_{ii} = 0 \quad 0 \le i \le n-1$.

This is certain even in the loops. A loop is a compact way to write a code sequence that, in its expanded version, repeats a series of operations but on different data. Each new iteration implies a new instance of the loop body but on new data. The execution of a loop body requires a conditional branch instruction between iterations. In that case, the conditional branch instruction can be inserted in the data flow graph as a special operation that manipulates the program counter register and keeps apart the loop body instructions in each iteration.

• **The data dependencies are not symmetrical**. An instruction cannot depend on another that depends at the same time on it, since this situation does not establish a precedence relation but a data dependence cycle. Consequently, the matrix *D* is not symmetric. Mathematically: $d_{ij} \neq d_{ji} = 1 \quad \forall i, j \le n-1$.

Therefore, if *D* is not symmetric, the associated characteristic polynomial for any data dependence matrix $p(G; \lambda) = \det(\lambda I - D)$, irrespective of the incidence relation, is always the same : $p(G; \lambda) = \lambda^n$. The descriptive value of this polynomial is very poor.

• **There is a graph vertex labelling under which the matrix *D* is lower triangular**. An instruction depends only on the precedents in the program, and never on the following ones (principle of causality). This means that the instructions only process data given by the instructions written above in the program and never from those which are about to come in the sequence. According to this, the programmatic labelling of the dependence graph vertices generates a lower triangular matrix *D* because $d_{ij} = 0$ whenever $j > i$. In other words, a labelling that fullfils the order imposed by the program counter insures that all components *(i, j)* of *D* are 0 when $j > i$ because an instruction cannot have data dependencies on the following ones. The matrix *D* will be termed *canonical* when it is lower triangular and will be denoted D_c.

As a summary, we enumerate the properties that the data dependence matrix *D* fullfils:

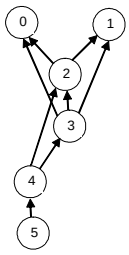
- it is square of dimension $n \times n$, n being equal to the number of instructions included in the code sequence graph,
- its values are binary, that is, $d_{ij} \in \{0, 1\}$,
- it has null diagonal,
- it is not symmetric and, consequently, its characteristic polynomial is equal to λ^n, irrespective of the graph incidence relation,
- there is at least a vertex labelling –programmatic labelling– under which it is lower triangular and presents the canonical form D_c,

- different vertex labellings represent permutations that are isomorphisms,
- it encloses a partial ordenation of instructions.

d. The data dependence matrix and the adjacency matrix.

There is an immediate relation between the orientation that generates the data dependence graph and the one that generates the data flow graph. Both share the same edge set (incidence relation) changing only the orientation. We can assure that if "*i* depends on *j*" then "*j* produces data for *i*". If $(d_{ij})^{\sigma} = 1$ under an orientation, then $(d_{ji})^{\bar{\sigma}} = 1$ under the opposite, which corresponds to definition of transposed matrix. Notice that the D^t matrix in Fig. 6 takes into account the incoming ends of the graph, which correspond to "data dependence orientation".

$$D^t = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



Data dependence graph

Fig. 6. D^t matrix and the regarded graph orientation.

Since the adjacency matrix *A* values both ends of each edge, then *A* is the sum of the data dependence matrix *D* plus its transpose D^t, *i.e.*:

$$A = D + D^t \tag{12}$$

This means that we are able to construct the adjacency matrix *A* from *D*, thus allowing us to work with the complete matrix representation of the graph and, hence, to take advantage of all the algebraic power associated to its characteristic polynomial.

e. The data dependence matrix and the reduced valence.

As the data dependence matrix values only one end of each edge, then some relationship with the reduced valence might be implicit there. Also, the inspection of matrix *D* in Fig. 5 and I^{σ} matrix in Fig. 4 allows us to appreciate certain similarity in the location of non-null components. Indeed, it must be so

because the same graph lies under both matrices: I^{σ} relates edges and vertices, whereas D relates vertices to vertices. The mathematical formalization that we have developed points to the same direction, since equations (9), (10) and (12) suggest a relationship between the matrix D and the the reduced incidence matrix I^{σ} through equation (7).

Proposition 2: the product $I^{\sigma} \cdot I^{\bar{\sigma}^t}$ generates the matrix D.

Proof 1: By substituting equations (9), (10) and (12) in equation (4) we obtain:

$$(I^{\sigma} - I^{\bar{\sigma}}) \cdot (I^{\sigma} - I^{\bar{\sigma}})^t = V^{\sigma} + V^{\bar{\sigma}} - D - D^t \qquad (13)$$

Some computation on the left-hand side yields:

$$\begin{aligned} &(I^{\sigma} - I^{\bar{\sigma}}) \cdot (I^{\sigma^t} - I^{\bar{\sigma}^t}) = \\ &= I^{\sigma} \cdot I^{\sigma^t} + I^{\bar{\sigma}} \cdot I^{\bar{\sigma}^t} - I^{\sigma} \cdot I^{\bar{\sigma}^t} - I^{\bar{\sigma}} \cdot I^{\sigma^t} \end{aligned} \qquad (14)$$

We know that equation (8) relates the reduced valence to the reduced incidence so we can simplify the previous result:

$$I^{\sigma} \cdot I^{\bar{\sigma}^t} + I^{\bar{\sigma}} \cdot I^{\sigma^t} = D + D^t \qquad (15)$$

If D values the incoming ends for the data flow graph orientation and D^t those for the data dependence graph orientation, we have:

$$I^{\sigma} \cdot I^{\bar{\sigma}^t} = D \qquad [16]$$

■

Proof 2: If we calculate the product component *(i, j)*:

$$\left[I^{\sigma} \cdot \left(I^{\bar{\sigma}} \right)^t \right]_{ij} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{kj}^{\bar{\sigma}^t} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\bar{\sigma}} \qquad (17)$$

The summation goes through all the edges in the index k. The product is different from zero only for the k-th edge if it enters the vertex i ($i_{ik}^{\sigma} = 1$) and leaves vertex j or, in other words, enters vertex j under the opposite orientation ($i_{jk}^{\bar{\sigma}} = 1$). But then, this agrees with the definition of the data dependence matrix D that we have presented in equation (11).

$$\sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\bar{\sigma}} = d_{ij} \qquad (18)$$

Hence

$$D = I^{\sigma} \cdot I^{\bar{\sigma}^t} \qquad (16)$$

■

It has been already shown that the reduced valence for an orientation can be obtained by counting the incoming ends (entry equal to 1) along rows, as equation (7) suggests. Observing the relationship expressed in equation (16) and the similarity displayed by the D and I^{σ} matrices, regarding the non-null values disposition, it is enticing to think about using matrix D in order to find the reduced valence.

Proposition 3: The counting of edges along the rows of matrix D permits to generate the reduced valence matrix for the data flow graph orientation.

Proof: Suppose the relation is true and replace each entry of D by the value given in the equation (18):

$$v_{ii} = \sum_{p=0}^{n-1} d_{ip} = \sum_{p=0}^{n-1} \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{pk}^{\bar{\sigma}} \qquad (19)$$

A simple computation leads to

$$v_{ii} = \sum_{k=0}^{m-1} \sum_{p=0}^{n-1} i_{ik}^{\sigma} \cdot i_{pk}^{\bar{\sigma}} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot \left(\sum_{p=0}^{n-1} i_{pk}^{\bar{\sigma}} \right) \qquad (20)$$

However, any given edge, say k, is incident only on one vertex and hence:

$$v_{ii} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot 1 = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \qquad (21)$$

This corresponds to equation (7) and proves that finding out the counting of incoming edges is equivalent to run through the rows either of the I^{σ} matrix or the D matrix. This fact allows us to give a new definition of reduced valence matrix:

$$v_{ij}^{\sigma} = \begin{cases} \sum_{k=0}^{n-1} d_{ik} = \text{number of edges if } i = j; \\ 0, \text{ if } i \neq j \end{cases} \qquad (22)$$

■

It becomes apparent that the data dependence matrix D exhibits a great descriptive ability, since it allows:

- to recover the adjacency matrix A and its characteristic polynomial,
- to calculate the reduced valence for data flow graph orientation,
- to describe the data dependence graph by means of its transpose matrix D^t,
- to obtain information about the code sequence without resorting to any other mathematical tool.

f. Code coupling.

The concept of reduced valence is very useful in the scope of instruction processing parallelism. Remember that if we select the data flow graph orientation σ, the reduced valence gauges how much an instruction is coupled (linked) with the rest, that is, it hints at how weaved a code sequence is. So, a large reduced valence indicates that an instruction consumes data coming from several instructions and, therefore, it must stall its own execution till all these data are available. Consequently, a larger coupling implies a potentially greater partial ordering of the code, since

there are more precedence relationships: The more partial ordering, the less available parallelism.

The diagonal of the reduced valence matrix for data flow graph orientation (V^{σ}) informs us about the coupling that each instruction bears.

The trace of the reduced valence matrix for data flow graph orientation is the total number of edges, that is, it quantifies the amount of dependence relationships among instructions. This value gives information about how much coupled (linked, weaved) a code sequence is. We denote this parameter as coupling C:

$$C = \text{tr } V^{\sigma} \tag{23}$$

The summation by rows in matrix D gives the reduced valence of each instruction for data flow graph orientation (22). However, we have already indicated that there is a vertex labelling for which the matrix D becomes lower triangular (canonical form D_c). In that case, and remembering that the diagonal is null, we have:

$$v_{ij}^{\sigma} = \begin{cases} \sum_{k=0}^{i-1} d_{cik} = \text{number of edges if } i = j \neq 0; \\ 0, \text{ otherwise} \end{cases} \tag{24}$$

Consequently, the reduced valence matrix trace for the orientation corresponding to the data flow graph (coupling C) will be:

$$C = \text{tr}V^{\sigma} = \sum_{i=0}^{n-1} v_{ii}^{\sigma} = \sum_{i=1}^{n-1}\sum_{k=0}^{i-1} d_{c_{ik}} \tag{25}$$

The maximum number of data dependencies (edges) in the graph is given by all the possible ordered vertex pairs, which is precisely the binomial coefficient $\binom{n}{2}$. Hence, the coupling C is bounded by:

$$0 \leq C \leq \binom{n}{2} = \frac{n^2 - n}{2} \tag{26}$$

With the purpose of obtaining a coupling measurement which is independent of the amount of instructions in the sequence, we define a normalized coupling, C_N, as the ratio C *vs.* the number of instructions n in the code sequence. When C_N is zero there is no dependence; in the worst case, each instruction depends on all the precedent ones.

$$0 \leq C_N \leq \frac{n-1}{2} \tag{27}$$

g. Data reuse.

If the orientation is the opposite $\overline{\sigma}$, that is, the edges are incoming to indicate which vertex the data are given to (data dependence graph), the reduced valence quantifies the data reuse. If the data generated by an instruction is used by many others, it must be stored, thus consuming temporary storage resources.

The reduced valence matrix diagonal for data dependence graph orientation ($V^{\overline{\sigma}}$) informs about the reuse of data produced by each instruction. When this value is 0, it means that the data is not consumed. If the value is 1, it means that each data generated by an instruction is consumed only by another one. With no reuse, the lifespan of the data in the storage elements is negligible. There is a special circumstance that takes place when all the non-null values of the $V^{\overline{\sigma}}$ diagonal are equal to 1. This situation is illustrated in the Fig 7.(a). Mathematically the product $D \cdot D^t$ generates a diagonal matrix that coincides with V^{σ}. Clearly all $d_{ik} \cdot d_{jk}$ products are zero since, if instruction i depends on instruction k, no other instruction can depend on instruction k.

When the reuse value is larger than 1 it means that several instructions consume a data. In this case, it is interesting to know the life span (storage time) of the data. In the Fig. 7.(b) and (c), two possible situations are illustrated. In both cases, instruction 0 generates data for instruction 1 and instruction 2. In the case (b) the data is consumed in the next computation step whereas in the case (c) this is not possible because there is a data dependence path of length 2 that is also coupling instruction 2 to the 0. We deduce that the life span of a data produced by instruction i and consumed by instruction j must be at least equal to the longest data dependence path between both instructions. For that reason we talk about the minimum life span T_{min}.

$$T_{min} = n \quad \text{such that } [D^n]_{ij} \neq 0 \text{ and } [D^{n+1}]_{ij} = 0 \tag{28}$$

The case of the Fig. 7.(d) shows that this time eventually depends on the scheduling criteria (at the physical layer) since instruction 1 can be scheduled immediately one computation step before instruction 3 or at the same time as the instruction 0, causing the storage time to be 2 computation steps.

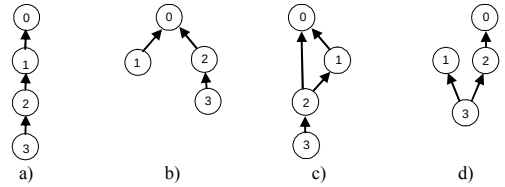


Fig. 7. Diferent classes of data reuse.

In the case of reuse, it is useless to calculate its value for the complete graph as we have done for the coupling C because the storage resources only make sense when referred to individual data requirements. Moreover, computing the trace of $V^{\overline{\sigma}}$ is nothing but counting edges in the graph –by columns in D instead of by rows as in the equation (24)–, which coincides with the coupling C.

h. Data dependence paths of length larger than 1.

Given a graph *G(V, E)*, a path of length *l* (edges) from vertex v_i to v_j is, by definition, a finite sequence of $l + 1$ different vertices that begins in v_i and finishes in v_j, such that two consecutive vertices are adjacent [6, 16].

The data dependence matrix *D* represents the data dependence paths of length 1 or direct dependencies between instructions. Nevertheless, data dependence chains between instructions of larger length can exist. For example, if $d_{ij} = 1$ and $d_{jk} = 1$ then instruction *i* depends on instruction *k* through the instruction *j* by a path of length 2. We can say that there exists a data dependence path of length 2 from instruction *i* to instruction *j* running through, at least, one of the instructions in the graph, if the following holds:

$$\left(d_{i0} \cdot d_{0j}\right) + \left(d_{i1} \cdot d_{1j}\right) + \cdots + \left(d_{in-1} \cdot d_{n-1j}\right) = \\ = \sum_{k=0}^{n-1} \left(d_{ik} \cdot d_{kj}\right) \neq 0 \qquad (29)$$

But this value corresponds to the *(i, j)* entry of the product $D \cdot D$ and, therefore, the matrix D^2 represents the data dependence paths of length 2.

- **D^l represents the data dependence path of length *l* (edges)**. Generalizing the reasoning for the paths of length 2, each power of matrix *D* represents the dependence paths of length equal to the power degree. In other words, given a graph *G(V, E)*, the number of data dependence paths of length *l* from v_i to v_j with orientation σ, is the *(i, j)* entry in the matrix D^l. In [6] we can find a proof by induction of this proposition which is based on the adjacency matrix *A* and can be extended to the case of the data dependence matrix *D*. Note that the length is measured in edges.
- **The *n*-th power of *D* is null**. The maximum length of a data dependence path is $n - 1$ (edges), *n* being the number of instructions in the code sequence. Hence, D^n will be necessarily null.
- **There are no cycles of dependencies**. There are no closed dependence paths because this case does not set a strict precedence relation and, consequently, the graph must be acyclic (DAG or Directed Acyclic Graph). If cycles were permitted, an instruction would depend on itself through others and so the task would not have solution in a finite number of steps. Therefore, no instruction can depend on itself under any path of length *l*. Algebraically, the diagonal of any power of the data dependence matrix (D^l) must be null: $d^l_{ii} = 0$, $1 \le l \le n - 1$, $1 \le i \le n$.

From another standpoint, if the *n*-th power of *D* were not the null matrix, then there would exist cycles of dependencies and, therefore, the matrix *D* would not represent a computable task (solutionable in a finite number of steps).

Regarding data dependence paths of value larger than one, we can emphasize, as a summary:

- each power of matrix *D* represents the dependence paths of length equal to the degree of the power (measured in number of edges),
- D^n must necessarily be null; otherwise it would not represent a computable task,
- there can be no cycles of dependencies and, hence, the diagonal of any power of *D* is null.

i. Critical path length and degree of parallelism.

One of the most important pieces of information that we can extract from data dependence matrices is the available instruction level parallelism degree contained in the code sequences represented by the matrices. This information is independent of the limitations that the physical layer can impose later on. It concerns the machine language layer only and it derives from the algorithm, the program implementation, the compiler impact and the instruction set architecture itself.

The available parallelism is inversely related to the length of the data dependence chains between instructions. The longer these chains are, the stricter is the partial ordering of the code sequence imposing a very sequential instruction execution and limiting the ability of concurrent processing. On the contrary, short data dependence chains imply a weak ordering between instructions amenable for concurrent execution.

Given a code sequence, represented by its data dependence matrix *D*, we define the *critical path length* L*(D)* (briefly L) as the length of the longest data dependence path. However, two possible metrics exist: measuring edges –as explained in the preceding Section, according to traditional graph theory– or measuring computation steps –appropriate to instruction level computation environment–.

There is an immediate relation between the two metrics of *critical path length* and the number of vertices involved. If the length of the critical path L involves $l + 1$ vertices, then this path has *l* edges and the minimum number of computation steps required to process the associated code sequence is $l + 1$. The execution of independent instructions frees a data dependence of the longest chain in each computation step. After *l* computation steps, we will be ready to process the last instruction(s) of the sequence (free of any dependence). We need, therefore, $l + 1$ computation steps to finish the execution of $l + 1$ instructions of the critical data path chain.

We know that D^l represents the data dependence paths of length *l* (edges). According to this, the first power of *D* that is identically zero indicates the length of the critical data path. This is:

$$\mathrm{L} = l \text{ computation steps if and only if } D^l = 0 \qquad (30)$$

With this metric, L is bounded in the following way:

$$1 \le \mathrm{L} \le n \qquad (31)$$

The minimum length L is 1: this means that there are no data dependencies among instructions and, should the resources be available, all the instructions could be processed concurrently in only one computation step. The maximum value of L is *n*. In that case, each computation step admits the issuing of just one instruction and the sequentiality is complete: *n* computation steps are required to carry out the code processing.

The more instructions are included in the data dependence graph, the longer the potential length of the critical path L. With the purpose of obtaining a measurement of instruction level parallelism independent of the number of vertices in the graph, we define the *normalized critical data path length*, L_N, as the ratio critical path length (L) expressed in computation step *vs.* the maximum value of L (L_{max}).

$$L_N = \frac{L}{L_{máx}} = \frac{L}{n} \tag{32}$$

When L_N approaches 1 there is no parallelism, and the nearer to 0, the more the parallelism the code bears.

$$L_N \in (0,1] \tag{33}$$

Also we define the *parallelism degree*, G_p, as the reciprocal of L_N expressed in computation steps ($G_p = (L_N)^{-1}$). G_p goes from 1 (absence of parallelism) to *n* (maximum parallelism degree).

$$G_p \in [1, n] \tag{34}$$

The meaning of G_p is clear. It indicates the number of instructions that can be concurrently processed in each computation step. To issue *n* instructions in a computation step means that no data dependence among instructions exists and they are capable of being processed concurrently. On the contrary, if only one instruction can issue in each computation step, this means that the sequentiality is absolute.

The code sequence of Fig. 2 has a graph with a critical path length of 4 edges, or 5 computation steps, and a parallelism degree G_p of 5/6, namely, 5/6 instructions are processed in each computation step. Indeed, the example is quite sequential ($G_p = 0.83 \approx 1$): we are able to issue 2 instructions concurrently only in the first computation step.

Let us discuss an interesting point. The parallelism degree is a function of the number of instructions in the code sequence, that is, $G_p = f(n)$. It might seem reasonable to think that as *n* increases, so does G_p, and, therefore, the larger the number of instructions in the sequence, the more potential parallelism. Following this rationale, the instruction window size of some superscalar processors has been enlarged, thus expecting to find more independent instructions ready to be executed simultaneously. Against this idea, the simulations of David Wall found an asymptotic behavior [42, 43]. After a certain point no more parallelism is uncovered, even examining more and more instructions. From another point of view, the dependence chain length grows when the code sequence grows because the logical resources of the software architecture of the instruction set are limited.

j. Algorithmic calculation of the critical path length.

The calculation of the successive powers of the matrix *D* allows the determination of the critical path length by means of an algebraic procedure, according to equation (30). Nevertheless, the calculation based on the matrix product is very heavy (complexity $O(n^4)$ product operations) and renders this method useless.

The algorithm to find the partial ordering that a graph bears also serves to find the critical path length. This procedure has a lower computation cost (complexity $O(n^2)$ sum operations) that makes it attractive for the automation of the analysis of code sequences. It consists of setting the precedence between instructions based on their dependencies. In each computation step, we list the independent instructions and we free of data dependencies those they supply data to. All the instructions that belong to the list of a computation step share the same level of precedence and, therefore, can be processed concurrently. The number of required computation steps to order all the instructions is the length of the critical path. On the other hand, the extracted partial ordering gives a scheme for the scheduling of the instructions, useful to assign physical resources. The following figure illustrates the pseudocode of this algorithm.

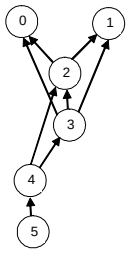
```
/* Partial ordering procedure */
1: GIVEN: computation step step = 0, instructions list for step s List(s),
dependence matrix D;

2:  while (unordered instructions left)
3:  {
4:      while (independent instructions left)
5:      {
6:              List(s) = Add_Independent_List_Step(s);
7:      }
8:      Remove_Dependencies (List(s), D);
9:      step ++;
10: }
```

Fig. 8. Partial ordering algorithm.

This algorithm allows both the evaluation of the critical path length (expressed in computation steps) and the elaboration of the partial ordering listing.

Figure 9 shows the partial ordering associated to the graph example. Notice that this partial ordering allows, if there are enough physical resources, to process the code sequence in 5 computation steps giving rise to a parallelism degree of $G_p = 5/6 = 0.83$ instructions per computation step.



Computation step 0: 0, 1
Computation step 1: 2
Computation step 2: 3
Computation step 3: 4
Computation step 4: 5

Fig. 9. Partial ordering corresponding to example graph.

3. Dependence sources composition.

If a code sequence can be represented by the data dependence matrix D and this is a formalization of the partial ordering implicit in the code, we can accurately analyze the impact of the different dependence sources since the final dependence map is really a simple composition of those sources. If D_{s1} and D_{s2} are matrices arising from two different sources of dependencies, then the resultant matrix D will be:

$$D = D_{s1} \text{ OR } D_{s2} \tag{35}$$

In other words, $d_{ij} = d_{s1_{ij}} \text{ OR } d_{s2_{ij}}$ means that instruction i depends on instruction j because of cause 1 or because of cause 2 or because of both causes simultaneously.

Figure 10 illustrates the composition of sources of data dependencies with a simple example. Consider the code sequence presented in the example of Fig. 2. The aim is to decompose the data dependencies into those due to explicit operands and those due to implicit operands (those arising from the operation code). According to this we will have:

$$D_{op\ expl.} \text{ OR } D_{op\ impl.} = D$$

$$\begin{pmatrix} 0&0&0&0&0&0 \\ 0`0&0&0&0&0 \\ 1&1`0&0&0&0 \\ 0&1&1`0&0&0 \\ 0&0&1&0`0&0 \\ 0&0&0&0&0`0 \end{pmatrix} \text{ OR } \begin{pmatrix} 0&0&0&0&0&0 \\ 0`0&0&0&0&0 \\ 0&0`0&0&0&0 \\ 1&0&0`0&0&0 \\ 0&0&0&1`0&0 \\ 0&0&0&0&1`0 \end{pmatrix} = \begin{pmatrix} 0&0&0&0&0&0 \\ 0`0&0&0&0&0 \\ 1&1`0&0&0&0 \\ 1&1&1`0&0&0 \\ 0&0&1&1`0&0 \\ 0&0&0&0&1`0 \end{pmatrix}$$

Fig. 10. Dependence sources composition.

It is noticeable that the composition of both data dependence sources has caused a sequentiality in the resulting graph larger than the one originally produced by each source on its own.

Unfortunately, the resulting or final data dependence matrix does not admit the inverse process or decomposition, that would allow us to make a more methodical and precise analysis of the impact of each data dependence source. The merged dependencies cannot be traced back to their origins.

a. Some possible compositions

The construction of the data dependence graphs admits, as previously seen, the composition of different categories. In a first approach, the composition of true dependencies, antidependencies and output dependencies would be helpful.

$$D = D_{true} \text{ OR } D_{output} \text{ OR } D_{antidependence} \tag{36}$$

This allows us to distinguish the type of data dependence and to predict whether it has computational relevance (true dependencies) or it is due simply to the physical storage reuse (anti-dependencies and output dependencies).

However, we can carry out a finer analysis if we differentiate possible data dependence sources. In this case we will have:

$$D = D_{s1} \text{ OR } D_{s2} \text{ OR } D_{s3} \text{ OR } \cdots \tag{37}$$

These diverse data dependence sources could be:

- a limited number of general purpose software registers,
- register reuse,
- implicit operands use,
- condition codes writes,
- memory address computation,
- stack traffic, etc.

Besides, any of them can be considered in the three variants presented in the first place: true dependencies, antidependencies and output dependencies.

b. Critical path length of the composition.

All the properties and procedures proposed for the matrix D can be applied to each of the matrices that represent different data dependence sources (D_{sn}).

It is interesting to consider the evolution of the critical path length of the resulting matrix as a function of its components. The resulting graph does not have a critical path length necessarily equal to the sum of the critical paths of its components because the data dependence chains can overlap. It is not possible, therefore, to determine in a direct way the length of the composition based on the components. At most, we can obtain a bound for it.

As for the upper bound, we know that the final length is limited by the number of instructions present in the graph. Besides, we can affirm that the critical path length of the composition will never be longer than the sum of the critical path lengths of the components assuming the most unfavorable case in which all of them contribute to create data dependence chains. Therefore, the upper bound will be imposed by the minimum of both:

$$\min\left\{\sum_i \mathrm{L}(D_{si}), n\right\} \geq \mathrm{L}(D) \tag{38}$$

Equation (38) uses the metric based on computation steps.

As for the lower bound, it will never be less than the one corresponding to the longest component of the

critical path, assuming that all the rest are overlapped with it, *i.e.*, there are no cross dependencies among the components. Formally:

$$\mathrm{L}(D) \geq \max_i \{\mathrm{L}(D_{si})\} \tag{39}$$

Actually, the critical path length L(*D*) of the compound matrix *D* is a number bounded by:

$$\min\left\{\sum_i \mathrm{L}(D_{si}), n\right\} \geq \mathrm{L}(D) \geq \max_i \{\mathrm{L}(D_{si})\} \tag{40}$$

That is, L(D) will always be equal to or greater than the longest data dependence path of components and it will always be smaller than or equal to either the sum of the components lengths or *n* (the number of instructions present in the code sequence), whichever is smaller.

c. Illustrative example.

An example based on an x86 code sequence is proposed. Some features of the instruction set architecture related to its behavior in superscalar execution are enumerated in Section 1.c. The assembler x86 instructions use explicit operands –those written by the programmer– and implicit operands –those necessarily associated to the operation code–. This feature, along with others, such as the dedicated use of certain registers, the peculiar way to access memory positions or the use of the state register, renders very interesting the analysis of the different data dependence sources [32].

In Table 1 we propose an x86 code sequence that can well represent a typical basic block. The 16 bits subset has been used for the sake of simplicity. The operands used by each operation are classified into two major sets: read operands and written operands. Within each one of these main sets, the operands are grouped by their functionality: data mapped in registers or memory, registers used in the calculation of effective memory addresses, registers involved in stack accesses and state register. The explicit operands (included in the format of the instruction) are set apart from the implicit operands (associated necessarily to the operation code). Each one of these categories represents a possible data dependence source whose impact we can study separately.

In some cases it is not easy to know the actual functionality that an operand access will have. It is for that reason that the explicit writing on operands related to the address computing or related to the stack appears empty in the table.

With respect to the memory locations, the most pessimistic situation is assumed, namely, the memory is considered a single resource as far as the data dependence generation is concerned: the accesses are not different because of their address. This is not true for the stack since the accesses are ordered by the stack pointer.

From the information in Table 1 the data dependence matrices *D* are built for each one of the selected sources and for the three types of data dependencies: true dependencies, antidependencies and output dependencies. Figure 11 shows all the results.

Table 1. Code sequence and the operands used in each operation.

code sequence		Readed operands								Writed operands							
		explicit				implicit				explicit				implicit			
		reg	adr	stack	cc	reg	adr	stack	cc	reg	adr	stack	cc	reg	adr	stack	cc
0:	MOV DX, 6B42	-	-	-	-	-	-	-	-	DX	-	-	-	-	-	-	-
1:	MOV CS:[BX], DX	DX	CS, BX	-	-	-	-	-	-	MEM	-	-	-	-	-	-	-
2:	SUB BX, AX	AX, BX	-	-	-	-	-	-	-	BX	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
3:	MOV AH, 30	-	-	-	-	-	-	-	-	AH	-	-	-	-	-	-	-
4:	INT 21	-	-	-	-	AX, CS, IP	-	SS, SP	Flags	-	-	-	-	AX, BX, CX, CS, IP	-	SP	IF, TF
5:	CLI	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	IF
6:	MOV BP, [BX][SI]	MEM	BX, SI	-	-	-	DS	-	-	BP	-	-	-	-	-	-	-
7:	MOV DS, DX	DX	-	-	-	-	-	-	-	DS	-	-	-	-	-	-	-
8:	OR SS:[DI], AX	AX, MEM	SS, DI	-	-	-	-	-	-	MEM	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
9:	CWD	-	-	-	-	AX	-	-	-	-	-	-	-	AX, DX	-	-	-
10:	XOR CX, BX	BX, CX	-	-	-	-	-	-	-	CX	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
11:	DEC DI	DI	-	-	-	-	-	-	-	DI	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
12:	INC SI	SI	-	-	-	-	-	-	-	SI	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
13:	MOV BL, ES:[SI]	MEM	ES, SI	-	-	-	-	-	-	BL	-	-	-	-	-	-	-
14:	TEST [BX][DI], AL	AL, MEM	BX, DI	-	-	-	DS	-	-	MEM	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
15:	JNE/JNZ IP+F7	-	-	-	ZF	-	-	-	-	-	-	-	-	IP	-	-	-

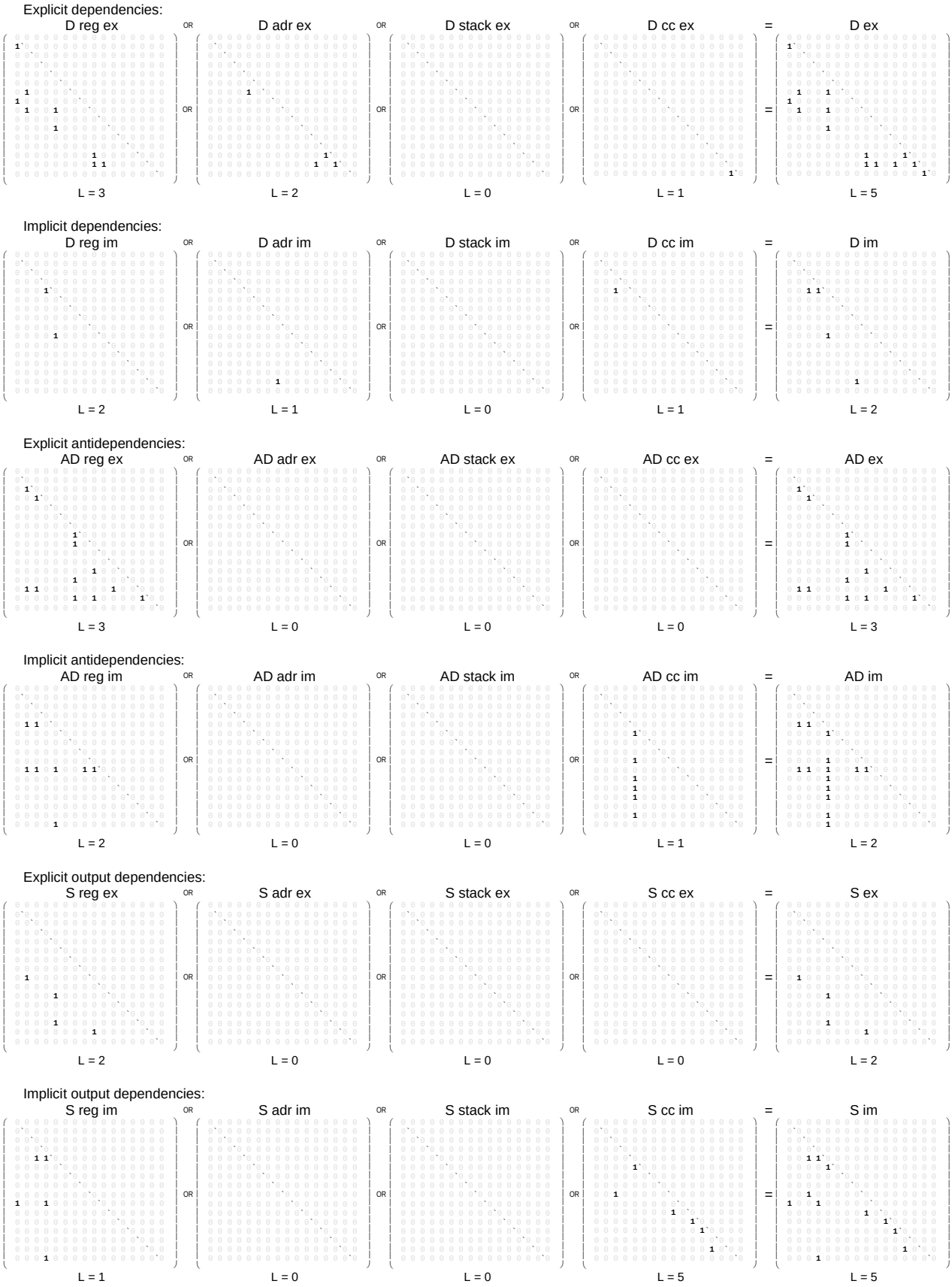


Fig. 11. Different data dependence sources composition for code sequence example.

The dependencies have been combined by groups, composing the four basic sources for each type of dependence and separately in explicit and implicit operands. The critical path length L has been computed for each matrix. The lengths of the composed matrices meet the bounds shown in equation (40).

The sequential composition of each dependence source with each basic type has been evaluated producing the graph of Fig. 12. The composition order does not change the final result but the used sequence is not absolutely arbitrary. We began with true dependencies, which are those that do not have hardware solution, and then we examined the antidependencies and the output dependencies. Within each basic type, the analysis began with the explicit operands and then the implicit operands. Finally, the operand functionality has also an order: we began with the accesses to operands with a greater computational meaning because they represent the direct manipulation of data, followed by the memory address computation (that turns out to be essential to variable access but that it is not directly involved in main computation); next, the impact of the operands related to the stack access is evaluated and we finish with the state register accesses whose only function is to save the condition codes used in conditional branches.

The composition sequence illustrates how data dependence over-ordering generated by collateral computational load (memory address computation, stack accesses and state backup) gets added to the dependencies with a true computational meaning.

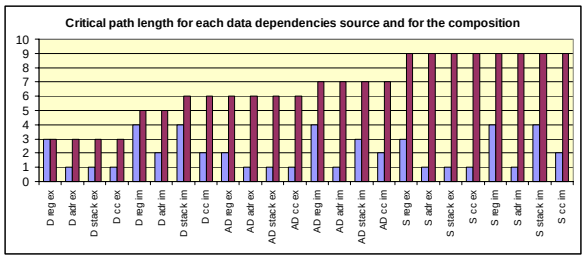


Fig. 12. Critical path length for each data dependence source and for the composition.

From the example some immediate consequences can be extracted whose generalization should be verified in works to come.

We began with true dependencies. The dependencies generated by the memory address computation are added to the dependencies between explicit data registers. Consequently, it is a cause of parallelism degradation at the instruction level. The next parallelism degradation comes from condition codes. In this case, the true dependencies through implicit operands no longer affect the length of the critical path. We find, therefore, two sources with remarkable impact on the parallelism degradation:

- memory address computing; and
- condition codes.

The antidependencies among explicit registers provide a new parallelism degradation that, in our example, does not change until the last contribution due to the output dependencies in implicit operands, specially the one due to condition codes. That is, among the data dependencies due to physical resources limitations, the most remarkable ones seem to be:

- explicit use registers in antidependencies; and
- condition codes in output dependencies.

4. Conclusions and future work.

A model of analysis applicable to the computation process at the machine language layer has been proposed. It allows the quantitative evaluation of the impact of both the instruction set architecture and the compilation procedure itself.

The topological properties and restrictions that the matrix D has to fullfil in the ILP scope have been identified along with a method that uses the matrix D to quantify the ordering degree of code, the data reuse and its life span. A metric to evaluate the available parallelism degree has been defined as well.

It is showed how the different data dependence sources can be composed, thus allowing a precise analysis of the impact of each one on the final parallelism degree.

The proposed analytical model has been applied to the evaluation of some aspects of the x86 instruction set architecture and valuable information has been obtained [32].

In future works, the contribution of each one of the dependence sources will be studied, analyzing its behavior on a greater set of test programs. It seems also possible to extend the use of the graphs to model the limitations of the physical layer and the processes of allocation-scheduling.

As a summary, we enumerate the contributions that have been made throughout the technical note:

- we introduce the data dependence matrix D from the traditional graph theory definitions and supported by the novel concept of the reduced valence,
- several topological properties and restrictions that the data dependence matrix D must satisfy in the instruction parallel processing scope, have been identified,
- a relation between the matrix D and the adjacency matrix A, traditionally used in graph theory, has been determined,
- D^t, the transposed matrix of D, has been identified as the way to relate the two graph orientations,
- the relation between the matrix D and the reduced valence matrix for an orientation has been proved,
- the concept of code coupling has been introduced as a method to measure the ordering degree of a code sequence,
- a way to quantify the data reuse degree has been established,
- the relation between the matrix D and the data dependence paths length longer than 1 has been identified,

- the relation between the critical path length and the powers of matrix *D* has been identified,
- variables relating critical path length and available parallelism degree in code have been defined,
- an algorithmic method to calculate the critical path length have been proposed,
- it has been shown how different data dependence sources can be composed and some dependencies sources have been suggested,
- the critical path length of a composition has been bounded,
- some possible lines of future work have been suggested from an illustrative example.

5. Referencies.

[1] T. L. Adams and R. E. Zimmerman, "An analysis of 8086 instruction set usage in MS DOS programs," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989, pp. 152 - 160.

[2] A. V. Aho, J. E. Hopcroft, J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Co., 1983.

[3] A. V. Aho and J. Ullman. *Foundations of Computer Science*. Computer Science Press, 1992.

[4] T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," in *Proceedings of the 19th International Symposium on Computer Architecture*, 1992, pp. 342-351.

[5] D. Bhandarkar and J. Ding, "Performance characterization of the Pentium Pro processor," in *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, 1997, pp. 288 –297.

[6] N. L. Biggs, *Algebraic Graph Theory* (2nd. edn.), ISBN: 0-521-45897-8, Cambridge University Press, 1993.

[7] M. Butler, Tse-Yu Yeh, Y. Patt, M. Alsup, H. Scales and M.Shebanow. "Single Instruction Stream is Greater than Two," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 163-174.

[8] T. H. Cormen, C. E. Leiserson and R. L. Rivert. *Introduction to Algorithms*. Mit Press, McGraw Hill, 1996.

[9] R. David, "Modular design of asynchronous circuits defined by graphs," *IEEE Transactions on Computers*, vol. C-26, 8, pages 727-737, August 1977.

[10] A. L. Davis and R. M. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, 2, February, 1982.

[11] J. B. Dennis, "Concurrency in software systems," in *Advanced Course in Software Engineering*, Springer-Verlag, pages 111-127, 1973.

[12] S. McFarling, "Combining Branch Predictors", *W.R.L. Technical Note TN-36*. Digital Equipment Corporation. Palo Alto, CA. June 1993. Available at: http://www.research.compaq.com/wrl/techreports/index.html.

[13] K. I. Farkas, N. P. Jouppi and P. Chow. "Register File Design Considerations in Dynamically Scheduled Processors," in *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA)*, 1996, pp. 40–51.

[14] D. G. Feitelson. "Metric and Workload Effects on Computer Systems Evaluation," *IEEE Computer*, vol. 36, 9, September, 2003.

[15] J. González and A. González. "Identifying Contributing Factors to ILP," in *Proceedings of the 22nd Euromicro Conference (Euromicro'96)*, 1996, pp. 45-50. Short Contribution.

[16] C. D. Godsil and G. F. Royle, *Algebraic Graph Theory*, ISBN: 0-387-95220-9, Springer-Verlag, 2001.

[17] I. J. Huang and T. C. Peng, "Analysis of x86 Instruction Set Usage for DOS/Windows Applications and Its Implication on Superscalar Design," *IEICE Transactions on Information and Systems*, Vol.E85-D, No. 6, pp. 929-939, June 2002. (SCI).

[18] I. J. Huang and P. H. Xie, "Application of Instruction Analysis/Scheduling Techniques to Resource Allocation of Superscalar Processors," *IEEE Transactions on VLSI Systems*, vol. 10, no. 1, pp. 44-54, February 2002.

[19] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272-282, April 1989.

[20] L. Joyanes and I. Zahonero. *Estructura de datos. Algoritmos, abstracción y objetos*. Mc Graw Hill, 1998.

[21] M. Kumar, "Measuring parallelism in computation intensive scientific/engineering applications," *IEEE Transactions on Computers*, 37(9), pp. 1088-1098, 1988.

[22] M. Lam and R. Wilson. "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 46-56.

[23] M. H. Lipasti and J. P. Shen. "Exceeding the Dataflow Limit Via Value Prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226-237, 1996.

[24] T. Monreal, V. Viñals, A. González and M. Valero. "Hardware Schemes for Early Register Release," in *Proceedings of the International Conference on Parallel Processing (ICPP'02)*, 2002, pp. 5-13.

[25] O. Mutlu, J. Stark, Ch. Wilkerson and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, 2003, pp. 129–140.

[26] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr and A. Hoffmann. "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation," in *Proceedings of the 39th Design Automation Conference (DAC 2002)*, pp. 22-27, 2002.

[27] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, 29(12), pages 1184-1201, December 1986.

[28] C. A. Petri, "Comunication with automata," *Supplement 1 to Technical Report RADC-TR-65-377*, vol. 1, 1966.

[29] M. A. Postiff, D. A. Greene, G. S. Tyson and T. N. Mudge, "The Limits of Instruction Level Parallelism in SPEC95 Applications," in *Proceedings of the 3rd Workshop on Interaction Between Compilers and Computer Architecture*, 1998.

[30] A. Ramirez, J. L. Larriba-Pey and M. Valero, "The effect of code reordering on branch prediction," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 189-198, October 2000.

[31] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. C-21, pages 1405-1411, December 1972.

[32] R. Rico, J. I. Pérez, J. A. Frutos. "The impact of x86 instruction set architecture on superscalar processing," *Journal of Systems Architecture*, vol. 51-1, pages 63-77, January 2005.

[33] M. Silva. *Las redes de Petri: en la automática y la informática*. Editorial AC, 1985.

[34] P. Simonen, I. Saastamoinen, M. Kuulusa and J. Nurmi. "Advanced Instruction Set Architectures for Reducing Program Memory Usage in a DSP Processor," in *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA '02)*, pp. 477-479, 2002.

[35] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Hill and V. S. Pai. "Challenges in Computer Architecture Evaluation," *IEEE Computer*, vol. 36, 8, August, 2003.

[36] M. Smith, M. Johnson and M. Horowitz. "Limits on Multiple Instruction Issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 272-282.

[37] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," in *Proceedings of the IEEE*, 83(12), pp. 1609-1624, December, 1995.

[38] J. Stark, M. D. Brown and Y. N. Patt. "On Pipelining Dynamic Instruction Scheduling Logic," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 57-66.

[39] D. Stefanovic and M. Martonosi, "Limits and Graph Structure of Available Instruction-Level Parallelism," in *Proceedings of the European Conference on Parallel Computing (Euro-Par 2000)*, 2000.

[40] K. B. Theobald, G. R. Gao and L. J. Hendren, "On the Limits of Program Parallelism and its Smoothability," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 10-19, 1992.

[41] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392-403.

[42] D. W. Wall, "Limits of instruction-level parallelism," *W.R.L. Technical Note TN-15*. Digital Equipment Corporation. Palo Alto, CA. December 1990.
Available at: http://www.research.compaq.com/wrl/techreports/index.html.

[43] D. W. Wall, "Limits of instruction-level paralelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176-188, April 1991.
Also as:
W.R.L. Research Report 93/06. Digital Equipment Corporation. Palo Alto, CA. 1993. Available at: http://www.research.compaq.com/wrl/techreports/index.html.

[44] K. Wang and M. Franklin. "Highly accurate data value prediction using hybrid predictors," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 281-290, 1997.

[45] M. Wolfe. *High Performance Compiler for Parallel Computing*. Addison-Wesley, CA, 1996.

[46] H. Zima and B. Chapman. "Supercompilers for Parallel and Vector Supercomputers," *ACM Press Frontiers Series*, 1990.