Quantifying ILP by means of Graph Theory

Virginia Escuder, Raúl Durán, Rafael Rico
Department of Computer Engineering
Universidad de Alcalá
28871 Alcalá de Henares (Spain)
+34 91 885 66 15
{virginia.escuder, raul.duran, rafael.rico}@uah.es

ABSTRACT

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 of execution performance, a step beyond the classical static analysis performed by compilers.

Starting off from Graph Theory basic foundations, this paper introduces the data dependence matrix *D* supported by the novel concept of the reduced valence. The matrix *D* characterizes a code sequence in a mathematical manner, is endowed with a number of properties and restrictions, and provides information about the ability of the code to be processed concurrently. Among other details, some low complexity techniques to calculate parallelism degree from the matrix *D* are presented.

Keywords

Instruction level parallelism; Graph theory.

1. INTRODUCTION

Performance in the field of superscalar execution depends on many factors: the intrinsic parallelism of algorithms, the capabilities of the used high level language, the compilation process, the target machine instruction set and, of course, the physical layer. Figure 1 schematically illustrates the factors that affect the available parallelism at each layer of the computation process.

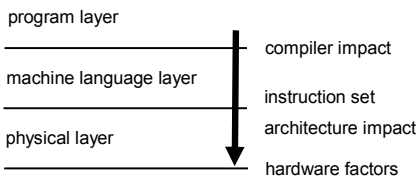

Figure 1. Factors affecting the available parallelism in the different layers of the computation process.

Valuetools'07, October 23–25, 2007, Nantes, France.

Nowadays, one of the most important objectives in Computer Engineering is code decoupling, in other words, avoiding data dependency among instructions in order to obtain full concurrency in superscalar processing of code. In particular, impacts from both the compiler [12] and the instruction set architecture [9, 14] can be responsible for an over-ordering of the code that has no solution in the physical layer and/or may cause increased execution complexity and power consumption. It is, therefore, important to steer the focus from the physical layer to the machine language layer and the program layer regarded on their own.

Moreover, considering a single unit for study (the instruction set and the hardware that should interpret it) has become a usual way to perform the research, under the assumption that this is a sounder computational approach. Another circumstance that has also contributed to the mentioned fact is the extensive use (sometimes abuse) of simulation as the performance evaluation method. Simulation does not differentiate between the impact on performance arising from the upper computation process layers and the impact from limited physical resources [15].

In this paper we propose an analytical model for the quantitative evaluation of ILP at the machine language layer based on Graph Theory, which provides an efficient and promising mathematical formalization for the analytical modeling of ILP.

Graphs had already been successfully applied to the study of other aspects of computation such as data structures [1, 3, 6] or software description [7, 8].

In particular, graphs have been traditionally applied to compiler extraction of medium- and coarse-grain parallelism [2, 17, 18] but using a purely static approach. Instead of this, we are interested in a dynamic approach allowing us to monitor the real execution of programs.

We propose a measurement method based on the data dependence graphs (DDG). It consists of building the DDG of a real machine code sequence. DDG-based quantification is a powerful tool of analysis when the matrix representation is used for a number of reasons. First of all, it permits a mathematical processing. We can determine the critical path length and, consequently, the parallelism degree of an instruction window. We can find out the life span of operands, data sharing reuse, parallelism distribution and other significant parameters.

Parallelism quantification by means of the critical path length has been previously employed in several works: In [11] it is used at the program layer and in [4, 13, 16] it is used to evaluate characteristics of the physical layer. But the main difference between these works and ours is the way in which the critical path

length is computed. In our work we derive the precise critical path length by means of our proposed mathematical formalization, whereas in the existing literature the critical path length is obtained by simply recording actual dependences thus losing any other information about the considered code fragment.

So far, the most often employed metric in parallelism quantification at the instruction level is IPC. Since IPC requires the measurement of instruction count as well as time, the results strongly depend on the characteristics of the physical implementation and therefore it is amenable to the study of the different architectural proposals at the physical layer level. However, this method demands a complex simulator, if the measurements are to be precise; moreover, the necessary assumptions and simplifications have a significant effect on the final result. For example, it is typically assumed that the measured events follow a Gaussian distribution, which is seldom the case, since parallelism appears to come in bursts [11]. This fact impairs the results [15].

2. THE THOERETICAL MODEL

2.1 Representation of instruction sequences as graphs

Data dependences in an instruction sequence can be represented as 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.

Traditionally, graphs allow two formalizations: as linked list style or as matrix style. Each of them presents pros and cons regarding mathematical treatment and memory consumption. In our work, we have selected the matrix style because it facilitates the operations we are interested in, though at the price of a (tolerable) higher memory consumption.

For the convenience of the reader, we introduce in this Section some concepts of Graph Theory (for more information see, for instance, [5, 10]) that will serve as environment for other novel concepts to be introduced later on.

In the matrix style, a graph topology can be represented by the so-called *adjacency matrix*[1] 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 n x n matrix where n is the number of instructions in the graph, with null diagonal and $a_{ij} \in \{0, 1\}$.

The *incidence matrix*[2] B is defined 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, then the dimension of B is n x m.

[1] The equivalent concept in the linked list style is called *adjacency list.*

[2] The equivalent concept in the linked list style is called *incidence list.*

In our formalism we use directed graphs. In a directed graph $G(V, A)$, each pair of vertices is connected by an arc from the set A, which is a directed edge, i.e., an ordered pair of distinct vertices. A directed graph may have two possible orientations corresponding to the following: either "instruction i produces data for instruction j" (orientation σ) or "instruction j consumes data from (depends on) instruction i" (orientation $\overline{\sigma}$). In either case, the arcs point in opposite directions and have a complementary meaning: the first orientation shows data flow whereas the second one records data dependences.

The incidence matrix B^σ with respect to orientation σ, is defined as the following n x m matrix:

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

The valence of a vertex is defined as the total number of arcs 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:

$$Q = B^\sigma \cdot \left(B^\sigma\right)^t = \Delta - A. \quad (4)$$

The $B^\sigma \cdot (B^\sigma)^t$ product is known as the *Laplacian matrix* Q. Adjacency, valence and Laplacian matrices are independent of the orientation.

Moreover, a graph representation using adjacency matrix A have the properties of the characteristic polynomial $\det(\lambda I - A)$.

2.2 Reduced valence

In this Section, we introduce the novel concept of *reduced valence*. We define the reduced valence of a vertex as the total number of arcs having an incoming end on 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 just one orientation, it is possible to formulate a special definition for the incidence matrix which we call the reduced incidence matrix I^σ with respect to orientation σ:

$$i_{ij}^\sigma = \begin{cases} +1, & \text{if } v_i \text{ is the incoming end of } a_j; \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

If the graph has n vertices and m arcs, 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.

$$V^\sigma = I^\sigma \cdot \left(I^\sigma\right)^t. \quad (6)$$

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

$$\left[I^\sigma \cdot \left(I^\sigma\right)^t\right]_{ij} = \sum_{k=0}^{m-1} i_{ik}^\sigma \cdot i_{jk}^\sigma. \quad (7)$$

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

$$\left[I^{\sigma} \cdot \left(I^{\sigma}\right)^{t}\right]_{ij} = \begin{cases} \sum_{k=0}^{m-1} i_{ik}^{\sigma} = \begin{cases} \text{number of incoming} \\ \text{ends if } i = j; \end{cases} \\ 0, \text{ if } i \neq j. \end{cases} \quad (8)$$

This result is in agreement with the definition of the oriented reduced valence matrix V^{σ} and so, the proof is completed. □

Proposition 2: The reduced incidence and the reduced valence matrices verify the following relations:

$$B^{\sigma} = I^{\sigma} - I^{\bar{\sigma}}, \quad (9)$$

$$B^{\bar{\sigma}} = I^{\bar{\sigma}} - I^{\sigma}, \quad (10)$$

$$\Delta = V^{\sigma} + V^{\bar{\sigma}}. \quad (11)$$

Proof: It is deduced directly from the definitions. □

2.3 Computational meaning of the reduced valence

The mathematical concept of reduced valence has a computational meaning that depends on the selected orientation. As stated in Section 2.1, there exist two orientations: the orientation σ ("instruction *i* produces data for instruction *j*") and orientation $\bar{\sigma}$ ("instruction *j* consumes data from instruction *i*").

The first one, which gives rise the reduced valence matrix V^{σ}, conveys the information about how the instructions are coupled: the diagonal element (*i*, *i*) shows the number of instructions on which instruction *i* depends.

The second one, which gives rise the reduced valence matrix $V^{\bar{\sigma}}$, conveys the information about data reuse: the diagonal element (*i*, *i*) shows how many instructions use the data produced by instruction *i*.

2.4 Data dependence matrix *D*

Another new concept introduce by us is the *data dependence matrix D* defined as:

$$d_{ij} = \begin{cases} 1, & \text{if } i \text{ instruction depends on } j; \\ 0, & \text{otherwise.} \end{cases} \quad (12)$$

Then, the rows of matrix *D* correspond to the vectors $\vec{d}_i$ of a code sequence. Each $\vec{d}_i$ holds the data dependence information for instruction *i*. Notice that by data dependence information we mean dependence on any operand involved in operation *i*.

This matrix is not to be confused with the matrix of the same name typically used in compiler theory. Though related, actually they do not store the same information: in loop transformation the information captured in the matrix is the static dependence distance (among iterations) [17, 18] not the dynamic dependence among instructions.

The matrix *D* represents the direct data dependence path or data dependence path of length 1, that is, instruction *i* consumes data processed directly by instruction *j* with no interveners. According to its definition, matrix *D* shows the graph orientation corresponding to orientation σ(data flow).

Similarly, the matrix $\overline{D}$ can also be defined for the orientation $\bar{\sigma}$ (data dependences). Obviously, if "instruction *i* depends on instruction *j*" then "instruction *j* produces data for instruction *i*", and hence this means that $\overline{D} = D^{t}$.

Finally, it is easy to check that the following relation holds:

$$A = D + \overline{D}. \quad (13)$$

Recall that the matrix *A* considers adjacency and so it includes both incoming and outgoing ends on vertices.

2.5 Topological properties and ILP restrictions for *D*

One of the aims of Graph Theory algebra is to precisely determine how graphs properties are presented in the algebraic properties of their associated matrices. Additionally, we try to define properties in the scope of parallel instruction processing.

- **Vertex labeling used should not affect the properties of *D***. Matrix *D* can be associated to a directed graph with a vertex set $V = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$ whose labeling is arbitrary. Consequently, properties of matrix *D* should remain invariant upon permutations of rows and columns.

  A natural labeling for the vertex of the graph is using the strict precedence order of the instructions in a program. We will call this labeling a *programmatic labeling*.

- **There must exist 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.

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

  This is true even for loops. A loop is a compact way to write a code sequence which would correspond, in an expanded version to the repetition a sequence of operations but on a different data. Each iteration implies a new instance of the loop body but on new data. The execution of the body of a loop 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. Remember that since we apply our formalism in a dynamic way, we resort to the use of trace files where loops are naturally unrolled and branches are solved.

- **Data dependences are not symmetrical**. An instruction cannot depend on another which, also depends on the former, as this situation does not establish a precedence relation but a data dependence cycle. Consequently, matrix *D* is not symmetric. Mathematically: $d_{ij} \neq d_{ji} = 1$ when $0 \leq i \leq n - 1$ and $0 \leq j \leq n - 1$.

- **There is at least a graph vertex labeling under which matrix *D* is lower triangular**. Instructions only process data produced

by instructions located above in the program and, therefore, an instruction depends only on the precedent ones (principle of causality) so that at least one labeling must exist such that $d_{ij} = 0$ whenever $j > i$. These labelings are called *canonical labelings*. According to this, the programmatic labeling is an example of canonical labeling since it generates a lower triangular matrix D that will be denoted D_c.

2.6 The matrix *D* and the reduced valence

In this Section we will show how the matrix D is related to the reduced valence and thus is connected to the classical Graph Theory.

Proposition 3: The product $I^{\sigma} \cdot I^{\bar{\sigma}^t}$ generates matrix D.

$$D = I^{\sigma} \cdot I^{\bar{\sigma}^t}. \tag{14}$$

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

$$\left[I^{\sigma} \cdot \left(I^{\bar{\sigma}} \right)^t \right]_{ij} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\bar{\sigma}}. \tag{15}$$

The summation goes through all the arcs in index k. The product is different from zero only for the k-th arc if it enters the vertex i ($i_{ik}^{\sigma} = 1$) and leaves vertex j. In other words, it enters vertex j under the opposite orientation ($i_{jk}^{\bar{\sigma}} = 1$). But then, this is in agreement with the definition of data dependence matrix D that presented in (12).

$$\sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\bar{\sigma}} = d_{ij}. \tag{16}$$

And the proof is completed. □

Proposition 4: The count of arcs along the rows of matrix D generates the reduced valence matrix for the data flow graph orientation (orientation σ).

Proof: Suppose the relation is true and replace each entry of D by the value given in (16):

$$v_{ii} = \sum_{p=0}^{n-1} d_{ip} = \sum_{p=0}^{n-1} \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{pk}^{\bar{\sigma}}. \tag{17}$$

A simple computation leads to

$$v_{ii} = \sum_{k=0}^{m-1} \sum_{p=0}^{n-1} i_{ik}^{\sigma} \cdot i_{pk}^{\bar{\sigma}} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot \left(\sum_{p=0}^{n-1} i_{pk}^{\bar{\sigma}} \right). \tag{18}$$

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

$$v_{ii} = \sum_{k=0}^{m-1} i_{ik}^{\sigma}. \tag{19}$$

This corresponds to (8) and proves that obtaining the counting of incoming arcs is equivalent to run through the rows either of matrix I^{σ} or matrix D. Thus the proof is completed. □

Corollary: This fact allows us to provide a new definition for the reduced valence matrix:

$$v_{ij}^{\sigma} = \begin{cases} \sum_{k=0}^{n-1} d_{ik} = \text{number of arcs if } i = j; \\ 0, \text{ if } i \neq j. \end{cases} \tag{20}$$

2.7 Coupling and reuse vectors

For the sake of rising computational meaning, we can define the *coupling vector* $\vec{c}$ that projects the V^{σ} diagonal over just 1 dimension. Then, from its definition, it is easy to see that each component of the coupling vector $\vec{c}$ is the count of arcs along the rows of matrix D:

$$c_i = \sum_{k=0}^{n-1} d_{ik} \tag{21}$$

In the same way, working with the orientation $\bar{\sigma}$, we can define the *reuse vector* $\vec{r}$ that projects the $V^{\bar{\sigma}}$ diagonal over just 1 dimension. Again, it is immediate that each component of the reuse vector $\vec{r}$ is the count of arcs along the rows of matrix $\bar{D}$:

$$r_i = \sum_{k=0}^{n-1} \bar{d}_{ik} = \sum_{k=0}^{n-1} d_{ki} \tag{22}$$

where the previous expression comes from the fact that $\bar{D} = D^{\mathrm{t}}$.

In summary, $\vec{c}$ conveys the coupling information whereas $\vec{r}$ conveys the data reuse information.

2.8 Data dependence paths of length larger than 1

In the classical Graph Theory, given a directed graph $G(V, A)$, a path of length l from vertex v_i to v_j is defined as a finite sequence of $l + 1$ different vertices that begins in v_i and finishes in v_j, such that two consecutive vertices are an arc belonging to A [10, 5].

In the following we will show how the path lengths are related to some properties of the matrix D.

- ***D*^{l} represents the data dependence path of length *l* (arcs)**. Given a directed graph $G(V, A)$ with orientation σ, the number of data dependence paths of length l from v_i to v_j is the (i, j) entry in the matrix D^l.

  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 it 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, whenever the following holds:

  $$\left(d_{i0} \cdot d_{0j} \right) + \cdots + \left(d_{in-1} \cdot d_{n-1j} \right) = \sum_{k=0}^{n-1} \left(d_{ik} \cdot d_{kj} \right) \neq 0. \tag{23}$$

- 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. By induction we can extend the statement to paths of length l. Note that the length is measured in arcs.

- **The *n*-th power of *D* is null**. The maximum length of a data dependence path is $n - 1$ (arcs), n being the number of instructions in the code sequence. Hence, D^n will necessarily be null.

- **There are no cycles of dependences**. A graph representing a code sequence must be acyclic, otherwise an instruction would depend on itself through others and the task would not have solution in a finite number of steps. Algebraically, the diagonal of any power of the data dependence matrix (D^l) must be null: $d_{ii}^{l} = 0$ when $1 \le l \le n-1$ and $1 \le i \le n$.

2.9 Computation step concept

As soon as a data is available one or more instructions can proceed to execution, thus starting a new computation step. Based on this fact, we can define a computation step as the process of eliminating all the nodes in the graph with no incoming ends. Notice that in fact this is equivalent to execute all the independent instructions.

The code sequence finalizes when the graph vanishes. Obviously in the last computation step none of the remaining nodes has incomings ends.

So, while an arc linking instructions shows a static precedence relation, a computation step shows the dynamic transformation of the graph. There is a direct relation between data dependence path length and computation steps: if a data dependence path involves l vertices, then this path has $l - 1$ arcs and the minimum number of computation steps required to process the associated code sequence is l.

Notice that computation step is an asynchronous concept.

3. METRICS

Based on the theoretical model above presented, we are going to introduce a set of functions that allow us to measure magnitudes related both to the graph itself and to the computational environment represented by such graph. Among the former we will introduce the code coupling, the data reuse and the path length; and among the latter, the critical path length and the degree of parallelism.

3.1 Code coupling

Remember (see Section 2.3) that if we select the data flow graph orientation σ, the reduced valence gauges how much an instruction is coupled with the rest. The coupling 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, larger coupling implies a potentially greater partial ordering of the code, since there are more precedence relationships.

Accordingly, we define the *instruction coupling function* C as follows:

$$C(\vec{d}_i) = \sum_{k=0}^{n-1} d_{ik}\,, \tag{24}$$

where $\vec{d}_i$ is the i-th row in matrix D and holds the data dependence information for instruction i.

Let us consider any given data T (which resembles the word "token" used in data flow setting) which creates a coupling between the i-th instruction and other instructions. This coupling is manifested in vector $\vec{d}_i$.

If the dependence that T creates between two instructions is a true dependence (RAW: Read After Write), namely, when a read is performed in i after a write in j, then vector $\vec{d}_i$ will exhibit a single component with a 1 in the j-th position. Then, T contributes to the instruction coupling function $C(\vec{d}_i)$ with a value of just 1.

If the dependence through T is an output dependence (WAW: Write After Write), namely, there is a write in instruction i after another write in instruction j, then vector $\vec{d}_i$ will again exhibit a single component with a 1 in the j-th position. Then, in this second case, T contributes to the instruction coupling function $C(\vec{d}_i)$ also with a value of just 1.

However, if the coupling through T is an anti-dependence (WAR: Write After Read), namely, T is written in instruction i after one or more reads, then vector $\vec{d}_i$ may contain several components with a 1 since there are chances that previous instructions may also read T. For anti-dependences, then, T contributes to the instruction coupling function $C(\vec{d}_i)$ with a value potentially greater than 1.

Naturally, the coupling vector $\vec{c}$ (see Section 2.7) carries the code coupling information about each instruction. Notice that the component $c_i = C(\vec{d}_i)$.

From the instruction coupling function, we can introduce another measure C_T by simply summing all the components of vector $\vec{c}$. This accounts for the total coupling of a code sequence thus giving some information about the "entanglement" of the instructions.

Since the maximum number of data dependences (arcs) in the graph is given by all the possible ordered vertex pairs then the total coupling C_T is bounded by:

$$0 \le C_T \le \binom{n}{2}. \tag{25}$$

To obtain a coupling measurement independent of the amount of instructions in the sequence, we define a normalized coupling, C_{TN}, as the ratio C_T vs. the binomial coefficient n over 2. When C_{TN} is zero there is no dependence; in the opposite case, each instruction depends on all the precedent ones, and so C_{TN} is one. In other words $0 \le C_{TN} \le 1$.

3.2 Data reuse and life span

The reduced valence matrix diagonal for the orientation $\overline{\sigma}$ informs us about the reuse of data produced by each instruction. The reuse indicates how many instructions consume data produced by a given instruction. The larger the data reuse the potentially larger the temporary storage.

To quantify the data reuse, we define the *data reuse function* R as follows:

$$R(\vec{\bar{d}}_i) = \sum_{k=0}^{n-1} \bar{d}_{ik}\,, \tag{26}$$

When the reuse value is 1, it means that each generated data is consumed by only one instruction. If it is larger, it means that

several instructions consume a data. In this case, it is interesting to know the life span of the data.

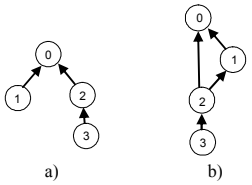

Figure 2. Different classes of data reuse.

In the Fig. 2.(a) and (b), two possible situations are illustrated. In both cases, instruction 0 generates data for instruction 1 and instruction 2. In the case (a) the data is consumed in the next computation step whereas in the case (b) 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 t_m of data T_m produced by instruction i and consumed by any other instruction from the set J must be at least equal to the longest data dependence path between the instruction i and any instruction j in J:

$$t_m = \max_{j \in J} \left\{ 0, k_j : \left[D^{k_j - 1} \right]_{ij} \neq 0, \left[D^{k_j} \right]_{ij} = 0 \right\}. \quad (27)$$

Notice that t_m is measured in computation steps.

3.3 Critical path length

As it is well known, the critical path length L is defined as the length of the longest data dependence path. In this Section we will show how we can derive L using the formalism of our theoretical model.

Remember that, given a graph of a code sequence, represented by its data dependence matrix D, D^l represents the data dependence paths of length l (see Section 2.8). Therefore, the first power of D that is identically zero indicates the length of the critical data path in computation steps:

$$L = l \text{ computation steps if and only if } D^{l-1} \neq \mathbf{0} \text{ and } D^l = \mathbf{0}. \quad (28)$$

With this metric, L is bounded as follows:

$$1 \leq L \leq n. \quad (29)$$

On the one hand, if L is 1: there are no data dependences among instructions and, should resources be available, all the instructions could be processed concurrently in just one computation step. On the other hand, if L is n, each computation step admits the issuing of just one instruction per computation step so the sequentiality is complete: n computation steps are required to process the code.

Remark that actually the life span t_m, as defined in Equation 27, can be considered simply as the critical path length of the subset J of instructions that consume the data produced by instruction i.

3.4 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. The parallelism degree is inversely related to the critical path length: the longer the length, the stricter the partial ordering of the code sequence, limiting the ability of concurrent processing.

Consequently, we define the *parallelism degree*, G_p, as:

$$G_p = \frac{n}{L}. \quad (30)$$

G_p ranges from 1 instruction per computation step (absence of parallelism) to n (maximum parallelism degree).

$$G_p \in [1, n]. \quad (31)$$

3.5 Calculation of the critical path length

According to Equation (28), the calculation of the successive powers of the matrix D allows the determination of the critical path length. Nevertheless, considering the worst case, the method has a very heavy complexity ($O(n^4)$ product operations) which is too high for practical purposes.

In this Section we are going to present a method that decreases the complexity.

In the same way as the coupling vector $\vec{c}$ was derived from matrix D (see Section 2.7), we can extend the definition to any power of D so that $\vec{c}^{\,i}$ will be the coupling vector associated to D^i, thus showing the coupling trough data dependence path of length i. We denote $\vec{c}^{\,1} = \vec{c}$.

Proposition 5: The following relation holds:

$$D \cdot \vec{c}^{\,i} = \vec{c}^{\,i+1}. \quad (32)$$

Proof: Consider first the case $i = 1$.

$$D \cdot \vec{c} = \sum_{j=0}^{n-1} d_{ij} \cdot c_j = \sum_{j=0}^{n-1} d_{ij} \sum_{k=0}^{n-1} d_{jk} = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} d_{ij} \cdot d_{jk} =$$
$$= \sum_{k=0}^{n-1} \left[D^2 \right]_{ik} = \vec{c}^{\,2}.$$

Consider now the general case:

$$D \cdot \vec{c}^{\,i} = \sum_{j=0}^{n-1} d_{ij} \cdot c_j^i = \sum_{j=0}^{n-1} d_{ij} \sum_{k=0}^{n-1} \left[D^i \right]_{jk} = \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} d_{ij} \cdot \left[D^i \right]_{jk} =$$
$$= \sum_{k=0}^{n-1} \left[D^{i+1} \right]_{ik} = \vec{c}^{\,i+1}.$$

Thus the proof is completed. □

Proposition 6: The coupling vector $\vec{c}^{\,i}$ is the null vector if and only if the matrix D^i is null.

Proof: Assume that D^i is null, then by definition, $\vec{c}^{\,i}$ is also null. Conversely, assume that $\vec{c}^{\,i}$ is null; since all the entries in matrix D^i are non negative numbers then it follows that D^i is null. □

Corollary: From the propositions above it follows:

$$L = l \text{ computation steps if and only if } \vec{c}^{\,l-1} \neq \vec{0} \text{ and } \vec{c}^{\,l} = \vec{0}. \quad (33)$$

The calculation of $\vec{c}^{\,i+1}$ from $\vec{c}^{\,i}$ has a complexity of $O(n^2)$ and hence the complexity of the method displays a complexity of $O(n^3)$ in the worst case.

4. CONCLUSIONS

In this paper, a model of analysis, based on Graph Theory, appropriate to quantify the data coupling has been proposed. It can be applicable to several layer of the computation process. For instance, we can determine with its help the parallelism degradation due to the compilation process or the availability of instruction level parallelism after the impact of using a particular instruction set architecture.

The topological properties and restrictions the matrix *D* must comply with in the ILP scope have been identified along with a method that uses the same matrix *D* to quantify the parallelism degree of code, the data reuse, and their life span. A metric to measure the available parallelism degree has been defined as well.

As a summary, we enumerate the main contributions that have been made throughout the paper:

- we introduce the data dependence matrix *D* from the Graph Theory definitions and supported by the novel concept of the reduced valence,
- we have identified several topological properties and restrictions that the matrix *D* must satisfy in the instruction parallel processing scope,
- we have determined a relation between the matrix *D* and the adjacency matrix *A*,
- we have proved the relation between the matrix *D* and the reduced valence matrix for an orientation,
- we have introduced the concept of code coupling to measure the ordering degree of a code sequence,
- we have established a way to quantify the data reuse degree,
- we have identified the relation between the powers of *D* and the data dependence paths of length longer than 1,
- we have proposed a method to calculate the critical path length.

5. ACKNOWLEDGMENTS

This work was partially supported by the Vicerrectorado de Investigación de la Universidad de Alcalá under Grant UAH PI2005/072.

6. REFERENCES

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

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

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

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

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

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

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

[8] Dennis, J. B. Concurrency in software systems. In *Advanced Course in Software Engineering*, Springer-Verlag, 1973, 111 – 127.

[9] Durán, R., and Rico, R. Quantification of ISA Impact on Superscalar Processing. In *Proceeding of EUROCON2005*, November 2005, 701 – 704.

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

[11] Kumar, M. Measuring parallelism in computation intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9), 1988.

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

[13] Postiff, M. A., Greene, D. A., Tyson, G. S. 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.

[14] Rico, R., Pérez, J. I., and Frutos, J. A. The impact of x86 instruction set architecture on superscalar processing. *Journal of Systems Architecture*, vol. 51-1, 2005.

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

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

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

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