Quantification of ISA Impact on Superscalar Processing

Raúl Durán, Rafael Rico, *Affiliate Computer Society, IEEE*

Abstract—**The differences found between the superscalar performance in x86 and non-x86 processors and the peculiar characteristics of the x86 ISA 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.**

We construct the matrix representation associated to the data dependence graph of execution traces. In this paper, we explain how this matrix characterizes the corresponding code in a mathematical manner, fulfills a number of properties and restrictions, and provides information about the ability of the code to be processed concurrently. Besides, we also show how different data dependence sources can be composed, thus providing a mechanism to explore their final influence on the parallelism degree. These techniques are applied to an example from which some conclusions are derived.

Keywords—**Instruction set design, graph theory.**

I. Introduction: New Challenges in Computer Architecture Evaluation

Quantitative evaluation is a crucial point in the computer architecture research. The simulation has become the first evaluation tool both in industry and research [21]. Unfortunately, the construction of good simulators and the selection of appropriate workloads have become appalling tasks. This scenario has driven the research to just those fields where quality proven tools are available, thus preempting other important research topics.

As is common in other research fields, the mathematical formalization facilitates the description of phenomena, allows predicting behaviors, supports reproductibility and simplifies the knowledge transference. However, in ILP research, the simulation is the most frequently used evaluation technique.

Here we propose and verify an analytical model for the quantitative evaluation of ILP that permits to study the behavior of architectural proposals in the superscalar setting. Besides, our analytical model could shed light on other aspects not covered by simulation.

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

The authors are with the Department of Computer Engineering, Universidad de Alcalá, 28871 Alcalá de Henares, Spain.
E-mail: {raul.duran, rafael.rico}@uah.es.

A. Applying graph theory to fine-grain parallelism analysis

Graph theory provides an efficient mathematical formalization that promises to be very useful for the analytical modeling of ILP. Moreover, graphs have already been successfully applied to the study of other aspects of computation: medium- and coarse-grain parallelism extracted by compilers [3, 17, 28], data structures [2, 3, 7], software description [8, 9], and so on.

B. ILP difference between x86 and non-x86 processors

The quantification of ILP is one of the most popular subjects in computer architecture. In the literature, numerous studies can be found identifying limiting factors, quantifying their effects, providing possible solutions and evaluating the results [18, 19, 25, 26, 27]. All these works have in common that they present non-x86processors and the reported IPC average results are in the range 2.5-15, peaking around 30 IPC.

Works using x86 instruction set processors are less frequent. In those cases, the reported parallelism is not so good. In [5] the CPI for SPECint95 is in the range 0.75 to 1.6. And in [16, 23] IPC values are in the range 0.5 - 3.5 in the best situations.

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 ISA and the superscalar model

For the sake of binary compatibility with previous processors the x86 ISA has inherited design characteristic suitable to older requirements but clearly harmful in the scope of superscalar processing, such as dedicated register usage, implicit operands, condition codes and so on (see reference [20] for an in-depth explanation).

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

The ISA 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.

D. Metrics

IPC is by far the metric most often employed in parallelism quantification at the instruction level.

A much less used metric consists of measuring the critical path length of a code sequence. This has been previously employed in several works: In [15] it is used at the program layer and in [4, 18, 24] it is used to evaluate characteristics of the physical layer.

We propose an alternate measurement method based on the data dependence graphs (DDG). It consists of building the DDG of a real machine code sequence.

It is interesting to remark that our proposal of DDG-based parallelism quantification is independent of the physical implementation, since it is located in a previous step of the computation process, namely, in the machine language layer.

II. Representation of Instructions Sequences by Graphs.

We define the data dependence matrix D as:

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

Let $\vec{d}_i$ be the vector carrying the data dependence information for an instruction i. Then, the rows of the matrix D are the vectors $\vec{d}_i$ of a code sequence.

Notice that the matrix D represents the direct data dependence path or data dependence path of length 1, that is, instruction i consumes a data processed directly by instruction j with no interveners.

A. Topological properties and ILP restrictions for D

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, information in the scope of parallel instruction processing.

- **The vertex labelling should not affect the properties of *D***. The 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 labelling is arbitrary. Consequently, the properties of matrix D should be invariant under permutations of rows and columns.

The natural vertex labelling of the graph is the one induced by the strictly precedence order in which they are written in the program (*programmatic labelling*).

- **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 cannot have the same instruction as source and as destination. Consequently, the matrix D has null diagonal. That is,

$$d_{ii} = 0 \qquad 0 \le i \le n-1.$$

- **The data dependences 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:

$$d_{ij} \neq d_{ji} = 1 \qquad 0 \le i \le n-1 \qquad 0 \le j \le n-1.$$

- **There is a graph vertex labelling under which the matrix *D* is lower triangular**. Instructions only process data given by instructions written above in the program and, therefore, an instruction depends only on the precedent ones (principle of causality). According to this, the programmatic labelling generates a lower triangular matrix D because $d_{ij} = 0$ whenever $j > i$. The matrix D will be termed *canonical* when it is lower triangular and will be denoted D_c.

B. Code coupling

If an instruction consumes data coming from several instructions it must stall its own execution till all these data are available and therefore it is coupled to them. A larger coupling implies a potentially greater partial ordering of the code, since there are more precedence relationships. We define coupling C as $C = \sum_{i=0}^{n-1}\sum_{k=0}^{n-1} d_{ik}$.

When using this tool in automatic code analysis, computational effort can be saved with the following equivalent expression $C = \sum_{i=1}^{n-1}\sum_{k=0}^{i-1} d_{c_{ik}}$, which is based on the matrix D_c.

The maximum number of data dependences in the graph is given by all the possible ordered vertex pairs. Hence, the coupling C is bounded by $0 \le C \le \binom{n}{2}$.

To obtain a coupling measurement 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 $0 \le C_N \le (n-1)/2$.

C. Data reuse

The life span of a data produced by instruction i and consumed by any other 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 $t_{i_{min}}$:

$$t_{i_{min}} = \max_{1 \le j \le n} \left\{ 0, k_j : \left[D^{k_j}\right]_{ij} \neq 0, \left[D^{k_j+1}\right]_{ij} = 0 \right\}. \quad (2)$$

D. Data dependence paths of length larger than 1

A path of length l from vertex v_i to v_j is 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 in the graph [6, 11].

- **D^l represents the data dependence path of length l (arcs)**. 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.
- **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 be necessarily 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:

$$d^l_{ii} = 0, \qquad 1 \le l \le n \quad 1, \qquad 1 \le i \le n.$$

E. Critical path length and degree of parallelism

Given a code sequence, represented by its data dependence matrix D, we define the *critical path length* L as the length of the longest data dependence path.

The first power of D that is identically zero indicates the length of the critical data path in computation steps:

$L = l$ computation steps if and only if $D^l = \mathbf{0}$. (3)

With this metric, L is bounded as $1 \le L \le n$. We define the *normalized critical data path length*, L_N, as $L_N = L/n$.

When L_N approaches 1 there is no parallelism, and the nearer to 0, the more the parallelism the code bears. It is clear that $L_N \in (0,1]$. We define the *parallelism degree*, G_p, as the reciprocal of L_N. Obviously, $G_p \in [1,n]$.

III. Dependence sources composition

Given a code sequence we can consider several sources of data dependences among its instructions. Each dependence source gives rise to a data dependence matrix D_s.

We define the law of composition for the matrices D_s as:

$D = D_{s1}$ OR D_{s2} OR ... OR D_{sn}. (4)

Obviously, the final dependence map represented by D is the composition of all sources. We thus have a method to study the impact superscalar setting of any combination of different dependence sources.

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}).

In particular we are going to study the critical path length of the resulting matrix as a function of its components. The following bound holds:

$$\max_i\{L_{si}\} \le L \le \min\left\{\sum_i L_{si}, n\right\}. \quad (5)$$

A. Illustrative example

The previous results provide valuable insight into the ILP offered by an ISA and thus the suitability of the processor to the task of superscalar processing.

In particular, an example is proposed based on an x86 code sequence from which useful information is derived. Besides, the composition of the different data dependence sources is illustrated.

In Table 1 we show 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 are set apart from the implicit operands. Each one of these categories represents a possible data dependence source whose impact we can study separately.

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

code sequence		Read operands								Written 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	-	-	-	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
15:	JNE/JNZ IP+F7	-	-	-	ZF	-	-	-	-	-	-	-	-	IP	-	-	-

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 dependences: true dependences, antidependences and output dependences.

The critical path length L has been computed for each matrix. The accumulative composition of each dependence source has been evaluated. The successive compositions follow this order: First true dependences, then the antidependences, and then the output dependences. Within each basic type, the composition begins with the dependences due to explicit operands and then those due to the implicit operands. Finally, for the operand functionality: first, the dependences due to accesses to operands with a greater computational meaning; next the dependences due to memory address computation; next, the dependences due to operands related to the stack access; and last the dependences due to state register accesses.

The results are plotted in Fig. 1. The light gray columns show the critical path length for each dependence source. The dark gray columns show the critical path length of the composition of all dependence sources located to the left.

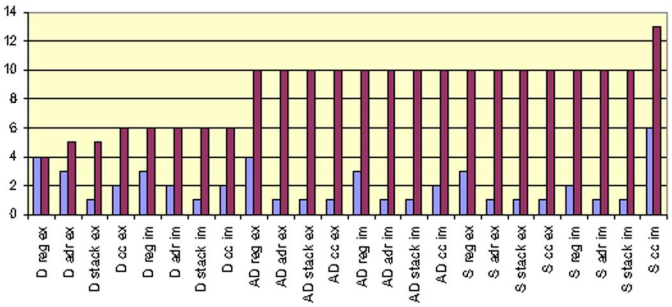

Fig. 1. Critical path length for both each data dependence source and for the composition.

From the example some immediate consequences can be extracted. As for the true dependences we find two sources with remarkable impact on the parallelism degradation:

- memory address computing; and
- condition codes.

As for the dependences due to physical resources limitations, the most remarkable sources seem to be:

- explicit use registers in antidependences; and
- condition codes in output dependences.

For the considered example, the values for the parameters defined in this paper are the following:

$$L = 13 \text{ computation steps}$$
$$L_N = 0.81 \text{ computation steps/instruction}$$
$$G_p = 1.23 \text{ instructions/computation step}$$
$$C = 38 \quad C_N = 2.37$$

The life span for each data dependence source is 1 except for the antidependences due to explicit data registers (which is 1.33) and for the antidependences due to implicit data registers (which is 1.4). However the life span for the composition of all dependence sources shown in D soars to 4.5.

IV. Conclusions and Future Work

A model of analysis applicable to the computation process has been proposed. When applied at the machine language layer, it allows the quantitative evaluation of the impact of both the ISA and the compilation procedure itself on availability of instruction level parallelism.

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 parallelism degree of code, the data reuse and their life span. A metric to measure 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 knowledge 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 ISA and valuable information has been obtained.

In future works, the contribution of each one of the dependence sources should be studied, analyzing its behavior on a sufficiently large set of test programs. It seems also possible to extend our development to also model the specifications of the physical layer and the processes of allocation-scheduling.

References

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

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

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

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

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

[12] 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).

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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